

Go To



Visual C++ 6

**Frank Heimann
Nino Turianskyj**

Go To



Visual C++ 6

eBook

Die nicht autorisierte Weitergabe dieses eBooks
ist eine Verletzung des Urheberrechts!



ADDISON-WESLEY

An imprint of Addison Wesley Longman, Inc.

Bonn • Reading, Massachusetts • Menlo Park, California
New York • Harlow, England • Don Mills, Ontario
Sydney • Mexico City • Madrid • Amsterdam

Deutsche Bibliothek – CIP-Einheitsaufnahme

Go To Visual C++ 6/Heimann, Frank/Turianskyj, Nino. –
Bonn: Addison-Wesley-Longman, 1999
ISBN 3-8273-1468-2

Copyright © 1999 Addison Wesley Longman Verlag GmbH

Lektorat Tomas Wehren und Judith Stevens

Korrektur Daniela Haralambie, Bonn

Layout Katja Lehmeier

Produktion Michael Schack, Leipzig

Satz Reemers EDV-Satz, Krefeld – gesetzt aus der StoneSerif mit FrameMaker

*Belichtung,
Druck & Bindung* Bercker Graphischer Betrieb, Kevelaer

Umschlaggrafik Barbara Thoben, Köln

Illustration Stefan Leowald, Köln

Das verwendete Papier ist aus chlorfrei gebleichten Rohstoffen hergestellt und alterungsbeständig. Die Produktion erfolgt mit Hilfe umweltschonender Technologien und unter strengsten Auflagen in einem geschlossenen Wasserkreislauf unter Wiederverwertung unbedruckter, zurückgeführter Papiere.

Text, Abbildungen und Programme wurden mit größter Sorgfalt erarbeitet. Verlag, Übersetzer und Autoren können jedoch für eventuell verbliebene fehlerhafte Angaben und deren Folgen weder eine juristische noch irgendeine Haftung übernehmen.

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Kein Teil dieses Buches darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form durch Fotokopie, Mikrofilm oder andere Verfahren reproduziert oder in eine für Maschinen, insbesondere Datenverarbeitungsanlagen, verwendbare Sprache übertragen werden. Auch die Rechte der Wiedergabe durch Vortrag, Funk und Fernsehen sind vorbehalten.

Die in diesem Buch erwähnten Soft- und Hardwarebezeichnungen sind in den meisten Fällen auch eingetragene Marken und unterliegen als solche den gesetzlichen Bestimmungen.



Inhaltsverzeichnis

	Vorwort	15
Teil I	Grundlagen	17
1	Einleitung	19
1.1	Es geht los	20
1.1.1	Die Icons in diesem Buch	20
1.1.2	Was das Buch erklärt	20
1.1.3	Was das Buch nicht erklärt	22
1.2	Aufbau des Buches	23
1.3	Hard- und Software-Voraussetzungen	24
1.4	Installation von Visual C++	25
1.5	Grundlagen des Developer Studios	32
1.5.1	Integration in Windows	32
1.5.2	Philosophie	34
1.5.3	Aufbau des Developer Studios	35
1.6	Der Entwicklungszyklus	40
1.6.1	Projekte	40
1.6.2	Editieren von Quelltexten	50
1.6.3	Kompilieren und Linken	60
1.7	Dokumentation und Online-Hilfe	64
1.7.1	Gedruckte Handbücher	64
1.7.2	Online-Dokumentation	64
1.7.3	Weitere Informationsquellen	67
1.7.4	Referenzen auf elektronische Dokumente	67
1.7.5	Quelltexte zur MFC	67
2	Hello – Das erste Programm	69
2.1	Ein kurzer Abriß in (Visual) C++	70
2.1.1	Begriffe und Definitionen	70
2.1.2	Aufbau eines MFC-Programms	72
2.1.3	Das Message-Map-Konzept	73

2.2	Die Klasse CWinApp	75
2.2.1	Ableiten einer Anwendungsklasse von CWinApp	75
2.2.2	Überlagern der Methode InitApplication	78
2.2.3	Überlagern der Methode ExitInstance	79
2.2.4	Globale Funktionen	79
2.2.5	Das Programm-Icon	81
2.2.6	Der Programmcursor	83
2.2.7	Daten-Member in CWinApp	85
2.2.8	Auswerten der Kommandozeile	85
2.2.9	Multitasking in eigenen Programmen	88
2.3	Die Klasse CWnd	95
2.3.1	Fensterklassen	95
2.3.2	Erzeugen eines neuen Fensters	97
2.3.3	Der Konstruktor von CMainWindow	98
2.3.4	Überlagern der OnPaint-Methode	100
2.3.5	Erzeugen der Message-Map	103
2.3.6	Zerstören eines Fensters	105
2.4	Das Beispiel Schritt für Schritt	106
2.4.1	Das Anlegen des Hello-Projekts	106
2.4.2	Hello – Schritt 1	109
2.4.3	Hello – Schritt 2	111
2.4.4	Hello – Schritt 3	112
2.4.5	Hello – Schritt 4	113

Teil II Mit den Assistenten zur Anwendung 115

3 Die Assistenten 117

3.1	Der Anwendungs-Assistent	118
3.1.1	Generieren eines Projekts	119
3.1.2	Vom Anwendungs-Assistenten erzeugte Dateien	126
3.1.3	Test des neuen Programms	128
3.1.4	Neuerungen in InitInstance	128
3.1.5	Initialisierungen in CMainFrame	131
3.2	Der Klassen-Assistent	132
3.2.1	Anlegen von Klassen, Member-Funktionen und Member-Variablen	132
3.2.2	Assistentenleiste	135
3.2.3	Anlegen von Funktionen und Variablen über die Klassen-Ansicht	137

4 Ressourcen 141

4.1	Menüs	142
4.1.1	Das Hauptmenü	142
4.1.2	Bedienung des Menü-Editors	143
4.2	Verändern von Menüpunkten	153
4.2.1	Die Klasse CMenu	154
4.2.2	Temporäre Objekte	155
4.2.3	Menüpunkte aktivieren und deaktivieren	156
4.2.4	Menüpunkte markieren	159

4.3	Dynamisch erzeugte Menüs	161
4.3.1	Menüpunkte zur Laufzeit löschen oder einfügen	161
4.3.2	Erzeugen eines Kontext-Menüs	165
4.3.3	Verändern des Systemmenüs	169
4.4	Beschleunigtasten	171
4.4.1	Anlegen einer Beschleuniger-Ressource	172
4.4.2	Integration einer Beschleuniger-Ressource	173
4.4.3	Anlegen versteckter Beschleuniger	174
5	Einfache Dialoge	177
5.1	Der Dialog Info über ...	178
5.2	Beenden des Programms	180
5.2.1	Beenden von Windows	182
5.3	Standard-Dialoge	183
5.3.1	Die verschiedenen standardisierten Dialoge	184
5.3.2	Vorgehensweise bei der Programmierung	185
5.3.3	Erweiterungsmöglichkeiten	186
5.4	Der Dialog Datei öffnen in SuchText	187
5.5	Der Suchen-Dialog	190
5.5.1	Implementierung in das Programm	191
5.5.2	Nachrichtenbehandlung des Dialogs	193
5.5.3	Implementierung von OnSuchen	194
5.6	Verändern der Programm-Titelleiste	197
6	Ausgaben in Fenstern	199
6.1	Die OnPaint/OnDraw-Routine	200
6.1.1	Die Klasse CPaintDC	202
6.1.2	Der Arbeitsbereich des Fensters	203
6.1.3	Regionen im Device-Kontext	205
6.2	Schrift- und Hintergrundfarbe	208
6.2.1	Einstellen der Schriftfarbe	208
6.2.2	Einstellen der Hintergrundfarbe	210
6.2.3	Verwenden von Systemfarben	211
6.3	Schriftattribute	212
6.3.1	Merkmale von Schriftarten	212
6.3.2	Vordefinierte Schriftarten	214
6.3.3	Erzeugen eigener Schriftarten	217
6.4	Ausgeben des Textes	219
6.4.1	Die Datenstruktur zur Darstellung des Textes	219
6.4.2	Größenmerkmale der Schrift	220
6.4.3	Textausrichtung	222
6.4.4	Die Textausgabe mit ExtTextOut	223
6.4.5	Einfachere Formen der Textausgabe	225
6.5	Bildschirm Ausgaben außerhalb von OnPaint	227

7	Bildlaufleisten und Scrollen	229
7.1	Anlegen der Bildlaufleisten	230
7.2	Initialisieren der Bildlaufleisten	233
7.3	WM_HSCROLL und WM_VSCROLL	234
7.4	Reaktion des Programms auf Scroll-Nachrichten	236
7.5	Beispielprogramm	239
8	Fehlersuche mit dem Browser	241
8.1	Arbeitsweise	243
8.2	Das Hauptfenster des Browsers	244
8.3	Vererbungshierarchie darstellen	245
8.3.1	Abgeleitete Klassen	245
8.3.2	Basisklassen	246
8.4	Aufrufabhängigkeiten darstellen	246
8.4.1	Aufgerufene Funktionen	246
8.4.2	Aufrufende Funktionen	247
8.5	Definitionen und Referenzen anzeigen	248
8.6	Integration des Browsers in das Developer Studio	249
9	Der integrierte Debugger	251
9.1	Haltepunkte (Breakpoints)	253
9.1.1	Standard-Haltepunkte	253
9.1.2	Komplexere Breakpoints	254
9.1.3	Haltepunkte im Register »Pfad«	255
9.1.4	Haltepunkte im Register »Daten«	255
9.1.5	Haltepunkte im Register »Nachrichten«	257
9.2	Programmausführung	257
9.3	Programmvariablen	258
9.3.1	Das Variablen-Fenster	258
9.3.2	Das Fenster Überwachung	259
9.3.3	Das QuickWatch-Fenster	260
9.4	Verschiedenes	261
9.4.1	Die Aufrufliste	261
9.4.2	Das Register-Fenster	261
9.4.3	Das Speicher-Fenster	261
9.4.4	Assembler-Listings einblenden	261
9.4.5	Ausnahmen	261
9.4.6	Threads	262
9.4.7	Module	262
9.5	Weitere Debugging-Möglichkeiten	263
9.5.1	TRACE-Makro	263
9.5.2	ASSERT-Makro	263
9.5.3	Memory Leaks	264

Teil III Die Benutzerschnittstellen 265**10 Steuerungen 267**

10.1	Buttons (Schaltflächen)	270
10.1.1	Anwendungen	270
10.1.2	Attribute	273
10.1.3	Benachrichtigungscodes	279
10.1.4	Die Klasse CButton	283
10.2	Textfelder	288
10.2.1	Anwendungen	288
10.2.2	Attribute	289
10.2.3	Benachrichtigungscodes	292
10.2.4	Die Klasse CEdit	295
10.2.5	Die Klasse CStatic	303
10.3	Listenfelder und Kombinationsfelder	306
10.3.1	Listenfelder (Listboxen)	306
10.3.2	Kombinationsfelder (Comboboxen)	307
10.3.3	Attribute	309
10.3.4	Benachrichtigungscodes	312
10.3.5	Die Klassen CComboBox und CListBox	316
10.4	Bildlaufleiste/Schieberegler	323
10.4.1	Anwendungen	323
10.4.2	Attribute	324
10.4.3	Benachrichtigungscodes	324
10.4.4	Die Klasse CScrollBar	326
10.5	Regler (Slider)	328
10.5.1	Attribute	328
10.5.2	Benachrichtigungscodes	329
10.5.3	Die Klasse CSliderCtrl	329
10.6	Strukturansicht (Tree Control)	331
10.6.1	Attribute	331
10.6.2	Benachrichtigungscodes	332
10.6.3	Die Struktur TV_INSERTSTRUCT	332
10.6.4	Anlegen einer Struktur	334
10.6.5	Die Klasse CTreeCtrl	336
10.7	Drehfeld (Spin Button Control)	337
10.7.1	Attribute	337
10.7.2	Benachrichtigungscodes	338
10.7.3	Die Klasse CSpinButtonCtrl	339
10.8	Zugriffstaste (Hotkey Control)	339
10.8.1	Attribute	340
10.8.2	Benachrichtigungscodes	340
10.8.3	Die Klasse CHotKeyCtrl	340
10.9	Statusanzeige (Progress Control)	341
10.10	Eigenschaftfenster (Property Sheets)	342
10.10.1	Anlegen der Dialog-Ressourcen	342
10.10.2	Anlegen der Property Pages	343
10.10.3	Anlegen des Property Sheets	344

10.11	Registerkarte (Tab Control)	346
10.11.1	Attribute	346
10.11.2	Benachrichtigungscodes	347
10.11.3	Die Struktur TC_ITEM	348
10.11.4	Die Klasse CTabCtrl	349
10.11.5	Beispielprogramm	351
10.12	Listenelement (List Control)	354
10.12.1	Anwendungen	354
10.12.2	Attribute	355
10.12.3	Benachrichtigungscodes	356
10.12.4	Die Klasse CListControl	357
10.12.5	Beispielprogramm	362
10.13	Animation (Animate Control)	366
10.13.1	Anwendung	366
10.13.2	Attribute	367
10.13.3	Die Klasse CAnimateCtrl	367
10.13.4	Beispielprogramm	368
10.14	Datum/Uhrzeit-Steuerung	368
10.14.1	Anwendung	368
10.14.2	Attribute	369
10.14.3	Benachrichtigungscodes	370
10.14.4	Die Klasse CDateTimeCtrl	370
10.14.5	Beispielprogramm	375
10.15	Die Monatskalender-Steuerung	378
10.15.1	Anwendung	378
10.15.2	Attribute	378
10.15.3	Benachrichtigungscodes	379
10.15.4	Die Klasse CMonthCalCtrl	380
10.15.5	Beispielprogramm	384
10.16	Die IP-Adressensteuerung	390
10.16.1	Anwendung	390
10.16.2	Attribute	390
10.16.3	Benachrichtigungscodes	390
10.16.4	Die Klasse CIPAddressCtrl	391
10.16.5	Beispielprogramm	392
10.17	Das erweiterte Kombinationsfeld-Steuerelement	394
10.17.1	Anwendung	394
10.17.2	Attribute	394
10.17.3	Benachrichtigungscodes	395
10.17.4	Die Klasse CComboBoxEx	396
10.17.5	Beispielprogramm	398
10.18	ActiveX-Controls	400
10.18.1	Anwendung	400
10.18.2	Ablauf	400
10.18.3	Attribute	402
10.18.4	Beispielprogramm	402

11	Weitere MFC-Klassen	405
11.1	Die Klasse CString	406
11.1.1	Allgemeines	406
11.1.2	Anlegen von CString-Objekten	406
11.1.3	Zugriff auf Zeichen	408
11.1.4	Zuweisung und Vergleich	408
11.1.5	Extrahieren von Teilzeichenketten und weitere Konvertierungen	409
11.1.6	Suchen von Teilzeichenketten	410
11.2	Die Klasse CObArray	410
11.2.1	Die Familie von Feldern	410
11.2.2	Anlegen und Zugriff	410
11.3	Die Klasse CImageList	412
11.3.1	Allgemeines	412
11.3.2	Anlegen einer Image-Liste	412
11.3.3	Hinzufügen und Löschen von Bildern	414
11.4	Die Symbolleiste	415
11.4.1	Allgemeines	415
11.4.2	Standard-Code vom Anwendungs-Assistenten	415
11.4.3	Laden und Speichern der Einstellungen	417
11.4.4	Mehrere Symbolleisten	419
11.4.5	Text-Label	419
11.4.6	Drop-down-Listen	420
11.5	Die Klasse CFile	422
11.5.1	Grundlagen	422
11.5.2	Hierarchie und abgeleitete Klassen	423
11.5.3	Die wichtigsten Methoden	424
11.5.4	Ausnahmebehandlung	428
11.5.5	Ein Beispiel	430
Teil IV	Die drei Formen der MFC-Anwendung	437
12	Dialogfelder und die Klasse CDialog	439
12.1	Grundlagen der Programmierung	440
12.2	Dialogvorlagen mit dem Dialogeditor	442
12.2.1	Bedienelemente des Dialogeditors	442
12.2.2	Festlegen der Tabulator-Reihenfolge	450
12.2.3	Test der Dialogvorlage	452
12.3	Dialoge ohne eigene Klasse	452
12.4	Ableiten eigener Dialogfeldklassen	455
12.4.1	Eigenschaften eigener Dialogfeldklassen	455
12.4.2	Dialogfeld-Klasse mit dem Klassen-Assistenten	459
12.5	Dialogbasierende Anwendung - ein Beispiel	462
12.5.1	Vorteile und Besonderheiten	462
12.5.2	Aufgabe	466
12.5.3	Bedienung	467
12.5.4	Implementierung	468

12.5.5	Projektanlage mit dem MFC-Anwendungs-Assistenten	468
12.5.6	Gestaltung der Dialogvorlage	470
12.5.7	Zusätzlicher Programmcode	471
12.6	Sonstiges	483
12.7	Nicht-modale Dialoge	484
12.7.1	Unterschiede zu modalen Dialogfeldern	484
12.7.2	Entwurf der Dialogvorlage	485
12.7.3	Programmierung	486
12.7.4	Anwendungsbeispiele	488
13	Dokument-Ansichten-Architektur	495
13.1	Dokument-Ansicht-Paradigma der MFC	496
13.2	Die Klasse CDocument	497
13.2.1	Basis aller MFC-Dokumente	497
13.2.2	Zugriff auf die Ansichtsklassen	498
13.2.3	Verwaltung der Daten	499
13.2.4	Die Registrierung der Dokumente	505
13.3	Die Klasse CView	507
13.3.1	Basis aller MFC-Ansichten	507
13.3.2	Zugriff auf das Dokument	508
13.3.3	Initialisierung	509
13.3.4	Ausgabemöglichkeiten	510
13.3.5	Nachrichtenbearbeitung mit Klassen-Assistenten	516
13.4	Die Klasse CDocTemplate	517
13.5	SDI-basierende Anwendung - ein Beispiel	519
13.5.1	Besonderheiten einer SDI-Anwendung	519
13.5.2	Aufgabe	520
13.5.3	Bedienung	521
13.5.4	Implementierung	522
13.5.5	Projektanlage mit dem MFC-Anwendungs-Assistenten	523
13.5.6	Anpassung der Ressourcen	524
13.5.7	Zusätzlicher Programmcode	528
13.6	Die Klasse CFormView	549
13.6.1	Verwendung	549
13.6.2	Formularvorlagen formatierter Ansichten	549
13.6.3	Der Einsatz in der Anwendung	550
13.7	Die Klasse CSplitterWnd	551
13.8	MDI-basierende Anwendung - ein Beispiel	555
13.8.1	Besonderheiten einer MDI-Anwendung	555
13.8.2	Aufgabe	557
13.8.3	Implementation	558
13.8.4	Projektanlage mit dem MFC-Anwendungs-Assistenten	558
13.8.5	Erweitern und Anpassen der Ressourcen	561
13.8.6	Zusätzlicher Programmcode	564

14	Die integrierte Hilfe	585
14.1	Der Einsatz von WinHelp	586
14.1.1	Allgemeines	586
14.1.2	Anwendungs-Assistent und die Hilfe	586
14.1.3	Aufbau und Ablauf des Hilfe-Prozesses	588
14.1.4	Änderungen und Anpassungen der Hilfe-Dateien	589
14.1.5	Der Einsatz in der Anwendung	590
14.2	HTML-Help	593
14.2.1	Grundlagen und Voraussetzungen	593
14.2.2	Anpassungen im Projekt	595
14.2.3	Der Aufruf des Help Viewers in der Anwendung	596
Teil V	Datenbankunterstützung	601
15	MFC Datenbankschnittstellen	603
15.1	Grundlagen	604
15.1.1	Elementares	604
15.1.2	Konzept	604
15.1.3	Voraussetzungen	606
15.2	Unterschied zur Serialisation	606
15.3	Die Klasse CDatabase	608
15.4	Die Klasse CRecordset	610
15.5	Die Klasse CRecordView	613
15.6	ODBC-Konfiguration	614
15.7	Assistentenunterstützung	616
15.7.1	Anwendungs-Assistent	616
15.7.2	Klassenassistent	618
15.8	Datenbank-Ausnahmebehandlung der MFC	619
15.9	ODBC-Datenbankanwendung - ein Beispiel	620
15.9.1	Vorteile und Besonderheiten	620
15.9.2	Aufgabe und Implementation	621
15.9.3	Projektanlage mit dem Anwendungs-Assistenten	622
15.9.4	Erweitern und Anpassen der Ressourcen	628
15.9.5	Zusätzlicher Programmcode	630
15.9.6	Fazit und Anregungen	641
15.10	Erweiterte SQL-Funktionen	641
Anhang A	Windows-Zeichensatz	643
Anhang B	Deutsche und englische Begriffe	644
Anhang C	Deutsche und englische Menüs	647
Anhang D	Symbolleisten von Visual C++ 6	653
Anhang E	MFC-Klassenbibliothek (Übersicht)	655
	Stichwortverzeichnis	661



Vorwort

Die Programmierung von Windows-Anwendungen ist keine einfache Sache. Wer sich in das *Software Development Kit* einarbeiten muß, um seine Programme in C zu schreiben, kann davon ein Lied singen. Zwar ist C eine leistungsfähige und systemnahe Sprache, aber es gibt viele Fallstricke und versteckte Fehlermöglichkeiten. Hinzu kommt ein beträchtlicher Einarbeitungsaufwand. Obwohl es durchaus möglich ist, das erste Windows-Programm schon nach einem Tag (ab-)zuschreiben, dauert es bis zum echten Verständnis in der Regel noch eine ganze Weile.

Mit dem Entwicklungssystem Visual C++ hat Microsoft eine bessere Lösung geschaffen, die sich seit einigen Jahren in der Praxis bewährt hat und zum Standard geworden ist – genau wie Word oder Excel. Das Entwicklungssystem besteht aus Compiler, Klassenbibliothek für Windows (das sind die *Microsoft Foundation Classes*, kurz MFC genannt) und einer Anzahl von Tools. Wenngleich auch hier eine längere Einarbeitungsphase erforderlich ist, zeichnet sich Visual C++ durch eine Reihe von Vorteilen aus:

- ▼ Verwendbarkeit von C++ und damit Anwendung einer standardisierten, objektorientierten Sprache.
- ▼ Verfügbarkeit einer homogenen Klassenbibliothek, welche die wichtigsten Aspekte der Windows-Programmierung abdeckt und außerdem allgemein verwendbare Klassen zum Einsatz in beliebigen Programmen anbietet.
- ▼ Integration von Tools und Methoden, die das Entwickeln stabiler Windows-Anwendungen deutlich beschleunigen und häufige Fehlerquellen bereits im Vorfeld ausschalten.
- ▼ Ein Programmgerüst mit standardisierter Architektur verkürzt die Einarbeitungszeit in Projekte.

- ▼ Zu jeder Zeit ist es möglich, aus einem MFC-Programm heraus normale SDK-Funktionen aufzurufen.
- ▼ Programmierer, die bereits Windows-Anwendungen in C unter dem SDK entwickelt haben, können ihr Wissen weiterverwenden.
- ▼ Die Entwicklungsumgebung ist mittlerweile zu einem leistungsfähigen Paket geworden, das aus mehreren Sprachen besteht.

Bei der praktischen Arbeit mit Visual C++ wird deutlich, daß die Entwickler ihre Ziele im großen und ganzen erreicht haben.

Somit ist das Erlernen der MFC-Programmierung – auch im Hinblick auf zukünftige Entwicklungen im Betriebssystembereich für Windows (NT) – eine lohnende Investition. Betrachtet man zusätzlich die Vorzüge der objektorientierten Programmierung, so kann man aus heutiger Sicht davon ausgehen, daß sich die Foundation Classes in der Windows-Programmierung etabliert haben.

Neue Versionen von Visual C++ tragen auch immer der Betriebssystementwicklung Rechnung. Die jetzt vorliegende Version 6 ist um einige Möglichkeiten verbessert und erweitert worden und spiegelt auch Microsofts Strategie wider, alles für die Präsenz im Internet zu tun. Wir haben in dem vorliegenden Werk Wert gelegt auf eine gute Darstellung der Möglichkeiten von Visual C++ anhand von Beispielprogrammen. Damit soll ein noch besserer Lerneffekt erzielt werden. Das Layout des Buches trägt mit Symbolen, die auf wichtige Stellen hinweisen, dazu bei. Alle wichtigen Bestandteile der Programme werden Schritt für Schritt erläutert. Der komplette Quelltext befindet sich auf der CD. Der Leser ist am Ende des Buches nicht nur in der Lage, einfache MFC-Applikationen zu schreiben, sondern auch Daten zu speichern und zu lesen. Erst durch die Möglichkeit der Verarbeitung von Daten werden viele Programme sinnvoll.

Die Autoren wünschen daher jedem, der sich mit der Entwicklung von Windows-Anwendungen unter Visual C++ beschäftigt, viel Erfolg und hoffen, daß dieses Buch die dazu erforderlichen Informationen vermitteln kann.

*Frank Heimann, Nino Turianskyj
Hurlach/Leipzig, im Oktober 1998*



Grundlagen

TEIL I



Einleitung

1

Kapitelüberblick

1.1	Es geht los	20
1.1.1	Die Icons in diesem Buch	20
1.1.2	Was das Buch erklärt	20
1.1.3	Was das Buch nicht erklärt	22
1.2	Aufbau des Buches	23
1.3	Hard- und Software-Voraussetzungen	24
1.4	Installation von Visual C++	25
1.5	Grundlagen des Developer Studios	32
1.5.1	Integration in Windows	32
1.5.2	Philosophie	34
1.5.3	Aufbau des Developer Studios	35
1.6	Der Entwicklungszyklus	40
1.6.1	Projekte	40
1.6.2	Editieren von Quelltexten	50
1.6.3	Kompilieren und Linken	60
1.7	Dokumentation und Online-Hilfe	64
1.7.1	Gedruckte Handbücher	64
1.7.2	Online-Dokumentation	64
1.7.3	Weitere Informationsquellen	67
1.7.4	Referenzen auf elektronische Dokumente	67
1.7.5	Quelltexte zur MFC	67

1.1 Es geht los

1.1.1 Die Icons in diesem Buch

Um Ihnen die Orientierung in diesem Buch zu erleichtern, haben wir den Text in bestimmte Abschnitte mit speziellen Funktionen gegliedert und diese durch entsprechende Symbole oder Icons gekennzeichnet. Folgende Icons finden Verwendung:



Beispiele helfen Ihnen, sich schneller im Feld der Windows-Programmierung zu orientieren. Sie werden darum mit diesem Icon gekennzeichnet.



Bitte beachten Sie die **Hinweise**, die mit diesem Icon gekennzeichnet sind!



Achtung, durch dieses Icon wird eine **Warnung** angezeigt. Die hier beschriebenen Zusammenhänge führen leicht zu Fehlern und Problemen.



Durch **Übungen** und **Aufgaben** wiederholen und vertiefen Sie das Gelernte.



Wo es sinnvoll ist, finden Sie einen **Verweis**, damit Sie erfahren, wo Sie sich weiter über einen Sachverhalt informieren können.



Praxistips finden Sie in den Abschnitten, wo dieses Icon steht.

1.1.2 Was das Buch erklärt

Microsoft Visual C++ 6.0 besteht im wesentlichen aus folgenden Teilen:

- ▼ der integrierten Entwicklungsumgebung *Developer Studio*¹ mit Projektverwaltung, Editor, Compiler, Linker, Debugger und Browser
- ▼ dem integrierten Ressourcen-Editor
- ▼ der Klassenbibliothek *Microsoft Foundation Classes 6.0*
- ▼ dem integrierten Applikationsgenerator *AppWizard* (Anwendungs-Assistent)
- ▼ dem *ClassWizard* (Klassen-Assistent) zum Entwickeln von Klassen und Verbinden von Windows-Nachrichten mit Methoden

¹ Der Begriff *Developer Studio* wird nur noch selten benutzt. Es wurde jedoch auch kein neuer Name für die integrierte Entwicklungsumgebung geschaffen. Oft wird vom *Visual Studio* im Zusammenhang mit der Entwicklungsumgebung Gebrauch gemacht, wobei dies jedoch ein Produkt ist, das alle Visual-Programmiersprachen unter einer Oberfläche beinhaltet.

- ▼ der elektronischen Online-Dokumentation *MSDN Library Visual Studio 6.0*
- ▼ einer Reihe von zusätzlichen externen Dienstprogrammen.

Der Schwerpunkt dieses Buches liegt auf der Klassenbibliothek Microsoft Foundation Classes 6.0 (MFC). Das Verständnis der Funktionsweise der Foundation Classes ist der Schlüssel zum Erfolg bei der objektorientierten Entwicklung mit Visual C++. Die Foundation Classes ermöglichen es, all jene Windows-Programme zu schreiben, die auch in C möglich wären, und bieten darüber hinaus durch eine Reihe von High-Level-Klassen wahlweise die Möglichkeit, auf einem sehr viel höheren Abstraktionsniveau zu arbeiten.

Um praktisch mit Visual C++ arbeiten zu können, ist es allerdings zunächst erforderlich, in den ersten Kapiteln die Grundlagen der Entwicklungsumgebung und den Umgang mit dem Developer Studio so weit zu erklären, daß die Beispielprogramme dieses Buches editiert, übersetzt und getestet werden können. Die nachfolgenden Kapitel beschäftigen sich dann mit der Verwendung der Microsoft Foundation Classes »pur« und verzichten größtenteils auf die Erläuterung von Details der Entwicklungsumgebung und der Assistenten (Wizards). Diese Vorgehensweise hat den Vorteil, daß die (für die Programmierung) elementaren Dinge zunächst ballastfrei eingeführt werden können, ohne dabei jegliche Art von Programmen von vornherein in das dokumentorientierte Paradigma hineinzupressen, das zur Verwendung des Anwendungs-Assistenten und Klassen-Assistenten erforderlich ist.

Welche Arbeitserleichterung Anwendungs- und Klassen-Assistent bringen, ohne die *Dokument-Ansicht-Architektur* zu nutzen, kann ab Kapitel 3 nachvollzogen werden. In Version 6.0 wird dieses nun auch direkt vom Anwendungs-Assistenten unterstützt.

Ein größerer Teil des Buches in Kapitel 10 beschäftigt sich mit den Steuerungen (Controls), der eigentlichen Benutzerschnittstelle. Es werden alle vorhandenen Steuerungen nahezu vollständig erläutert. In Beispielprogrammen wird die Anwendung der Steuerungen demonstriert, die oft von Feinheiten abhängt.

Die bekannte dialogbasierende Anwendung HEXE (ein Taschenrechner) in Kapitel 12 und eine Anwendung mit den *Single Document Interface* (SDI) in Kapitel 13 fehlen ebensowenig wie der Zugriff auf externe Dateien.

Ab Kapitel 13.1 wird dann das komplette Paradigma der Foundation Classes und der Entwicklungsumgebung vorgestellt, so daß Sie sich einen Eindruck von dem Nutzen der zusätzlichen Tools und der darauf zugeschnittenen Klassen machen können. Hier wird vor allem deutlich werden, wie

das Konzept der Dokument-Ansicht-Architektur anzuwenden ist. Ein weiteres Beispiel mit der MFC zeigt, wie das *Multi-Document-Interface* (MDI) genutzt wird.

Des weiteren finden Sie ab Kapitel 15 die Grundlagen zur Nutzung der Datenbankchnittstelle in Visual C++.

1.1.3 Was das Buch nicht erklärt

Leider muß der in diesem Buch vermittelte Einblick in die Programmierung mit Visual C++ und den Microsoft Foundation Classes unvollständig bleiben. Hätte das Buch den Anspruch, allen Details einigermaßen gerecht zu werden, so müßte es ein Vielfaches seines heutigen Umfangs haben. Da es in diesem Buch jedoch um einen Einstieg und nicht um die Erstellung eines möglichst kompletten Nachschlagewerks geht, was ersteren ohnehin nur erschweren würde, wurde bewußt auf einiges verzichtet:

- ▼ Es werden nicht alle Hauptaspekte der MFC-Programmierung behandelt, d.h., bestimmte Klassen werden nur am Rande oder gar nicht erwähnt. Es wurde allerdings versucht, alle Themen aufzunehmen, die am Anfang wichtig sind; speziellere Themen können dann leicht mit Hilfe weiterführender Dokumentationen erarbeitet werden. In der deutsche Version leistet auch die Online-Hilfe teilweise gute Dienste. Leider ist die Dokumentation der MFC nur in englischer Sprache enthalten.
- ▼ Das Buch ist teilweise unvollständig bei der Darstellung von Details. Taucht eine bestimmte Methode in einem Beispielprogramm auf, so wird diese im allgemeinen zwar in ihrer aktuellen Ausprägung erklärt, die darüber hinausgehenden Aspekte werden aber meist verschwiegen. Hier hilft vor allem das Nachschlagen in der *Microsoft Foundation Class Reference* oder in einem der anderen Handbücher, die in elektronischer Form auf CD-ROM mitgeliefert werden.
- ▼ Nach einem Thema finden sich oft Referenzen auf einen Teil der Dokumentation, wo dann detailliertere Informationen geboten werden.
- ▼ Das Buch setzt gewisse Minimalkenntnisse der Windows-Programmierung voraus. Zwar muß man kein SDK¹-Experte sein, um es zu verstehen, ein Wissen um Grundzüge der Windows-Programmierung ist allerdings schon erforderlich. Zusätzlich werden fundierte C- und C++-Kenntnisse sowie allgemeine Kenntnisse in der Erstellung von Computerprogrammen erwartet. Es ist auch von Vorteil, Windows-Programme bereits in einer anderen Sprache (z.B. Visual Basic) geschrieben zu haben.

1 Software-Development-Kit – Programminterface zur Windows-Programmierung

- ▼ Es fehlen die Beschreibungen von Erweiterungen der Entwicklungsumgebung und der Klassenbibliothek wie ATL, OLE und DAO.
- ▼ Auf die Beschreibung der Internet-Server-Erweiterungen und von DLL-Bibliotheken wurde komplett verzichtet.

Daraus läßt sich die berechnete Vermutung ableiten, daß es ohne Grundkenntnisse, zusätzliche Dokumentation und den Willen zum Experimentieren nicht möglich ist, ein guter MFC-Programmierer zu werden – wenigstens ist es nicht sehr wahrscheinlich. Aber auch das Vorhandensein dieser Voraussetzungen ist noch keine Garantie für überwältigende Anfangserfolge oder gar für die Entstehung ausgereifter Windows-Programme bereits nach wenigen Tagen. Zusammenfassend läßt sich feststellen, daß man für das Erlernen der Windows-Programmierung mit den Microsoft Foundation Classes genügend Zeit einplanen sollte. Die Maßeinheit Tage ist dafür in der Regel nicht geeignet, auch wenn das oft suggeriert wird.

1.2 Aufbau des Buches

Das Buch besteht aus mehreren Teilen. Der erste Teil beschäftigt sich mit den Grundlagen der MFC-Programmierung und erklärt die Bedienung des Developer Studios. Dazu bedient er sich eines recht einfachen Beispielprogramms, das auf dem Bildschirm den Text »hello world« anzeigt.

Der zweite Teil erläutert die Programmierung einfacher Benutzerschnittstellen und behandelt Menüs, Beschleuniger sowie einfache Dialoge und zeigt, wie im Hauptfenster des Programms Text in verschiedenen Variationen ausgegeben werden kann. Zudem wird erläutert, wie man den Anwendungs- und Klassen-Assistenten anwenden kann, ohne gleich ein dokumentorientiertes Programm zu schreiben. Als Beispiel dient uns dabei ein Suchprogramm für Text in Dateien. Der Text wird auch dann gefunden, wenn die Dateien sehr groß sind und nicht in einen Editor geladen werden können.

Um mit dem Benutzer zu kommunizieren, sind Steuerelemente erforderlich, die dem Anwender die Eingaben erleichtern. Diese sind zum großen Teil in Windows standardisiert. Der dritte Teil erläutert alle Standard-Steuerelemente nahezu vollständig. Die Anwendung wird in Beispielprogrammen gezeigt. Des weiteren werden zusätzlich hilfreiche Klassen, zum Beispiel zur Verwendung von Feldern und Zeichenketten, erläutert.

Der vierte Teil beschäftigt sich eingehend mit Dialogboxen und Benutzerschnittstellenobjekten. Er erklärt im Detail das Erzeugen und Aufrufen von Dialogboxen und beschreibt ausführlich alle verfügbaren Dialogboxelemente. Doch auch das Anwendungsprogramm selbst wird dialogorientiert geschrieben. Als Beispielprogramm wird ein kleiner Taschenrechner

implementiert, der in verschiedenen Zahlensystemen rechnen kann und im wesentlichen aus einer Dialogbox besteht, die alle möglichen Arten von Steuerungen enthält.

Des weiteren behandelt dieser Teil die Dokument-Ansicht-Architektur der Foundation Classes mit Hilfe der beiden Tools Anwendungs-Assistent und Klassen-Assistent. Darüber hinaus erklärt er die Grundlagen der Grafikprogrammierung und veranschaulicht sie anhand einer einfachen Implementierung, nämlich der bekannten Simulation *Evolution* als eine SDI¹-Applikation.

Schließlich wird auf der Dokument-Ansicht-Architektur aufgebaut, und die Klassen *CFormView* und *CDocTemplate* werden behandelt. Es wird erläutert, wie man eine Online-Hilfe in eigene Programme einbaut. Außerdem wird der Zugriff auf Daten über Serilisation erläutert. Demonstriert wird dies an einem Beispielprogramm, das nach der Schablone noch erweitert werden kann.

Der fünfte Teil beschäftigt sich mit der ODBC-Datenbankschnittstelle der MFC. Alle relevanten Klassen werden erläutert. Das Beispielprogramm setzt auf dem bisher gelernten auf und demonstriert die Anwendung der Klassen an einem Ahnen-Programm, das es ermöglicht, einen Stammbaum aufzubauen und diesen in einer Datenbank zu speichern.

Jeder Teil enthält ein oder mehrere Kapitel. Innerhalb der Kapitel werden die relevanten thematischen Aspekte in aufeinander aufbauender Reihenfolge anhand der Beispielprogramme erklärt. Besonders wichtige Teile, wie z.B. Syntaxauszüge, werden durch eine Nichtproportionalschrift hervorgehoben. Alle Bestandteile der Programmiersprache sind kursiv dargestellt. Im Anhang sind der Windows-Zeichensatz, eine Klassenhierarchie sowie Gegenüberstellungen von Begriffen und Menüs der deutschen und englischen Version zu finden.

1.3 Hard- und Software-Voraussetzungen

Die Voraussetzungen zum Einsatz von Visual C++ 6.0 sind wie für fast alle Software-Programme mit der Entwicklung der Hardware gewachsen. Während bei der Version 1.5 ein 386er PC mit mindestens 4 MB RAM zur Minimalausstattung gehörte, ist es jetzt ein PC Pentium Prozessor mit einer Taktfrequenz von mindestens 133 MHz. Microsoft verlangt mindestens einen Pentium Prozessor mit 90 MHz. Aber selbst die Ausführungsgeschwindigkeit von Windows 95 läßt auf einem solchen Rechner zu wünschen übrig.

1 Single Document Interface

Als Betriebssysteme eignen sich nur die 32-Bit-Systeme Windows 95 und Windows NT ab Version 4.0 mit Service Pack 3. Da diese von sich aus bereits relativ hohe Anforderungen an die Hauptspeicherkapazität stellen, sind für den Einsatz von Visual C++ 24 MB (32 MB bei Windows NT) das Minimum. Empfohlen werden kann seitens der Autoren nur ein Speicheranbau ab 64 MB. Ein CD-ROM-Laufwerk sowie ausreichend Speicherplatz auf der Festplatte – ca. 305 MB belegt das Programm nach der *Standard Installation* – verstehen sich von selbst. Bitte denken Sie auch an den Speicherbedarf für den Internet Explorer (mindestens 43 MB) und für die Online-Hilfe MSDN (mindestens 57 MB). Für das CD-ROM-Laufwerk müssen unbedingt Protected-Mode-Treiber unter Windows 95 vorhanden sein, da sonst die Installation nicht möglich ist. Zudem ist zu beachten, daß nach der Installation noch mindestens 10 MB freier Festplattenplatz vorhanden sein muß, um überhaupt ein Projekt übersetzen zu können (auch die Auslagerungsdatei benötigt noch Platz!).

Des weiteren ist zur angenehmen Arbeit ein guter (großer) Monitor zu empfehlen, der mindestens mit einer Auflösung von 800x600 Punkten betrieben werden sollte. Dies ist erforderlich, um bei der Vielzahl der Elemente der Entwicklungsumgebung auch noch genügend Platz zum Editieren einer Datei zu haben.

In der heutigen Zeit sollte man als Entwickler von Programmen auch die Möglichkeit des Zugriffs auf das Internet mit einplanen. Oft werden dort neben Updates auch Beispielprogramme oder die kostenlose Lösung von Problemen (in Newsgroups) angeboten. So ist auch der Internet Explorer ab Version 4.01 Voraussetzung zur Arbeit mit Visual C++. Er wird benutzt, um die Online-Dokumentation zu lesen. Des weiteren stellt er einige neue Steuerelemente zur Verfügung.

Prinzipiell sollten die hier angegebenen Parameter als Mindestvoraussetzung verstanden werden. Auf einem 400 MHz Pentium II-Rechner mit 128 MB RAM und einer schnellen Festplatte macht die Arbeit mit Visual C++ natürlich mehr Spaß ...

1.4 Installation von Visual C++

Microsoft Visual C++ wird neben der 32-Bit-Version für Windows 95 und Windows NT 4.0 auch in einer Version für Alpha-Rechner geliefert. Applikationen lassen sich so für all diese Zielplattformen entwickeln. Programme für 16-Bit-Windows werden weiterhin mit Visual C++ 1.5 gefertigt. Die letzte verfügbare Version ist die 1.52c. Diese Version wird nicht weiterentwickelt oder mit neuen Eigenschaften versehen. Es finden lediglich noch Fehlerkorrekturen statt. Sie liegt nicht mehr dem Paket der Version 6 bei. Das vorliegende Buch bezieht sich ausschließlich auf die Version 6.0 für Windows 95 bzw. Windows NT 4.0.

Diese Version ist derzeit in drei verschiedenen Ausbaustufen erhältlich:

- ▼ *Einsteiger Edition*: Sie dient als kostengünstiger Einstieg in die C++-Programmierung und enthält nahezu alle Komponenten der Professional Edition. Die damit erstellten Programme dürfen vorläufig in der Version 6 auch weitergegeben werden.
- ▼ *Professional Edition*: Diese Version ist die gebräuchlichste. Sie enthält zusätzlich weitere MFC-Komponenten, Datenbankklassen, Klassen für Internet-Zugriff sowie die Möglichkeit der Erstellung von ActiveX-Controls. Zudem enthält sie einen verbesserten Compiler.
- ▼ *Enterprise Edition*: Sollen die mit Visual C++ erstellten Programme in Client/Server-Umgebungen genutzt werden, ist die Enterprise Edition erforderlich. Sie enthält zusätzlich zur Professional Edition eine begrenzte Version des SQL-Servers sowie weitere Programme zur Datenbankbearbeitung einschließlich des Debuggens von SQL-Prozeduren. Darüber hinaus enthält sie mit *Visual Source Safe* ein Programmpaket, das die Arbeit mehrerer Programmierer an einem Projekt komfortabler gestaltet. Sie ist damit auch die teuerste Variante.

Die Installation erfolgt komfortabel über ein Setup-Programm, welches noch durch die Autostart-Funktion in Windows 95 erleichtert wird. Die Autostart-Funktion ruft das Setup-Programm auf der CD auf, von dem aus die Installation von Visual C++ gesteuert wird.



Abbildung 1.1: Setup-Programm der Enterprise Edition von Visual C++

Die einzelnen Installationsschritte werden im folgenden erläutert. In diesem Buch werden alle Arbeiten unter Windows NT 4.0 durchgeführt. Die Funktionalität gilt aber auch für Windows 95.

1. Zum Starten der Installation wird die CD eingelegt. Durch die Auto-start-Funktion wird das Setup-Programm gestartet, dessen Eingangs-fenster in Abbildung 1.1 dargestellt ist. Hier kann man sich noch eine Info-Datei zur Installation und zur Edition anzeigen lassen.

Das Setup gestaltet sich so einfach wie die Installation von Windows 95 – über einen Installations-Assistenten. Er führt Schritt für Schritt durch die Installation und erlaubt es auch, zurückzugehen.

2. Nach einem einführenden Hinweis und der Bestätigung des Lizenz-Abkommens müssen die persönlichen Daten sowie der CD-Key, der sich auf der CD-Hülle befindet, eingegeben werden.
3. Danach wird die Installation des Internet Explorers der Version 4 überprüft (Abbildung 1.2). In den meisten Fällen wird dieser, auch wenn er schon installiert ist, noch aktualisiert. Ohne eine korrekte Installation des Explorers läßt sich Visual C++ nicht installieren. Nach der Installation des Internet Explorers wird der Rechner neu gestartet, und Sie setzen anschließend das Setup fort.

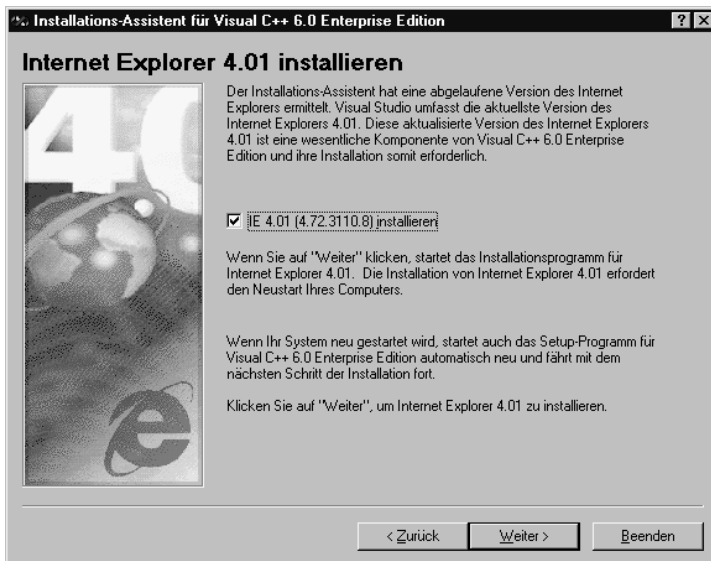


Abbildung 1.2: Die Installation des Internet Explorers ist Voraussetzung für Visual C++ 6.0

4. Hatten Sie zuvor das Visual Studio 97 mit einer Komponente wie Visual C++ installiert, können Sie dieses im nächsten Schritt entfernen (siehe Abbildung 1.3). Wenn Sie das nicht wünschen, können auch beide Produkte parallel installiert sein. Auch nach der Deinstallation wird der Rechner neu gestartet.

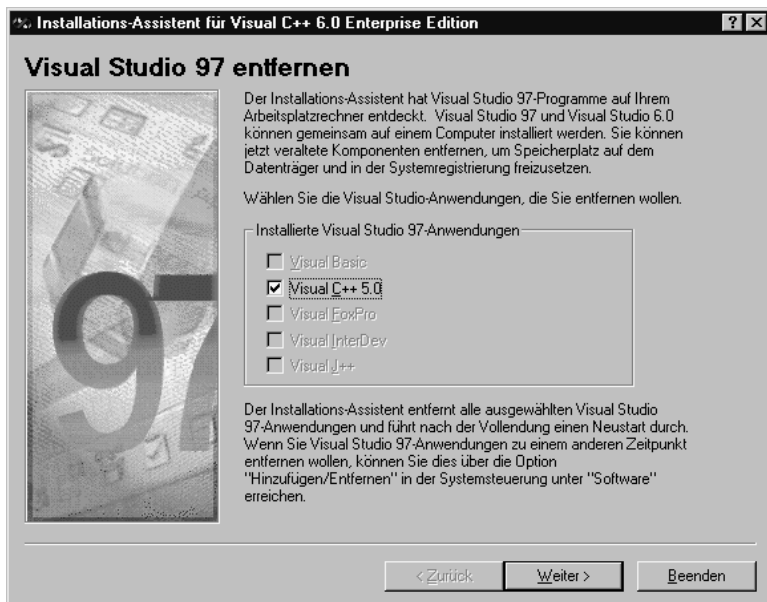


Abbildung 1.3: Entfernen einer alten Version

5. Der nächste Schritt dient zur Auswahl der Installationsvariante. Es können entweder nur die Server-Programme, oder auch die Programme auf den Clients gewählt werden. (siehe Abbildung 1.4).
6. Im nächsten Schritt wird das Basis-Verzeichnis der Installation für gemeinsam genutzte Dateien abgefragt. Dafür werden mindestens 50 MB benötigt.
7. Danach erfolgt die Frage nach der Installationsmethode von Visual C++ und des Verzeichnisses der Installation (siehe Abbildung 1.5). Zur Wahl stehen hier nur *Standard* und *Benutzerdefiniert*. Wenn Sie sich nicht genau mit den einzelnen Komponenten auskennen, können Sie beruhigt die Schaltfläche *STANDARD* betätigen. Jedoch verlangt schon die Standard-Installation mindestens 267 MB Platz auf der Festplatte.

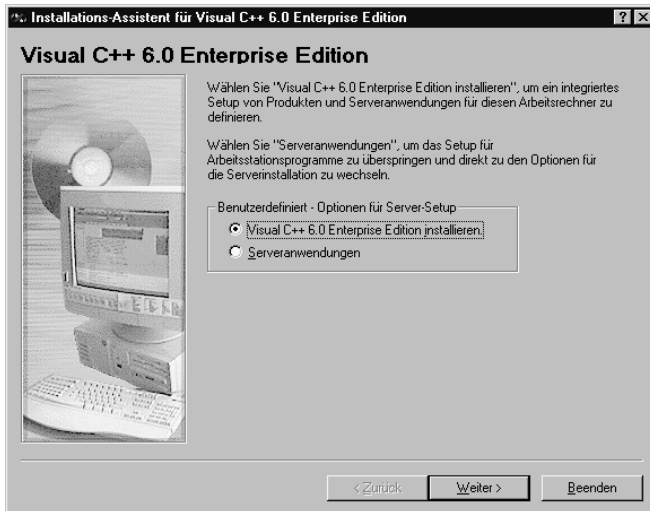


Abbildung 1.4: Varianten zur Installation

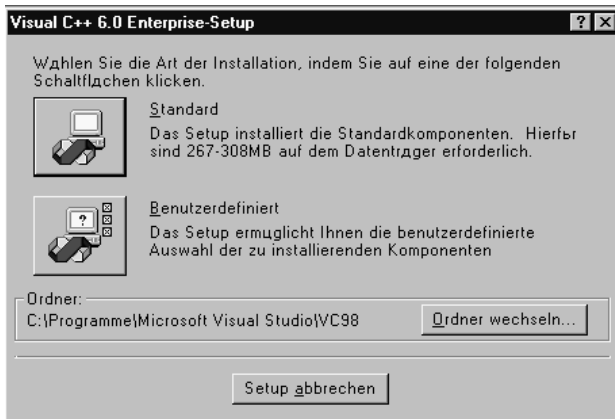


Abbildung 1.5: Festlegung der zu installierenden Komponenten

Wenn Sie genau wissen, was Sie installieren wollen, sollten Sie zur benutzerdefinierten Installation greifen. Wie in Abbildung 1.6 zu sehen ist, können Sie die zu installierenden Komponenten selbst bestimmen. Über den Button **OPTIONEN ÄNDERN** sind zu einem Punkt weitere Optionen auswählbar. Die einzelnen Stati der Checkboxes sollten zwar bekannt sein, hier aber trotzdem noch einmal eine kurze Erläuterung: Checkbox leer – keine Option gewählt; Checkbox grau unterlegt – einige Optionen gewählt; Checkbox mit Haken – alle Optionen wurden ausgewählt.

8. Neben der Auswahl der Methode kann auch noch das Verzeichnis bestimmt werden, in dem das Programmpaket installiert wird. Als Standard wird hier das Verzeichnis `\PROGRAMME\MICROSOFT VISUAL STUDIO\VC98` auf dem Installationslaufwerk des Betriebssystems vorgeschlagen.



Abbildung 1.6: Komponenten der Online-Dokumentation

Sie können bei dieser Methode auch Speicherplatz sparen, indem Sie nicht benötigte Komponenten deaktivieren.

9. In Abbildung 1.7 sind die Optionen einer Komponente dargestellt. Bei der MFC ist es günstig für die spätere Arbeit, die Browser-Datenbank zu installieren. Für alle, die sich mit UML auskennen, ist der Visual Modeler sehr interessant. Mit ihm lassen sich Klassen modellieren und danach in Quellcode verwandeln.
10. Der letzte Schritt bei der Installation ist die Möglichkeit des Registrierens der Umgebungsvariablen. Dies ist nur erforderlich, wenn Sie Compiler oder Linker von der Eingabeaufforderung aus aufrufen möchten.
11. Nach einem Neustart des Rechners geht die Installation über zur Online-Dokumentation (siehe Abbildung 1.8). Diese befindet sich für alle Visual Studio-Programme auf den Microsoft Developer Network CDs (MSDN). Die Online-Dokumentation ist damit nicht mehr in das Developer Studio integriert.

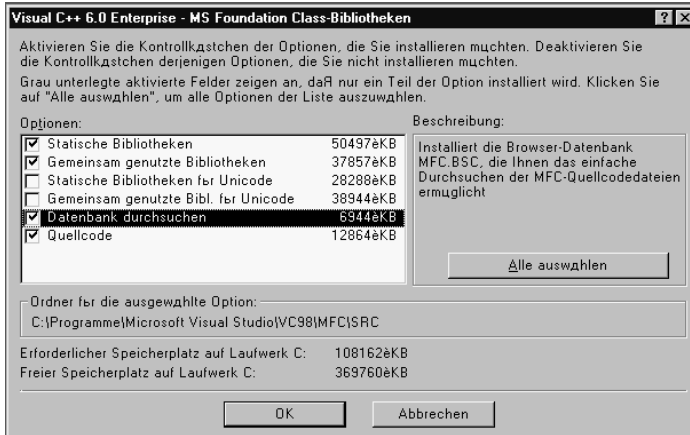


Abbildung 1.7: Optionen der MFC-Bibliothek



Abbildung 1.8: Installation der MSDN-Library

12. Im nächsten Schritt muß auch bei der MSDN die Installationsmethode gewählt werden. Hier kann zwischen *Standard*, *Benutzerdefiniert* und *Vollständig* gewählt werden. Standard entspricht zugleich dem Minimum an benötigtem Speicherplatz – immerhin 60 MB. Sie sollten hier deshalb in jedem Fall *Standard* oder *Benutzerdefiniert* (Abbildung 1.9) wählen.

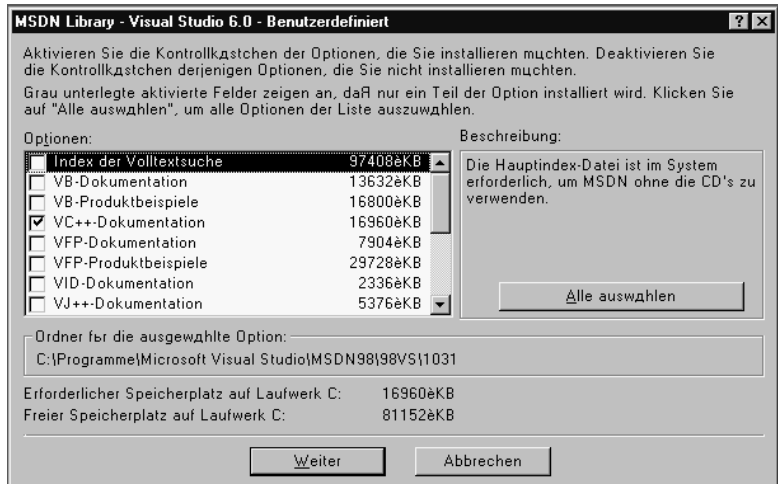


Abbildung 1.9: Optionen der benutzerdefinierten MSDN-Installation

13. Nach der MSDN-Installation besteht die Möglichkeit, weitere Programme von der CD zu installieren. Hier wird nur eine Version von *InstallShield* angeboten.
14. Beim letzten Schritt der Enterprise-Installation können noch die Server-Komponenten installiert werden.

Bei der Installation werden keine Beispielpprogramme kopiert, da diese erfahrungsgemäß sehr viel Platz in Anspruch nehmen. Sie können bei Bedarf über die Online-Dokumentation von der CD kopiert werden.

Falls später noch weitere Komponenten installiert werden sollen, müssen Sie nur noch einmal das Installationsprogramm starten und die benutzerdefinierte Installation wählen. Das Setup-Programm erkennt automatisch alle schon installierten Teile und zeigt sie auch an.



Weitere Informationen zur Installation von Visual C++ finden Sie auf der mitgelieferten Seite »Erste Schritte« sowie in der Info-Datei.

1.5 Grundlagen des Developer Studios

1.5.1 Integration in Windows

Zum Abschluß der Installation wurde eine neue Programmgruppe *Microsoft Visual C++ 6.0* mit Untermenü angelegt (Abbildungen 1.10 und 1.11). Das Programm *Visual Sourcesafe 6.0* ist Bestandteil der Enterprise Edition und unterstützt das Arbeiten mehrerer Personen an einem Projekt. Es ist in einer eigenen Programmgruppe zu finden (Abbildung 1.12).



Abbildung 1.10: Visual C++ Programmgruppe

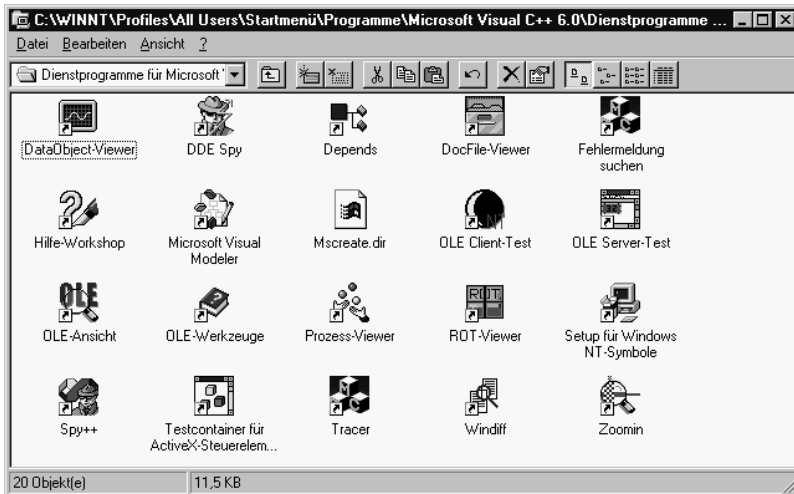


Abbildung 1.11: Die Dienstprogramme zu Visual C++ 6.0

Je nach eingestellten Optionen bei der Installation sind außer dem *Microsoft Developer Studio* weitere Verknüpfungen zu diversen Tools vorhanden. Dazu zählen unter anderem eine *Prozeßanzeige*, der *Testcontainer für ActiveX-Steuerelemente* oder der *Hilfe-Workshop*, der es erlaubt, eigene Hilfe-Dateien inklusive der dazugehörenden Grafiken zu erstellen und automatisch zu testen. Ein sehr interessantes Programm ist *WinDiff*, mit dem man Textdateien auf komfortable Weise vergleichen kann. Der Zugriff auf die Online-Hilfe der MFC sowie C/C++ erfolgt innerhalb der *MSDN Library*. Diese kann separat oder durch Drücken von **[F1]** innerhalb der Entwicklungsumgebung aufgerufen werden.

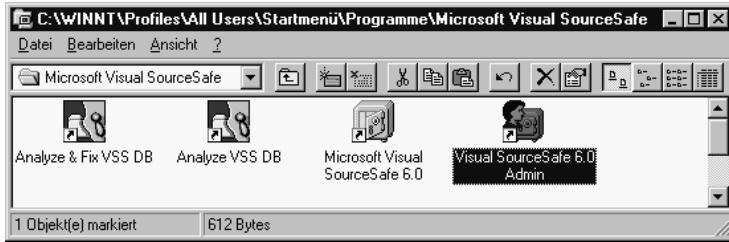


Abbildung 1.12: Die Programmgruppe Visual SourceSafe

1.5.2 Philosophie

Das *Visual Studio* ist die integrierte Entwicklungsumgebung (siehe Abbildung 1.13). Sie ist schon seit der Version 4.x unter dem Namen Developer Studio bekannt und wurde noch erweitert und verbessert. Das Visual Studio ist nur ein anderer Name für das Developer Studio. Die Grundidee liegt in einer Workbench als Entwicklungswerkzeug mit allen benötigten Komponenten. Darüber hinaus ist das Visual Studio nicht nur die Entwicklungsumgebung für Visual C++, sondern auch für alle anderen Programme von Microsoft wie Visual Basic, Visual FoxPro, Visual J++ oder Visual InterDev zur Entwicklung von Internet-Server-Applikationen. Es können auch aus verschiedenen Programmiersprachen zusammengesetzte Projekte erstellt werden.

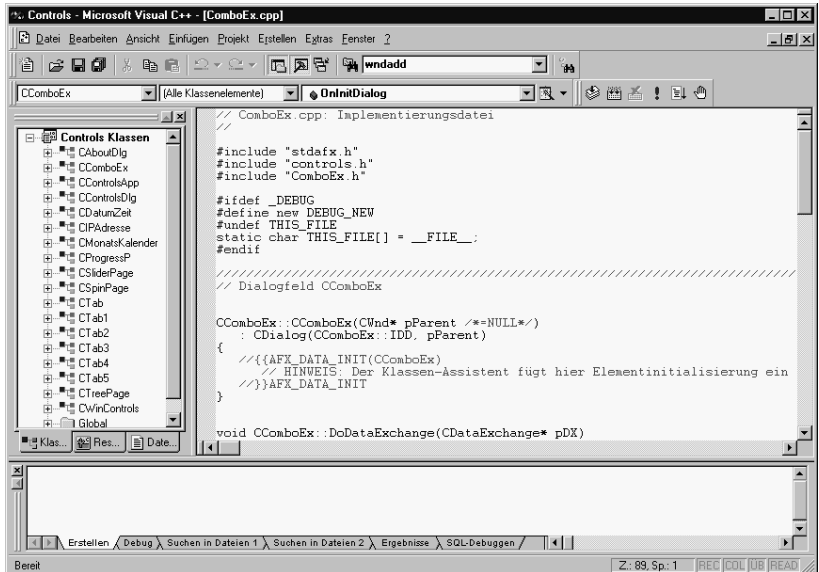


Abbildung 1.13: Ansicht des Visual Studios während der Entwicklung

Die vielfältigen Möglichkeiten des Developer Studios wirken auf den ersten Blick erdrückend. Enthalten sind unter anderem ein komfortabler Editor mit Syntax-Coloring, Browser für Quelltexte, Klassen und Ressourcen oder der Debugger. Vom Konzept der integrierten Online-Dokumentation ist man wieder weggegangen und hat die gesamte Dokumentation in die MSDN-Library gepackt (Abbildung 1.14). Die Möglichkeit des Aufrufs von Compiler und Linker mit Einstellung der Optionen versteht sich von selbst. Bei Compiler-Fehlern kann sofort an die entsprechende Stelle gesprungen werden, bei Laufzeitfehlern hilft der integrierte Debugger weiter. Die Erstellung und Änderung von Ressource-Dateien erfolgt innerhalb der Entwicklungsumgebung ohne Aufruf eines externen Programms. Das Developer Studio stellt somit alle wünschenswerten Komponenten zur Entwicklung von Applikationen bereit und muß nicht für Arbeiten wie etwa die Ressource-Erstellung verlassen werden.

Nach dem Einlegen der *MSDN Library*-CD kann diese durchsucht werden.

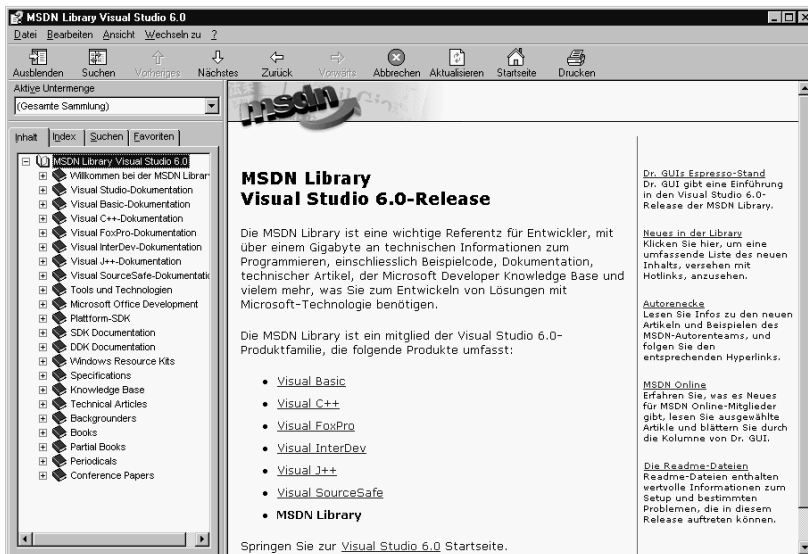


Abbildung 1.14: Die Online-Dokumentation in der MSDN Library

1.5.3 Aufbau des Developer Studios

Die Vielzahl der Komponenten des Developer Studios lassen schnell erkennen, warum ein großer Monitor mit einer hochauflösenden Grafikkarte benutzt werden sollte. Ansonsten müßte zugunsten der Übersichtlichkeit auf einige Fenster oder Symbolleisten verzichtet bzw. zwischen ihnen hin und her geschaltet werden, was den Komfort der Entwicklungs-

umgebung mindern würde. Die Funktion der einzelnen Elemente soll im folgenden erläutert werden. Es handelt sich dabei um eine MDI-Applikation (Multiple Document Interface), die es ermöglicht, mehrere Fenster gleichzeitig zu öffnen, und die dadurch die Möglichkeiten der Oberfläche von Windows 95 und Windows NT ausnutzt.

Bezeichnung der Elemente

Im oberen Teil des Developer Studios befinden sich wie bei jeder Windows-Anwendung das Menü sowie darunter optional eine oder mehrere Symbolleisten (Toolbars). Am unteren Rand liegt die Statuszeile, die ihr Aussehen immer der gerade benutzten Funktion entsprechend anpaßt. Während bei der Arbeit mit dem Quelltext-Editor die Zeile und Spalte der Cursorposition angezeigt wird, ist es beim Ressource-Editor die Position und Größe des markierten Elements. Menü und Symbolleisten entsprechen dem Layout, das mit dem Internet Explorer 3.x eingeführt wurde. Das Menü kann wie die Symbolleisten ebenfalls an beliebige Stellen verschoben bzw. angedockt werden. Die Symbole und Menütexte werden dreidimensional dargestellt, sobald die Maus darüber hinwegbewegt wird. Symbolleisten und Menüs können völlig frei definiert und zusammengestellt werden, so daß einzelne Menüs auch in Symbolleisten enthalten sein können.

Über der Statuszeile liegt das *Ausgabe*-Fenster (Output). Es nimmt die Ausschriften diverser Programme wie Compiler und Linker auf. Über die Reiter am unteren Rand kann zwischen verschiedenen Programmen umgeschaltet werden. Tabelle 1.1 faßt die Programme zusammen.

Option	Bedeutung
Erstellen (Build)	Ausgaben von Compiler und Linker, über Doppelklick kann zur fehlerhaften Zeile gesprungen werden.
Debug	Ausschriften während des Debuggens eines Programms, u.a. Meldungen des TRACE-Makros.
Suchen in Dateien (Find in Files)	Gefundene Textstellen in den über BEARBEITEN/SUCHEN IN DATEIEN durchsuchten Dateien werden angezeigt. Über Doppelklick kann die selektierte Datei geöffnet und die Fundstelle angezeigt werden.
Profil (Profile)	Ausschriften des Profilers, der über ERSTELLEN/PROFIL gestartet wird, dient zur Analyse des Laufzeitverhaltens eines Programms.

Tabelle 1.1: Anzeigoptionen des Ausgabe-Fensters

Den Hauptteil des Bildschirmbereiches nehmen das *Editor*-Fenster sowie das *Arbeitsbereich*-Fenster (Project Workspace) ein. Im Sinne der MDI-Spezifikation enthält das Editor-Fenster immer ein Dokument. Zwischen mehreren geöffneten Dokumenten kann im Menü FENSTER umgeschaltet werden. Das *Editor*-Fenster wechselt den Inhalt nach Bedarf. Es kann die

Quelltexte, Ressourcen oder auch die Hilfetexte aufnehmen. Die Auswahl des Inhalts erfolgt entweder über das Menü **FENSTER** oder das Fenster *Arbeitsbereich*. Ein Doppelklick auf ein Element im Arbeitsbereich bringt es zur Anzeige. Dateien können aber auch einfach durch Auswahl des Menüpunkts **ÖFFNEN** im Menü **DATEI** dargestellt werden.

Das *Arbeitsbereich*-Fenster ist eine interessante Methode, das Arbeiten erheblich zu vereinfachen. Am unteren Ende befinden sich Reiter, die zwischen folgenden Ansichten umschalten:

- ▼ Klassen
- ▼ Ressourcen
- ▼ Dateien
- ▼ Daten (nur Enterprise Edition)

Innerhalb einer Kategorie werden die einzelnen Komponenten in einer hierarchischen Struktur baumartig dargestellt, was zum Beispiel das Navigieren zu einer Dialog-Ressource sehr einfach gestaltet. Der Doppelklick öffnet dann das gewählte Element, und das Editieren kann beginnen. Auch in langen Quelltexten kann so ohne aufwendiges Blättern gezielt zu einer bestimmten Position oder Methode gesprungen werden. Der Arbeitsbereich stellt damit einen Browser für alle Elemente des Projekts zur Verfügung.

Symbolleisten

Die Symbolleisten oder Toolbars haben an Umfang und somit an Möglichkeiten zugelegt. Neue Symbolleisten können individuell zusammengestellt werden. Beliebig viele dieser Toolbars können angezeigt und auf der Oberfläche verteilt werden. Die wichtigsten Symbolleisten mit ihren Buttons werden in den Tabellen 1.2 und 1.3 zusammengefasst. Die Nummer des Elements entspricht seiner Position in der Symbolleiste (siehe Abbildungen 1.15 und 1.16).



Abbildung 1.15: Standard-Symbolleiste



Abbildung 1.16: Minileiste erstellen (Build Minibar)

Symbolleiste	Element	Bedeutung
Standard	1	Neue Quelltext-Datei anlegen
	2	Öffnen einer Datei (nicht auf Quelltexte beschränkt)
	3	Speichern der Datei im <i>Editor</i> -Fenster
	4	Speichern aller geöffneten Dateien
	5	Ausschneiden
	6	Kopieren
	7	Einfügen
	8/9	Rückgängig machen / Wiederherstellen
	10	Ein-/Ausblenden <i>Arbeitsbereich</i> -Fenster
	11	Ein-/Ausblenden <i>Ausgabe</i> -Fenster
	12	Anzeigen der Fensterliste zum Wechseln von Fenstern
	13	Suchen von Text in mehreren Dateien
	14	Eingabe/Auswahl des zu suchenden Textes
	15	Suchen in der Online-Dokumentation

Tabelle 1.2: Standard-Symbolleiste und deren Elemente

Symbolleiste	Element	Bedeutung
Minileiste erstellen	1	Kompilieren der aktuellen Datei
	2	Kompilieren und Linken (Erstellen/Build) der geänderten Dateien
	3	Stoppen des Erstellungs-Vorgangs
	4	Ausführen der erstellten EXE-Datei
	5	Ausführen der erstellten EXE-Datei im Debug-Modus
	6	Setzen/Löschen eines Breakpoints

Tabelle 1.3: »Minileiste erstellen« und deren Elemente

Vordefiniert sind des weiteren Symbolleisten, die bei Bedarf aktiviert werden können (z.B. über das Kontextmenü mit der rechten Maustaste). In der folgenden Aufzählung werden diese Symbolleisten aufgeführt:

- ▼ Ressource (Resource)
- ▼ Bearbeiten (Edit)
- ▼ Debug
- ▼ Erstellen (Build)
- ▼ InfoViewer
- ▼ Durchsuchen (Browse)

- ▼ ATL
- ▼ Datenbank (Database)
- ▼ Assistentenleiste (WizardBar)
- ▼ Quellcodeverwaltung (in der Enterprise-Edition)

Das Anpassen und Erzeugen neuer Symbolleisten erfolgt über das Menü EXTRAS und den Punkt ANPASSEN oder über das Kontextmenü (rechte Maustaste).

Funktionstasten

Da viele Anwender schneller mit der Tastatur als mit der Maus arbeiten, können viele Funktionen nicht nur über Menü oder Button, sondern auch über die Tastatur ausgelöst werden. Theoretisch ist es möglich, allen Funktionen eigene Tastenkombinationen zuzuweisen, jedoch ist dies sicher nicht sinnvoll. Diese Einstellung kann ebenfalls im Menü EXTRAS|ANPASSEN|TASTATUR vorgenommen werden. Einige wichtige Tastenkürzel sind in der Tabelle 1.4 zusammengestellt.

Tastenkombination	Bedeutung
F1	Aufruf der (kontextsensitiven) Hilfe
F3/Shift-F3	Weitersuchen vorwärts/rückwärts
F4/Shift-F4	Sprung zum nächsten/vorhergehenden Fehler
F5/Strg-F5	Programm über Debugger starten/ohne Debugger starten
F6/Shift-F6	Nächstes/vorhergehendes Fenster aktivieren
F7	Erstellen (Build)
F9	Haltepunkt (Breakpoint) setzen/löschen
F10/F11	Einzelschrittmodus beim Debuggen mit Sprung über/in die nächste Funktion
Strg-F10	Programmausführung bis Cursorposition
Strg-X/Strg-C/Strg-V	Cut/Copy/Paste bzw. Ausschneiden/Kopieren/Einfügen
Strg-W	Aufruf des MFC-Klassen-Assistenten (ClassWizard)
Alt-0	Wechseln zum <i>Arbeitsbereich</i> -Fenster
Alt-2	Wechseln zum <i>Ausgabe</i> -Fenster

Tabelle 1.4: Hotkeys im Developer Studio

Eine Liste aller derzeitigen wirksamen Tastenkürzel, auch der Standard-Tastenkombinationen wie Cut/Copy/Paste, sind in der Hilfe aufgeführt, erreichbar im Menü ? unter dem Punkt TASTATURBELEGUNG.

Veränderung der Arbeitsoberfläche

Die Oberfläche des Developer Studios kann den eigenen Wünschen und Vorstellungen gemäß angepaßt werden. Es können verändert werden:

- ▼ Anzahl und Anordnung der Fenster (Ein- und Ausschalten von Fenstern, z.B. Arbeitsbereich, Ausgabe etc.)
- ▼ Art der Behandlung von Fenstern als normales Fenster oder andockbar (EXTRAS|OPTIONEN|ARBEITSBEREICH)
- ▼ Anzahl und Anordnung der Toolbars (EXTRAS|ANPASSEN|SYMBOLLEISTEN)
- ▼ Funktionstastenbelegung und Hotkeys (EXTRAS|ANPASSEN|TASTATUR)
- ▼ Aufrufe für externe Programme im TOOLS (EXTRAS|ANPASSEN|TOOLS)
- ▼ Allgemeine Einstellungen der Entwicklungsumgebung (EXTRAS|OPTIONEN)



Weitere Hinweise zum Thema Verändern der Oberfläche finden Sie in der Online-Hilfe unter dem Stichwort »Anpassen von Visual C++«.

1.6 Der Entwicklungszyklus

1.6.1 Projekte

Grundlagen

Die Arbeit in der Entwicklungsumgebung von Visual C++ wird in *Projekten* organisiert. Der *Arbeitsbereich* ist physisch gesehen ein Verzeichnis auf dem Datenträger, der den Namen des Projekts trägt. In diesem Arbeitsbereich werden alle nötigen Daten des Projekts in diversen Dateien und Unterverzeichnissen gespeichert. Das Projektverzeichnis ist das Wurzelverzeichnis aller Dateien dieses Projekts. Es kann weitere Unterprojekte enthalten, von denen das Hauptprojekt abhängig ist (z.B. DLLs). Durch Aufstellung von Regeln über Abhängigkeiten werden beim Generieren der Programme immer alle nötigen Dateien mitübersetzt, auch die der Unterprojekte. Ein Projekt kann auch aus Programmteilen, die in verschiedenen Programmiersprachen geschrieben wurden, zusammengesetzt werden, z.B. Visual C++ und Visual InterDev. Folgende Daten werden im Projektverzeichnis gespeichert:

- ▼ Einstellungen der Entwicklungsumgebung (u.a. Größe und Position der Fenster)
- ▼ Quelldateien, ggf. Bibliotheken und DLLs
- ▼ Einstellungen der Optionen zum Kompilieren und Linken
- ▼ Datenbanken des Source-Browsers (*.SBR)

- ▼ Zustand der Entwicklungsumgebung (u. a. offene Dateien, Position des InfoViewers)
- ▼ Unterprojekte mit ihren Dateien
- ▼ Make-File (.MAK) zum Generieren des Projekts mit NMAKE.

Das Projekt wird im Developer Studio als Arbeitsbereich bezeichnet. Alle Informationen zum Arbeitsbereich werden in einer Datei gespeichert, die den Namen des Projekts sowie die Erweiterung .DSW trägt. Des weiteren werden eine Projektdatei (.DSP) – eine Make-Datei für die Entwicklungsumgebung – und eine Datei mit den eingestellten Optionen des Arbeitsbereichs (.OPT) erstellt.

Wenn Sie einen Arbeitsbereich über das Menü DATEI|ARBEITSBEREICH ÖFFNEN aufrufen wollen, werden Ihnen alle Dateien mit den Erweiterungen .DSW und .MDP angezeigt. Die Erweiterung .MDP wurde in der Version 4.x für Arbeitsbereiche verwendet. Wenn Sie ein solches Projekt öffnen, werden Sie gefragt, ob die Steuerdateien in die neue Version konvertiert werden sollen.



Grundsätzlich ist es auch möglich, ohne die Entwicklungsumgebung zu arbeiten und auf Projekte zu verzichten, aber gerade die Integration aller Entwicklungswerkzeuge in einer Oberfläche macht den Komfort beim Programmieren aus. Sollte es doch einmal nötig sein, ohne Developer Studio zu arbeiten, findet man Compiler, Linker etc. im Verzeichnis PROGRAMME\MICROSOFT VISUAL STUDIO\VC98\BIN. Zudem befindet sich in diesem Verzeichnis die Datei VCVARS32.BAT, die zum Setzen der Environment-Variablen dient. Es ist dabei auf einen ausreichend dimensionierten Bereich für die Umgebungsvariablen zu achten.

Verzeichnisstruktur eines Projekts

Jedem Projekt ist ein Projektverzeichnis zugeordnet. Alle darin enthaltenen Dateien und Verzeichnisse sind wiederum nach einer festgelegten Struktur definiert. Die Struktur finden Sie in Abbildung 1.17, wobei das Vorhandensein der Unterverzeichnisse natürlich vom Projekttyp und den ihm zugeordneten Optionen abhängt:

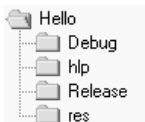


Abbildung 1.17: Verzeichnisstruktur eines Projekts

Verzeichnis	Inhalt
Projektverzeichnis	alle zum Projekt gehörenden Quelldateien.
RES	Ressourcen wie Icons und Bitmaps, nicht das RC-File des Projekts.
DEBUG	Objekt-, Browser- und andere Hilfsdateien sowie EXE-Datei der Debug-Version.
RELEASE	Objekt-, Browser- und andere Hilfsdateien sowie EXE-Datei der Release-Version.
HLP	Hilfe-Texte sowie zugehörige Grafikdateien.
SUB...	Verzeichnisstruktur eines Unterprojekts, Name des Verzeichnisses ist der Projektname (Unterprojekte können auch in anderen Verzeichnissen abgelegt werden. Ein Verweis befindet sich in der DSW-Datei.).

Tabelle 1.5: Inhalt der Unterverzeichnisse eines Projekts

In Tabelle 1.5 sind die Namen der Unterverzeichnisse sowie deren Inhalt aufgeführt. Von Vorteil ist es, daß alle vom Compiler und Linker während des Erstellungs-Vorgangs erzeugten Dateien in einem Unterverzeichnis abgelegt werden. Bei Platzmangel auf der Festplatte können diese komplett gelöscht werden. Auch das Anlegen einer Sicherheitskopie wird vereinfacht, da alle Quelldateien im Projektverzeichnis liegen, und nur dieses zusammen mit den Verzeichnissen RES und HLP gesichert werden muß.

Anlegen eines neuen Projekts

Das Anlegen eines Projektes ist vergleichbar mit der Erstellung des Make-Files. Die Entwicklungsumgebung nimmt einem die Arbeit des Anlegens und Änderns des Make-Files (*Projekt.DSP*) ab und trägt alle erforderlichen Dateien, Abhängigkeiten und Schalter zum Erzeugen des gewählten Programmtyps ein. Das Make-File darf nicht von Hand geändert werden, da das zu Inkonsistenzen führen kann. Zu Beginn des Make-Files steht daher auch gleich die Aufforderung »NICHT BEARBEITEN«. Ein Blick in ein vorhandenes Make-File macht verständlich, warum darin nicht editiert werden sollte.



Soll ein Projekt von der Kommandozeile aus mit NMAKE übersetzt werden, muß zuvor ein Make-File (.MAK) über den Menüpunkt PROJEKT|MAKEFILE EXPORTIEREN erstellt werden.

Zum Programmieren ist es zwingend erforderlich, mit Projekten zu arbeiten, da nur diese sich übersetzen lassen. Projekte werden in verschiedene Typen unterteilt, wobei jedes Projekt immer nur von einem Typ sein kann. Folgende Projekttypen sind vordefiniert:

- ▼ *MFC Anwendungs-Assistent (exe)*: Applikation unter Nutzung der MFC, wobei eine »leere« Anwendung bereits beim Anlegen des Projekts erzeugt wird
- ▼ *MFC Anwendungs-Assistent (dll)*: DLL unter Nutzung der MFC mit Anlegen eines Programmgerüsts

- ▼ *Win32 Anwendung*: Anwendung in C oder C++ mit Win32 API-Funktionensaufrufen
- ▼ *Win32 Dynamic-Link Library*: DLL unter Anwendung von Win32 API-Funktionen
- ▼ *Win32 Konsolenanwendung*: Windows-Konsolenanwendung unter Nutzung von Console API-Funktionen, die die zeichenorientierte Ein- und Ausgabe über *printf()* oder *scanf()* ermöglichen
- ▼ *Win32 Bibliothek (statische)*: Erzeugen einer statischen Bibliothek, die beim Linken in andere Projekte eingebunden werden kann
- ▼ *Makefile*: Einbinden von Make-Files in die Entwicklungsumgebung, die nicht mit dem Developer Studio erstellt wurden
- ▼ *MFC ActiveX Steuerelement-Assistent*: Erzeugen eines Programmgerüsts für ActiveX Controls, die Nachfolger der weit verbreiteten VBX- und OCX-Controls
- ▼ *ATL COM Anwendungs-Assistent*: Erstellen einer Active-Template-Library-Anwendung, die meist sehr klein und damit zur Ausführung über das Internet geeignet ist
- ▼ *Assistent für ISAPI-Erweiterungen*: Mit Hilfe dieses Assistenten werden Dateien angelegt, die zur Erweiterung von Internet-Servern über das entsprechende API dienen
- ▼ *Add-In-Assistent für DevStudio*: Erstellen von Programmen und Bibliotheken zur Automatisierung von Aufgaben in der Entwicklungsumgebung
- ▼ *Benutzerdefinierter Anwendungs-Assistent*: Modifizieren des Standard-AppWizards für das Erzeugen von Applikationen, die auf eigenen Templates (Schablonen) basieren
- ▼ *Dienstprogramm*: Ein Dienstprogramm-Projekt enthält keine Dateien. Es erstellt keine zuvor festgelegten Ausgabedateien. Es kann als Container für Dateien oder Projekte verwendet werden, die Sie erstellen wollen, ohne zu binden.
- ▼ *Assistent für erweiterte gespeicherte Prozeduren*: Dieses Projekt dient zum Erstellen von erweiterten gespeicherten Prozeduren für den Microsoft SQL-Server. Es ist nur in der Enterprise Edition verfügbar.
- ▼ *Cluster-Ressourcentyp-Assistent*: Wenn Sie Anwendungen für einen Microsoft Cluster Server schreiben wollen, ist dieser Projekttyp notwendig. Er erzeugt zwei DLL-Projekte dafür.
- ▼ *Datenbank-Assistent*: Mit diesem Assistenten können Sie eine neue Datenbank auf einem Microsoft SQL-Server erstellen.

- ▼ **Datenbankprojekt:** Ein Datenbankprojekt dient zur Bearbeitung und zum Test von gespeicherten Prozeduren in ODBC-Datenbanken (siehe Abbildung 1.18). Man umgeht damit das Erstellen einer Testanwendung für die Prozeduren. Diese Projekte sind nur in der Enterprise Edition verfügbar.

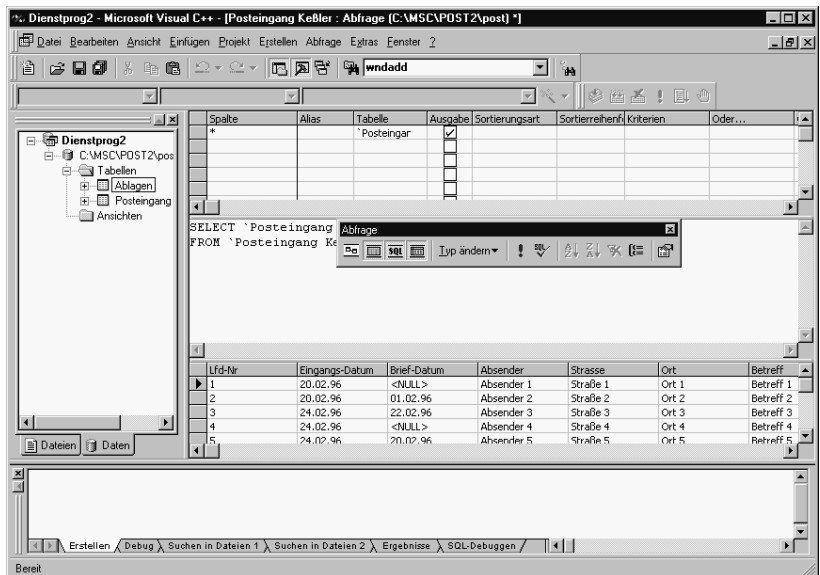


Abbildung 1.18: Ein Datenbankprojekt im Visual Studio

In diesem Buch beschäftigen wir uns mit dem Erzeugen von ausführbaren Programmen (EXE) über den Anwendungs-Assistenten bzw. zu Beginn mit einer einfacheren Form, einem Projekt vom Typ Win32 Application.

Das Anlegen eines neuen Projektes geht in folgenden Schritten vor sich:

1. Aufruf des Menüpunktes DATEI|NEU (**Strg** **N**) zum Anlegen einer neuen Datei.
2. Den Reiter PROJEKTE anwählen.
3. Im folgenden Dialog werden die Einstellungen für das Projekt vorgenommen (siehe Abbildung 1.19). Der Typ der Anwendung wird im Listenelement bestimmt. Im Eingabefeld für *Projektname* wird der Name des Projekts festgelegt. Dieser entspricht den Dateinamenskonventionen mit langen Dateinamen. Da jedoch Namen von Funktionen und Klassen zum Teil aus dem Projektnamen abgeleitet werden, sollte dieser nicht zu lang werden. Der Button mit den drei Punkten dient zur Auswahl des Startverzeichnisses für Projekte. Über einen Aus-

wahldialog wird dieses per Mausklick selektiert. In diesem Verzeichnis wird dann ein Unterverzeichnis mit dem Namen des Projekts angelegt, das als Projektverzeichnis dient. In der Liste *Plattformen* können die Zielplattformen des Projekts ausgewählt werden. Im allgemeinen wird das nur die Win32-Zielplattform sein.

4. Das Drücken des Buttons OK führt nach einem Hinweis, was alles erzeugt wird, zum Anlegen des entsprechenden Projekts. Das neu angelegte Projekt wird danach sofort geöffnet und kann bearbeitet werden.

Beim Anlegen eines neuen Projekts vom Typ MFC Anwendungs-Assistent (EXE) werden noch eine Reihe weiterer Informationen abgefragt, die die Eigenschaften der späteren Anwendung bestimmen. Gemeinsam ist beiden Typen das Anlegen der *Arbeitsbereich*-Dateien. Der Unterschied zu herkömmlichen Entwicklungsumgebungen besteht darin, daß das Makefile nicht mehr von Hand editiert wird, und daß Abhängigkeiten automatisch erkannt werden.

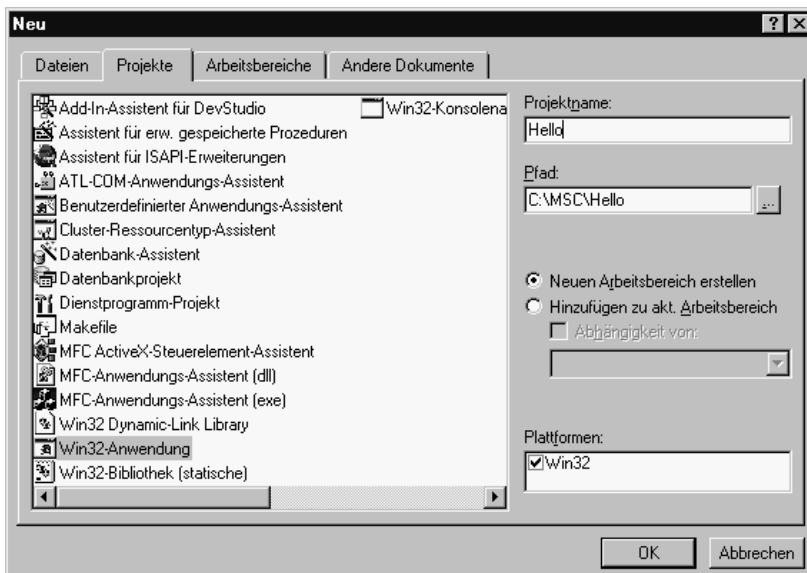


Abbildung 1.19: Anlegen eines neuen Projekts

Öffnen und Schließen eines Projekts

Nachdem ein Projekt neu angelegt wurde, ist es automatisch geöffnet. Mit dem Öffnen eines Projekts werden auch alle Einstellungen aktiviert. Das Bildschirmlayout entspricht demjenigen beim Verlassen des Projekts, alle

ehemals offenen Dateien werden wieder geöffnet, und es werden alle Einstellungen für Compiler und Linker vorgenommen. Es kann an der Stelle weitergearbeitet werden, die zuletzt bearbeitet wurde.

Das Schließen eines Projekts erfolgt über `DATEI|ARBEITSBEREICH SCHLIESSEN`. Dabei werden alle projektspezifischen Einstellungen gesichert, falls dies noch nicht geschehen ist. Außerdem wird noch abgefragt, ob alle offenen Dateien ebenfalls geschlossen werden sollen. Das Schließen aller Fenster kann aber auch über `FENSTER|ALLE SCHLIESSEN` erfolgen.

Für das Öffnen eines Projekts existieren mehrere Möglichkeiten. Man kann im Menü `DATEI` den Punkt `ARBEITSBEREICH ÖFFNEN` anwählen und im folgenden Dateiauswahldialog den entsprechenden Arbeitsbereich auswählen. Weiterhin existiert im Menü `DATEI` eine Liste der zuletzt geöffneten Dateien und Arbeitsbereiche. Ist das gesuchte Projekt in den Arbeitsbereichen enthalten, kann es einfach angewählt werden. Da die Erweiterung des Projekts (`.DSW`) mit dem Developer Studio verknüpft ist, kann ein Projekt auch durch Doppelklick auf die Projektdatei im Explorer oder Dateimanager geöffnet werden. Bei dieser Methode wird in jedem Fall auch das Developer Studio gestartet, und zwar unabhängig davon, ob es bereits lief.

Bearbeiten eines Projekts

Bearbeiten eines Projekts heißt, dem Projekt Dateien hinzuzufügen bzw. aus ihm zu entfernen. Diese Dateien können wiederum verschiedenen Typs sein. Es lassen sich Quelltexte, Ressourcen oder ganze Projekte einfügen. Das Einfügen erfolgt über das `PROJEKT`-Menü und den Punkt `DEM PROJEKT HINZUFÜGEN|DATEIEN`. Die in das Projekt einzufügenden Dateien können anschließend ausgewählt werden. Wird anstelle von `DATEIEN` der Punkt `NEU` angewählt, können in dem Projekt neue Quelltexte, Ressource- oder andere Dateien angelegt werden. Das können auch Word- oder Excel-Dokumente sein, die in einer Beziehung zum Projekt stehen. Darüber hinaus lassen sich auch Unterprojekte zum aktiven Projekt erstellen. Geht es darum, ein vorhandenes Projekt in das aktuelle einzufügen, wird der Menüpunkt `PROJEKT|PROJEKT IN DEN ARBEITSBEREICH EINFÜGEN` gewählt. Dadurch kann man ein großes Projekt in Teilprojekte zerlegen, wobei in Abhängigkeit von etwaigen Änderungen auch die Unterprojekte mitübersetzt werden. Diese Arbeitsweise ist mit Sicherheit für die Arbeit an großen Projekten empfehlenswert, bei kleineren Programmen sind die Vorteile nicht so bedeutend.

Der Menüpunkt `DATEIEN ZUM PROJEKT HINZUFÜGEN` dient dem Einfügen von Quelltexten (`*.C;*.CPP;*.CXX`), aber auch von Ressource-Files, Definition-Files oder Make-Files. Include-Files, die in den Quelltexten mit den `#include`-Statements definiert sind, müssen nicht eingefügt werden. Diese

werden automatisch erkannt und in die Liste der Abhängigkeiten aufgenommen. Wenn in einen vorhandenen Quelltext neue *Include*-Dateien von Hand eingefügt oder entfernt werden, erkennt das System diese nicht sofort. Erst beim Compiler-Lauf werden C- und C++-Abhängigkeiten aktualisiert. Beim Aufruf von ALLES NEU ERSTELLEN oder ERSTELLEN IN STAPEL-VERARBEITUNG aus dem Menü ERSTELLEN werden ebenfalls alle Abhängigkeiten aktualisiert.

Die Online-Hilfe beschreibt, daß Abhängigkeiten auch über den Menüpunkt ERSTELLEN|ALLE ABHÄNGIGKEITEN AKTUALISIEREN aktualisiert werden können. Jedoch ist dieser Menüpunkt nicht vorhanden. Wir gehen davon aus, daß das Aktualisieren automatisch erfolgt. Dieser Automatismus ist sehr bequem und erspart dem Programmierer das manuelle Aktualisieren der Make-Datei.

Eine neue Ressource wird über EINFÜGEN|RESSOURCE (**Strg** **R**) der aktuellen Ressourcen-Datei hinzugefügt. Aus einer Liste muß der gewünschte Ressource-Typ ausgewählt werden. Eine neue Ressource des gewählten Typs wird danach im Editor bereitgestellt. Diese muß nach der Bearbeitung nur noch gesichert werden.

Der Vollständigkeit halber muß noch erwähnt werden, daß auch komplett vorgefertigte Programmkomponenten, wie zum Beispiel der Tip des Tages oder die Tooltips, in das Projekt eingefügt werden können. Das erfolgt über den Menüpunkt PROJEKT|DEM PROJEKT HINZUFÜGEN|KOMPONENTEN UND STEUERELEMENTE. Die *Sammlung von Komponenten und Steuerelementen (Component Gallery)*, die eine umfangreiche Zusammenstellung der vorgefertigten Komponenten enthält, wird in diesem Buch nur kurz behandelt.

Alle Dateien eines Projekts werden im *Arbeitsbereich*-Fenster dargestellt. In einer baumartigen Struktur werden alle Quelltexte, Header-Dateien, Ressourcen- und anderen Dateien angezeigt. Ein Doppelklick genügt, um eine Datei im Editor zu öffnen. Auch im *Arbeitsbereich*-Fenster ist durch das Drücken der rechten Maustaste der Aufruf eines Kontextmenüs möglich, welches abhängig vom jeweiligen Dateityp die Optionen bereitstellt.

Das Entfernen von Quelldateien aus dem Projekt erfolgt durch Drücken der Taste **Entf**. Alternativ dazu kann der Befehl LÖSCHEN im Menü BEARBEITEN aktiviert werden.

Interessant ist noch der Befehl BEREINIGEN im Menü ERSTELLEN. Damit werden alle temporären Dateien des Projekts gelöscht – leider auch die EXE-Datei des Projekts.

Das Arbeitsbereich-Fenster

Das *Arbeitsbereich*-Fenster ist eines des wichtigsten Elemente im Developer Studio. Es stellt sozusagen eine Navigationszentrale dar, die das komfortable Browsen eines Projektes ermöglicht. Die Zeitspanne, um eine bestimmte Stelle in einem langen Quelltext wiederzufinden, wird damit auf wenige Sekunden reduziert. Das Fenster bietet verschiedene Ansichten, die die jeweilige Kategorie strukturiert darstellen. Zur Darstellung wird ein Steuerelement namens *Strukturansicht* (Tree View Control) benutzt, welches die Kategorie in einer baumartigen Struktur zeigt. So ist es möglich, durch einen Doppelklick auf ein Element in der Struktur zu einer Quelltext-Datei zu wechseln bzw. sie zu öffnen, eine Ressource in den Resource-Editor zu laden oder direkt zu einer Klasse oder Variablen einer Klasse zu springen. In Tabelle 1.6 sind die Kategorien und deren Navigationsmöglichkeiten zusammengefasst.

Kategorie	Sprung zu
Dateien	Öffnen oder Aktivieren eines Quelltextfiles oder von Headerdateien, die als Abhängigkeiten definiert sind. Bei Ressourcen wird zum ResourceView gewechselt.
Ressourcen	Öffnen der Ressource und Darstellung im <i>Editor</i> -Fenster, bei Dialogen wird die Toolbar »Steuerungen« eingeblendet.
Klassen	ermöglicht das Navigieren zu Klassen, Methoden der Klassen oder deren Variablen, unabhängig davon, in welchem File sie sich befinden.
Daten	eine Datenbank wird strukturiert dargestellt, Tabellen können betrachtet werden.

Tabelle 1.6: Ansichten des Arbeitsbereich-Fensters

Aktiviert wird das *Arbeitsbereich*-Fenster durch Drücken von **[Alt] [0]** oder einfach mit der Maus. Auch in diesem Fenster ist es möglich, das Kontextmenü der entsprechenden Einträge mit der rechten Maustaste zu aktivieren.



Abbildung 1.20: Arbeitsbereich-Fenster mit ClassView

In Abbildung 1.20 ist das *Arbeitsbereich*-Fenster mit aktivierter *Klassen*-Ansicht dargestellt. Die *Klassen*-Ansicht stellt eine Erweiterung zum Klassen-Assistenten (ClassWizard) dar, mit dem es ebenso möglich ist, neue Funktionen oder Member-Variablen anzulegen. Dazu genügt es, die rechte Maustaste auf das Symbol der Klasse zu drücken. Im Kontextmenü kann dann die gewünschte Funktion angewählt werden. Außerdem ist es mit dieser Methode möglich, Funktionen der Klassen-Browser-Datenbank zu aktivieren, wie zum Beispiel diejenige, die das Anzeigen der abgeleiteten Klassen veranlaßt. Die im ClassView verwendeten Symbole sind in Tabelle 1.7 zusammengefaßt.

Symbol	Bedeutung
	Klasse
	Member-Funktion, protected
	Member-Funktion, privat
	Member-Funktion, public
	Member-Variable, protected
	Member-Variable, privat
	Member-Variable, public
	Schnittstelle
	Methode eines COM-Objekts
	Eigenschaft eines COM-Objekts
	Dialog

Tabelle 1.7: Symbole der Klassen-Ansicht

Die Assistentenleiste

Die Assistentenleiste als Ergänzung zum Klassen-Assistenten hat sich bewährt. Sie liegt in Form einer Symbolleiste vor. Die Assistentenleiste ermöglicht einen Schnellzugriff auf den Klassen-Assistenten, der für das Anlegen neuer Klassen und Member-Variablen sowie das Zuweisen von Windows-Nachrichten zu Funktionen zuständig ist.

Ressourcenvorlagen

Ressourcenvorlagen (Templates) zielen auf die Wiederverwendbarkeit von Code hin. Bei Ressourcen ist das nicht anders. Wenn beispielsweise in einem Projekt alle Dialoge den gleichen Grundaufbau haben, kann ein Template mit dem Dialog-Layout angelegt werden, das dann beim Anlegen neuer Dialogboxen benutzt wird. Dieses Template muß mit dem Befehl **SPEICHERN UNTER** aus dem **DATEI**-Menü als *Ressourcenvorlage* gespeichert werden. Diese Vorlage-Datei (*.RCT) muß in das Verzeichnis `PROGRAMME\MICROSOFT VISUAL STUDIO\COMMON\MSDEV98\TEMPLATE` kopiert werden, um sie nutzen zu können.

Das Benutzen eines Templates erfolgt einfach beim Anlegen der neuen Ressource. Nach dem Aufruf von **EINFÜGEN|RESSOURCE** kann durch einen Mausclick auf das Zeichen  vor einem Ressource-Typ die Liste der Templates zu diesem Typ dargestellt werden. Das gewünschte Template läßt sich danach selektieren und einfügen.

1.6.2 Editieren von Quelltexten

Da das Developer Studio den Texteditor enthält, ist es nicht nötig, auf externe Editoren auszuweichen. Selbst große Projekte lassen sich mit ihm verwirklichen, auch wenn vielleicht einmal eine Funktionalität fehlt. Aber gerade die Integration in die Entwicklungsumgebung und die Zusammenarbeit mit den anderen Komponenten machen ihn unentbehrlich. Die wichtigsten Eigenschaften im Überblick:

- ▼ Multi-Dokumentfähigkeit
- ▼ Keyboard-Shortcuts und Symbolleisten konfigurierbar
- ▼ Suchen und Ersetzen auch über reguläre Ausdrücke, inkrementelle Suche
- ▼ Zusammenarbeit mit Arbeitsbereich und Fehlerliste
- ▼ Syntax-Coloring
- ▼ IntelliSense: Automatisches Ergänzen von Befehlen und Funktionen
- ▼ Tooltips mit Eigenschaften der Befehle und Variablen
- ▼ Springen zu Klammernpaaren
- ▼ Drag & Drop
- ▼ Lesezeichen

Da das Developer Studio als MDI-Applikation implementiert ist, profitiert auch der Editor davon. Mehrere Quelltexte lassen sich parallel bearbeiten. Selbstverständlich kann auch die Zwischenablage benutzt werden, um Cut-Copy-Paste-Operationen durchzuführen.

Dateien lassen sich über die Buttons der Symbolleiste öffnen und sichern. Sie können auch die Funktionen **ÖFFNEN**, **SPEICHEN**, **SPEICHERN UNTER** oder **ALLE SPEICHERN** im **DATEI**-Menü verwenden. Am einfachsten erfolgt das Öffnen von Dateien in der *Datei*-Ansicht des *Arbeitsbereich*-Fensters. Die über Doppelklick ausgewählte Datei wird geöffnet oder, wenn sie bereits offen ist, aktiviert. Gleiches erfolgt unbemerkt in der *Klassen*-Ansicht. Wird die Funktion einer Klasse selektiert, dann wird die Datei in den Editor geladen und die Funktion gesucht. Das Öffnen der Datei erfolgt dabei automatisch. Neue Quelldateien können über **DATEI|NEU|C++-QUELLCODE-**

DATEI (C++-HEADERDATEI) oder über den Button  der Symbolleiste angelegt werden. Neue Dateien, die Sie über den Dialog anlegen, werden automatisch dem Projekt hinzugefügt, wenn das entsprechende Kontrollkästchen aktiviert ist. Voraussetzung ist dabei, daß der Name der Datei angegeben wird. Unbenannte Dateien, wie sie auch mit dem Button der Symbolleiste erzeugt werden, sind erst nach dem manuellen Einfügen in das Projekt Bestandteil desselben.

Das Syntax-Coloring ist eine gute Hilfe beim Editieren. So ist von vornherein an der Farbe eines Wortes im Text zu erkennen, um welches Sprachelement es sich handelt. Schreibfehler sowie das Vergessen von schließenden Kommentarzeichen können so vermieden werden. Dieses Feature ist standardmäßig eingeschaltet. Es kann für jedes Fenster individuell ausgeschaltet werden. Dazu muß im Menü **ANSICHT** der Befehl **EIGENSCHAFTEN** gewählt werden, was auch mit dem Shortcut **[ALT] [↵]** funktioniert. In der Listbox unter *Sprache* muß *keine* selektiert werden, um das Syntax-Coloring auszuschalten. Des weiteren ist es möglich, den einzelnen Sprachelementen andere Farben zuzuweisen. Über **EXTRAS|OPTIONEN** erreicht man eine *Property Page*¹, die verschiedene Einstellungen ermöglicht. Die *Format*-Seite ermöglicht die Einstellungen der Farben und Fonts der verschiedenen Sprachelemente und Fenster.

Eine Einstellung, nämlich die Farbe für Strings, sollten Sie auf jeden Fall ändern. Dadurch ist sofort zu erkennen, ob Anführungszeichen vergessen wurden.



Wem all das nicht genügt, der kann noch eine ASCII-Textdatei im Verzeichnis `\MICROSOFT VISUAL STUDIO\COMMON\MSDEV98\BIN` anlegen, die den Namen `USERTYPE.DAT` trägt. In dieser Datei können Schlüsselwörter (eins pro Zeile) eingetragen werden, die dann in der Farbe für *Benutzerdefinierte Schlüsselwörter* dargestellt werden. Die Änderungen in dieser Datei werden erst nach einem Neustart der Entwicklungsumgebung wirksam.

¹ *Property Page* ist eine mehrseitige Dialogbox, die über Reiter umgeschaltet werden kann.

Im Editor ist eine mehrstufige Undo-/Redo-Funktionalität integriert, die mehrere Aktionen rückgängig machen oder wiederherstellen kann. Es können damit auch Aktionen des Klassen-Assistenten rückgängig gemacht werden. Wieviel Aktionen im Undo-/Redo-Puffer gespeichert werden, hängt von dessen Größe ab. Standardmäßig ist er auf 64 KB eingestellt. Änderungen sind allerdings nicht in der Entwicklungsumgebung möglich. Dazu muß die Registry, die Registrier-Datenbank von Windows, geändert werden. Zu finden ist dieser Wert unter HKEY_CURRENT_USER\SOFTWARE\MICROSOFT\DEVSTUDIO\6.0\TEXTEDITOR\UNDOSIZE. Für unerfahrene Nutzer sind Änderungen in der Registry nicht zu empfehlen, weil dadurch Windows selbst instabil werden kann!

Wenn es erforderlich ist, zwischen zwei oder mehr Stellen im Text hin und her zu springen, ist die Arbeit mit Lesezeichen (Bookmarks) angebracht. Die Tastenkombination **[Strg] [F2]** dient zum Setzen/Löschen eines Lesezeichens in einer Zeile. Sind an den entsprechenden Stellen die Lesezeichen gesetzt, können diese über **[F2]** zyklisch angesprungen werden. Der Shortcut **[Alt] [F2]** ermöglicht das Setzen von Lesezeichen mit Namen. Dadurch können Lesezeichen direkt angesprungen werden. Über **[Strg] [G]** kann der *Gehe zu*-Dialog aktiviert werden, mit dem unter anderem gezielt Lesezeichen angesprungen werden können. Die Funktionen für Lesezeichen sind auch in der *Bearbeiten*-Symbolleiste enthalten.

Der Editor bietet auch einen Vollbild-Modus, der im ANSICHT-Menü über GANZER BILDSCHIRM aktiviert werden kann. In diesem Modus sind alle Elemente des Developer Studios verborgen. Damit ist genügend Platz zum Editieren vorhanden. Verlassen wird dieser Modus durch Drücken von **[Esc]** oder des Buttons GANZER BILDSCHIRM EIN/AUS der Funktionsleiste.

Das Suchen und Ersetzen ist ein Muß für jeden Editor. Nur in den Möglichkeiten unterscheiden sie sich. Über die Tastenkombinationen **[Alt] [F3]** oder **[Strg] [F]** wird die *Suchen*-Dialogbox geöffnet. Das Wort, in dem der Cursor steht, wird als Suchstring vorgegeben. Durch Drücken von WEITERSUCHEN wird der String in der angegebenen Richtung gesucht. Diese Funktionalität zum Suchen ist auch in die *Standard*-Symbolleiste integriert. In ein Kombinationsfeld wird die zu suchende Zeichenkette eingetragen. Durch Drücken von **[↵]** wird zum nächsten Vorkommen gesprungen.

Die Dialogbox bietet zusätzlich die Möglichkeit, nach regulären Ausdrücken zu suchen, wenn zum Beispiel im Suchtext veränderliche Teile enthalten sind. Die Sonderzeichen für reguläre Ausdrücke sind in Tabelle 1.8 enthalten. Es muß bei Verwendung von regulären Ausdrücken auch unbedingt das entsprechende Kontrollkästchen aktiviert werden.

Weitere Hinweise zur Suche mit regulären Ausdrücken finden Sie in der Online-Hilfe unter dem Stichwort »Text suchen«.



Von Vorteil beim Suchen ist es zudem, daß die zuletzt gesuchten Zeichenketten in einem Kombinationsfeld gespeichert werden. So müssen nicht alle Suchtexte neu eingegeben werden. Das Ersetzen von Zeichenketten erfolgt nach dem gleichen Schema. Interessant ist die Funktion *Suchen in Dateien*, mit der mehrere Dateien auf das Vorhandensein eines Suchtextes hin durchforscht werden können, ähnlich dem Programm GREP (siehe Abbildung 1.21). Die gefundenen Textstellen werden im *Ausgabe*-Fenster aufgelistet. Ein Doppelklick in einer der aufgelisteten Zeilen genügt, um an die betreffende Stelle zu springen.

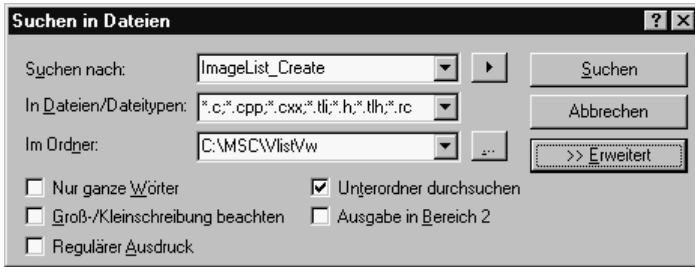


Abbildung 1.21: Suchen von Text in mehreren Dateien und Verzeichnissen

Eine andere Methode, um Textstellen zu finden, ist die *inkrementelle Suche*. Sie wird über die Tastenkombination **[Strg] [I]** (**[Strg] [U] [I]** rückwärts) gestartet. In der Statuszeile erscheint der Text »Inkrementelle Suche:«. Jetzt kann der Suchtext eingegeben werden, wobei nach Eingabe eines jeden Zeichens eine Suche mit dem bis dahin eingegebenen Text erfolgt. Dies erspart unter Umständen das Eingeben von langen Suchtexten. In diesem Suchmodus ist die Tastenkombination **[Strg] [C]** aktiv, die die Unterscheidung zwischen Groß- und Kleinschreibung ein- und ausschaltet. Der aktuell ausgewählte Modus wird in der Statuszeile angezeigt. Die Suche kann auch über **FIND NEXT** (**[F3]**) oder **FIND PREVIOUS** (**[U] [F3]**) fortgesetzt werden. **[Esc]** beendet diesen Suchmodus.

Ausdruck	Beschreibung	Beispiel-Eingabe	Fundstellen
.	ein einzelnes Zeichen	.aus	1aus, Maus, Laus, aber nicht aus
*	vorhergehendes Zeichen kann nicht oder mehrfach vorkommen	Hallo*!	Hall!, Hallo!, Hallo!, Hallooo! ...
+	vorhergehendes Zeichen kann einfach oder mehrfach vorkommen	Hallo+!	Hallo!, Hallo!, Hallooo! ...

Tabelle 1.8: Sonderzeichen für reguläre Ausdrücke

Ausdruck	Beschreibung	Beispiel-Eingabe	Fundstellen
<code>^</code>	Zeilenanfang	<code>^CWnd</code>	wird nur am Zeilenanfang gefunden
<code>\$</code>	Zeilenende	<code>()\$</code>	wird nur am Zeilenende gefunden
<code>[]</code>	ein beliebiges Zeichen, das zwischen den Klammern steht	<code>r[eo]+d</code> <code>n[0-9]</code>	red, reed, rod, rood, nicht reod n0, n1, ..., n9
<code>\(\)</code>	numerierte Ausdruck in der Klammer von 1-9 mit \Nr. kann dieser beim Ersetzen benutzt werden	<code>\(n\)</code> First für Suchen, <code>\1</code> Last für Ersetzen	ersetzt nFirst durch nLast
<code>\~</code>	nicht das folgende Zeichen	<code>\~Mord</code>	Lord, aber nicht Mord
<code>\{ c\! c\}</code>	eine beliebige Kombination der Zeichen	<code>\{H\!N\!V\}</code>	H, HN, HNV, HH ...
<code>\{ \}</code>	findet eine Sequenz der Zeichen in der Klammer	<code>B\{an\}+e</code>	Bane, Banane, nicht Be
<code>[^]</code>	alle Zeichen außer	<code>n[^0-9]</code>	na, nb, nc, nicht n0, n1
<code>\:a</code>	beliebiges alphanumerisches Zeichen	<code>\:a</code>	<code>[a-zA-Z0-9]</code>
<code>\:b</code>	Leerzeichen oder Tab		
<code>\:c</code>	beliebiger Buchstabe	<code>\:c</code>	<code>[a-zA-Z]</code>
<code>\:d</code>	beliebige Dezimalziffer	<code>\:d</code>	<code>[0-9]</code>
<code>\:n</code>	vorzeichenlose Zahl	<code>\:n</code>	122, .25, 122.25
<code>\:z</code>	vorzeichenlose Integer-Zahl	<code>\:z</code>	122, nicht 122.25
<code>\:h</code>	Hexadezimalzahl	<code>\:h</code>	11, 9A, BA
<code>\:i</code>	C/C++ Identifier	<code>\:i</code>	<code>[a-zA-Z_ \$]</code> <code>[a-zA-Z0-9_ \$]+.</code>
<code>\:w</code>	Zeichenkette aus Buchstaben	<code>\:w</code>	<code>[a-zA-Z]+.</code>
<code>\:q</code>	beliebige Anführungszeichen		<code>[' »]</code>
<code>\</code>	hebt die Sonderbedeutung eines der Zeichen in der Tabelle auf	<code>100\\$</code>	100\$

Tabelle 1.8: Sonderzeichen für reguläre Ausdrücke

Weitere Funktionen des Editors sind :

- ▼ das Verschieben oder Kopieren von markierten Textstellen mit der Maus
- ▼ das Teilen von Fenstern oder Öffnen neuer Views eines Fensters
- ▼ Aufzeichnen und Wiedergeben von Tastatur-Aktionen
- ▼ Emulation des BRIEF- oder Epsilon-Texteditors.

IntelliSense

IntelliSense ist der von Microsoft geschaffene Begriff für das automatische Vervollständigen von Anweisungen. Ein Vorteil davon ist, daß Sie nicht mehr so häufig in die Online-Hilfe schauen müssen, um die korrekte Schreibweise oder die Parameter eines Befehls zu suchen. Wie Sie sich sicher vorstellen können, benötigt auch dieses Feature gewisse Ressourcen. Gerade bei Rechnern, die nicht dem optimalen Entwicklungs-PC entsprechen, kann sich dieses Feature als störend erweisen. Deshalb kann es auch abgeschaltet werden. Das erfolgt im Menü EXTRAS unter OPTIONEN und der Registerkarte EDITOR (siehe Abbildung 1.22). Dort kann auch gewählt werden, für welche Kategorien die Funktion aktiv sein soll.

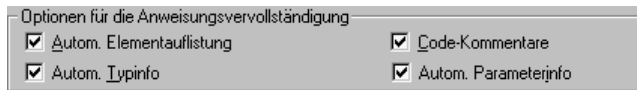


Abbildung 1.22: IntelliSense-Optionen

Die Elementauflistung zeigt alle Member-Variablen und Funktionen an. Sie wird immer dann angezeigt, wenn Sie einen Punkt, »->« oder »:« eingeben (Abbildung 1.23).

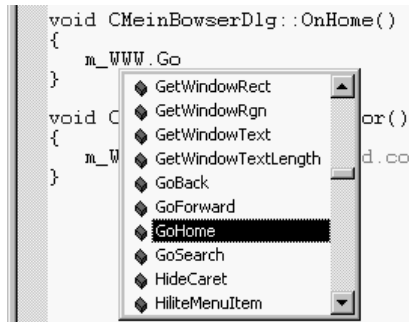


Abbildung 1.23: IntelliSense-Elementauflistung

Sie können, wie im Beispiel, auch die ersten Zeichen eingeben, um dann weiterzublättern. Das Einfügen in den Quelltext kann über die Tasten **↵**, **↩**, **Strg** **↵** oder einem Doppelklick mit der Maus erfolgen. Die Anweisung wird auch vervollständigt, wenn Sie sofort das auf den Bezeichner folgende Element eingeben (z.B. eine runde Klammer, Leerzeichen oder Semikolon). Möchten Sie globale Elemente in den Quelltext einfügen, bewegen Sie den Cursor an eine freie Stelle im Quelltext, und Drücken Sie die Tastenkombination **Strg** **Alt** **T**. Die Bedeutung der

Symbole entspricht der der *Klassen*-Ansicht des *Arbeitsbereich*-Fensters. Weitere Symbole sind in der Online-Dokumentation unter »Symbole in der Elementliste« zu finden.

Die Option *Code-Kommentare* ermöglicht das Anzeigen von Kommentaren der Deklaration oder Definition der Member-Funktionen in der Elementliste. Die Kommentare werden dann in einem eigenen Popup-Fenster neben der Elementliste angezeigt.

Die *Parameterinfo* zeigt die komplette Deklaration einer Funktion in einem Popup-Fenster an (siehe Abbildung 1.24). Der jeweils einzugebende Parameter wird fett dargestellt. Sind mehrere überladene Member-Funktionen vorhanden, können diese über kleine Schaltflächen im Popup-Fenster umgeschaltet werden.



Abbildung 1.24: IntelliSense-Parameterinfo

Die *Typinfo* ist beim Programmieren sehr nützlich. Wissen Sie nicht mehr, wie eine Variable oder Funktion deklariert wurde, führen Sie einfach den Mauszeiger darüber, und als Tooltip wird die gesamte Deklaration angezeigt (Abbildung 1.25 und 1.26).



Abbildung 1.25: Typinfo einer Variablen



Abbildung 1.26: Typinfo einer Funktion

Noch eine Funktion soll die Eingabe erleichtern: das automatische Vervollständigen auf Tastendruck. Dazu müssen von einem Bezeichner so viele Zeichen eingegeben werden, bis der Begriff eindeutig ist. Anschließend wird durch Drücken von **Strg** das Wort vervollständigt. Falls die Eindeutigkeit noch nicht gegeben ist, wird ein Popup-Fenster aufgeblendet, aus dem wiederum ausgewählt werden kann.

All diese Informationen werden, wenn in den Optionen eingestellt, automatisch während der Eingabe des Quellcodes angezeigt. Es ist aber auch möglich, diese Informationen nach der Eingabe nochmals aufzurufen.

Dazu existieren zwei Möglichkeiten: Sie können den Cursor einfach an die Stelle setzen, an der bei Eingabe eines Punktes die Elementauflistung erfolgt, und den Punkt nochmals eingeben. Der Nachteil daran ist, daß der überflüssige Punkt wieder gelöscht werden muß. Die zweite Variante besteht in der Nutzung des Kontext-Menüs. Klicken Sie einfach mit der rechten Maustaste auf ein Element, und wählen Sie dann im Kontext-Menü die gewünschte Funktion an (siehe Abbildung 1.27).

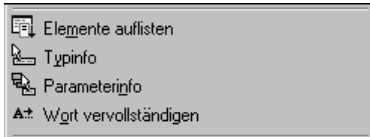


Abbildung 1.27: IntelliSense-Funktionen im Kontext-Menü

VBScript-Makros

Das Developer Studio besitzt auch eine Makrosprache: VBScript. Mit diesem etwas vereinfachten Visual Basic lassen sich Arbeitsabläufe automatisieren. Im Prinzip ist es möglich, das gesamte Developer Studio damit zu steuern.

Für etwas Ungeübte ist es möglich, die Makros aufzeichnen zu lassen. So schafft man meist erst einmal ein Grundgerüst und kann es später anpassen. Man kann ein Makro aber auch einfach programmieren. Alle für Makros notwendigen Funktionen verbergen sich hinter einem Dialog (Abbildung 1.29). Aufgerufen wird dieser über den Punkt MAKRO im Menü EXTRAS. Solange Sie es wünschen, folgt dem Aufruf ein Hinweis, daß Makros über ein Symbol in der Task-Leiste abgebrochen werden können (siehe Abbildung 1.28).

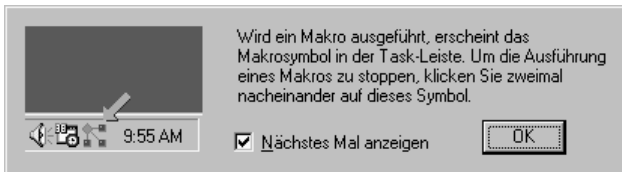


Abbildung 1.28: Hinweis zum Abbruch von Makros

Makros werden in sogenannten Makrodateien gespeichert. Sie unterscheiden sich von normalen Quelltexten dadurch, daß in einer Makrodatei mehrere Makros enthalten sein können. Microsoft liefert bereits eine Makrodatei SAMPLES mit, in der eine Reihe von Beispielmakros enthalten sind. Allerdings muß diese Datei erst aktiviert werden (Abbildung 1.30).

Dazu wird der Dialog ANPASSEN über EXTRAS|ANPASSEN geöffnet. Auf der Dialogseite *Add-Ins und Makrodateien* können die Einstellungen vorgenommen werden. Hier ist eine weitere Besonderheit im Umgang mit Makros zu sehen: Makrodateien können, zum Beispiel projektbezogen, aktiviert oder deaktiviert werden. Alle Makrodateien werden im Verzeichnis \PROGRAMME\MICROSOFT VISUAL STUDIO\COMMON\MSDEV98\MACROS abgelegt. Sie haben die Erweiterung *.DSM*.



Abbildung 1.29: Die Makro-Zentrale

Über die Schaltfläche OPTIONEN kann der Dialog nochmals erweitert werden. Dadurch haben Sie die Möglichkeit, eine neue Makrodatei anzulegen (NEUE DATEI), das Laden von Makrodateien zu aktivieren bzw. zu deaktivieren (GELADENE DATEIEN) und einzelne Makros SYMBOLLEISTEN oder Tastenkombinationen (TASTATUREINGABEN) zuzuweisen.

Wenden wir uns nun dem Schreiben eines Makros zu. Als Beispiel soll hier das Einfügen einer *switch*-Anweisung mit mehreren *case*-Zweigen dienen. Dazu wird als erstes der Dialog *Makros* geöffnet. Nach dem Betätigen der Schaltfläche AUFZEICHNEN müssen der Makroname und eine Beschreibung eingegeben werden. Nach dem Drücken von Ok beginnt die Aufzeichnung. Zu sehen ist das an der Form des Cursors. Die eingblendete Symbolleiste *Aufzeichnen* bietet eine Pausen-Taste sowie eine Stop-Taste zum Beenden der Aufzeichnung. Schreiben Sie ganz normalen Quelltext für eine *switch*-Anweisung, und drücken Sie am Ende die Stop-Taste. Danach wird der aufgezeichnete Quelltext in den Editor geladen. Sie können diesen sofort wieder verlassen und das aufgezeichnete Makro testen. Dazu muß wieder der Makro-Dialog aufgerufen werden. Anschließend wird das Makro mit dem Cursor selektiert und der Button AUSFÜHREN betätigt. Das Ergebnis sollte jetzt der soeben eingegebene Quelltext sein.

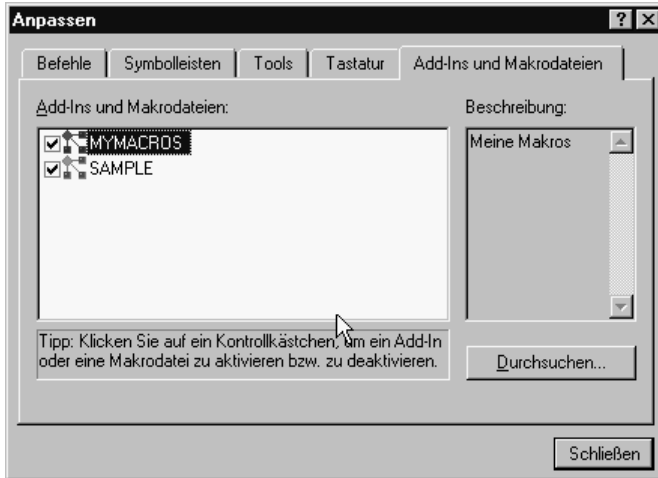


Abbildung 1.30: Aktivieren von Makrodateien

Nachdem das aufgezeichnete Makro ausgeführt wurde, werden Sie feststellen, daß dieses nicht sehr flexibel ist. Es wäre zumindest schön, die Anzahl der *case*-Zweige eingeben zu können und festzulegen, ob ein *default*-Statement benötigt wird oder nicht. Dies wurde mit dem folgenden Quelltext realisiert. Zu sehen ist dabei die Reaktion auf Eingaben durch den Nutzer sowie die Verwendung von Schleifen und *if*-Anweisungen.

```
Sub CaseSequenz()
'DESCRIPTION: case-Anweisung platzieren
Dim nSequ, i, bDef
    nSequ = InputBox("Anzahl der Sequenzen?", "Case Sequenz", "3")
    bDef = MsgBox("Default-Statement einfügen?", vbYesNo, "Case
Sequenz")
    ActiveDocument.Selection = "switch ()"
    ActiveDocument.Selection.NewLine
    ActiveDocument.Selection = "{"
    ActiveDocument.Selection.NewLine
    for i=1 to nSequ
        ActiveDocument.Selection = "case : "
        ActiveDocument.Selection.NewLine
        ActiveDocument.Selection = "{"
        ActiveDocument.Selection.NewLine
        ActiveDocument.Selection.NewLine
        ActiveDocument.Selection = "break;"
        ActiveDocument.Selection.NewLine
    ActiveDocument.Selection.Backspace
```

```

        ActiveDocument.Selection = "}"
        ActiveDocument.Selection.NewLine
        ActiveDocument.Selection.Backspace
    next
    if bDef=vbYes then
        ActiveDocument.Selection = "default:"
        ActiveDocument.Selection.NewLine
        ActiveDocument.Selection = "{"
        ActiveDocument.Selection.NewLine
        ActiveDocument.Selection.Backspace
        ActiveDocument.Selection = "}"
        ActiveDocument.Selection.NewLine
        ActiveDocument.Selection.NewLine
        ActiveDocument.Selection.Backspace
    end if
    ActiveDocument.Selection.Backspace
    ActiveDocument.Selection = "}"
    ActiveDocument.Selection.NewLine
End Sub

```



Ändern Sie das Makro so ab, daß zusätzlich noch abgefragt wird, ob die *break*-Anweisungen erforderlich sind oder nicht.

Schreiben Sie ein Makro, das für Funktionen eine Beschreibung in den Quelltext einfügt. Dabei sollen der Funktionsname, der Rückgabewert, das Datum und eine kurze Beschreibung der Funktion eingegeben werden können.



Die Makrodatei MYMACROS.DSM mit den Lösungen befindet sich auf der beiliegenden CD.

1.6.3 Kompilieren und Linken

Um ein fertiges Programm zu erzeugen, müssen die Quelltexte in einen für den Computer verständlichen Code übersetzt werden. Der gesamte Vorgang wird als *Erstellen* bezeichnet. Er läuft in zwei Stufen ab:

5. Das *Kompilieren*, bei dem der Quelltext auf Korrektheit überprüft und daraus ein Zwischencode erzeugt wird (Objektdateien *.OBJ).
6. Das *Linken*, bei dem das Programm aus den einzelnen Teilen (Objektdateien und Libraries) zusammengesetzt und auf seine Vollständigkeit hin geprüft wird.

Da auch Compiler- und Linker-Aufrufe in die Entwicklungsumgebung integriert sind, profitiert man zum Beispiel davon, daß die Fehlermeldungen in das *Ausgabe*-Fenster geschrieben werden. Dadurch ist es möglich, durch Drücken von F1 Hilfe zu dem jeweiligen Fehler zu erhalten, oder

mittels Doppelklick direkt zur fehlerhaften Stelle im Text zu springen. Mit den Tasten **F4**/**↕**/**F4** kommt man zum nächsten/vorherigen Fehler der Liste.

Daneben besteht noch die Möglichkeit, den Erstellungsvorgang in den Hintergrund zu verlagern. Das wird durch die Multitasking-Fähigkeiten von Windows bzw. das Multi-Threading des Developer Studios ermöglicht. Jedoch sollte der Rechner gut bestückt sein, um diese Fähigkeiten richtig nutzen zu können. Dann kann entweder zu einem anderen Windows-Programm gewechselt oder sogar das Editieren der Quelltexte fortgesetzt werden.

Die Aufrufe zum Übersetzen von Programmen sind im Menü ERSTELLEN zu finden. Für den Schnellzugriff hält die Symbolleiste *Minileiste erstellen* einige Programmaufrufe bereit (siehe Kapitel 2.5.3). Um Zeit zu sparen, kann bei der Erkennung von Fehlern der Erstellungsvorgang über ERSTELLEN ANHALTEN (**Strg** **Pause**) auch abgebrochen werden.

Release- und Debug-Version

In das Developer Studio ist ein Projekt so integriert, daß unterschiedliche Versionen durch einfaches Umschalten erzeugt werden können. Diese werden grundlegend in *Debug* und *Release* unterschieden, wobei der Name schon über ihre Verwendung Auskunft gibt. Je nach Projektkonfiguration (Abbildung 1.31) können noch weitere Versionen (z. B. Static Debug/Static Release) erzeugt und verwaltet werden.

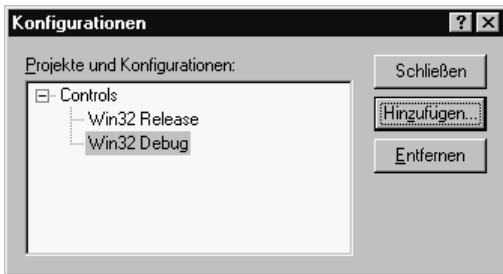


Abbildung 1.31: Verwalten von Projektkonfigurationen

Das Erstellen erfolgt über das Menü ERSTELLEN|KONFIGURATIONEN und dem Betätigen der Schaltfläche HINZUFÜGEN (siehe Abbildung 1.32).

Die Debug-Version enthält Code zum Debuggen einer Applikation. Da dieser Code die Anwendung erheblich vergrößert und an Kunden nicht weitergegeben werden sollte, gibt es die Release-Version. Die beiden Varianten werden durch unterschiedliche Einstellungen der Compiler- und Linker-Optionen erzeugt. Ein mühsames Umstellen vor der Auslieferung

kann dadurch erspart werden. Das Umschalten der Konfiguration erfolgt über das Kombinationsfeld in der *Erstellen*-Symbolleiste oder über den Menüpunkt AKTIVE KONFIGURATION FESTLEGEN im Menü ERSTELLEN.

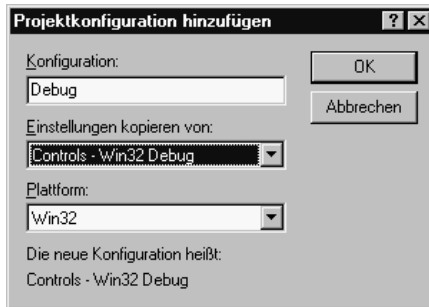


Abbildung 1.32: Anlegen einer neuen Projektkonfiguration

Optionen für Compiler und Linker

Die Optionen für Compiler und Linker beinhalten eine fast unüberschaubar große Anzahl von Schaltern und Einstellungsmöglichkeiten. Der Aufruf des Dialogfensters erfolgt über PROJEKT|EINSTELLUNGEN oder **[Alt] [F7]**. Um die Vielzahl der Optionen etwas übersichtlicher zu gestalten, ist der Dialog mit einem Register-Steuerelement versehen (Abbildung 1.33). Über die Reiter wird eine Klasse von Optionen ausgewählt, die dann die entsprechenden Einstellungen zulässt. Da auch das oft nicht ausreicht, kann noch einmal zwischen verschiedenen Kategorien umgeschaltet werden, die aus einer Liste zu wählen sind (z. B. beim Linker). Damit wird das Aussehen der Dialogseite noch einmal verändert.

In der linken Hälfte der Dialogbox wird angegeben, für welche Konfigurationen die Einstellungen gelten sollen. Sollen die Einstellungen für alle vorgenommen werden, kann in der Liste der Eintrag *Alle Konfigurationen* selektiert werden. Optionen, die für die gewählten Konfigurationen gleich sind, werden in der Textfarbe dargestellt, während bei unterschiedlichen Einstellungen das Feld grau unterlegt ist. Im unteren Teil des Dialoges unter *Projekt Optionen* werden die per Menü eingestellten Optionen als Schalter so angezeigt, wie sie auch in der Kommandozeilenversion von Compiler und Linker verwendet werden müßten.

Die Integration von Compiler und Linker in das Developer Studio macht das mühsame Zusammensuchen dieser Optionen bzw. das Ändern des Make-Files überflüssig. Die Einstellungsmöglichkeiten gehen sogar so weit, daß für jedes einzelne CPP-File noch gesonderte Optionen eingetragen werden können. Dazu muß nur der Baum der Debug- oder Release-Version expandiert und das entsprechende File angewählt werden.

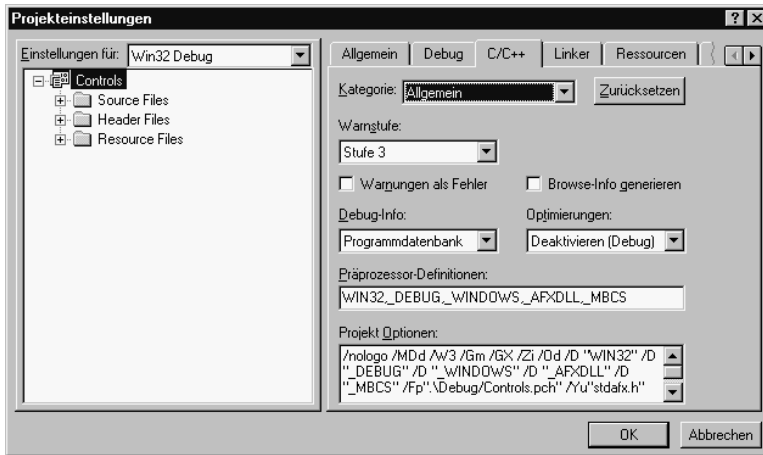


Abbildung 1.33: Einstellen von Compiler- und Linker-Optionen

Beim Anlegen eines neuen Arbeitsbereiches setzt Visual C++ für alle Schalter Standardwerte entsprechend dem Projekttyp ein. Deshalb soll an dieser Stelle nur auf einige wenige Schalter eingegangen werden.

Auf der Seite *Allgemein* erfolgt die Auswahl, wie die MFC-Klassenbibliothek genutzt werden soll, es sei denn, das Projekt erfordert keine MFC. Es kann zwischen einer statischen und der DLL-Version der MFC gewählt werden. Die statische Version der MFC wird beim Linken mit in die EXE-Datei eingebunden. Dadurch wird die Programmdatei zwar vergrößert, aber zugleich werden auch die Zugriffe auf Routinen beschleunigt. Wird die DLL-Version gewählt, ist darauf zu achten, daß die MFC-DLL auch auf dem System vorhanden sein muß, auf dem die Applikation ausgeführt wird.

Die Einstellung von Präprozessor-Anweisungen, die unter anderem zur bedingten Kompilierung benötigt werden, erfolgt auf der Seite *C/C++* in der Kategorie *Preprocessor*.

Auf der Seite *Browse Informationen* wird eingestellt, wie die Datei für den Klassenbrowser benannt wird. Zusätzlich kann noch festgelegt werden, daß für jede Quelldatei eine Sourcebrowser-File (*.SBR) angelegt wird. Dazu muß die Option *BROWSE-INFORMATIONSDATEI ERSTELLEN* aktiviert sein. Der Klassenbrowser ermöglicht es unter anderem, eine grafische Darstellung zu erzeugen, die zeigt, von welchen Klassen die selektierte Klasse abgeleitet ist, oder an welchen Stellen im Quelltext diese Klasse benutzt wird.

Es soll noch einmal darauf hingewiesen werden, daß die Einstellungen der Projekt-Optionen nur dann verändert werden sollten, wenn es dringend notwendig ist. Falsche Einstellungen können dazu führen, daß das Pro-



gramm nicht mehr übersetzt werden kann. Im Dialog der Projekteinstellungen ist auch eine HILFE-Schaltfläche vorhanden, bei deren Betätigung eine Erläuterung aller Einstellungsmöglichkeiten der Seite erfolgt.

1.7 Dokumentation und Online-Hilfe

1.7.1 Gedruckte Handbücher

Die Beigabe von gedruckten Handbüchern zu Programmiersprachen wird heutzutage auf ein Minimum beschränkt. Grund dafür sind nicht zuletzt die häufig wechselnden Versionen, die Handbücher schnell zu Altpapier werden lassen. So verhält es sich auch mit Visual C++ 6.0. Zum Lieferumfang gehört eine »Broschüre« mit dem Titel *Visual Studio Entwickeln für ...*. Alle anderen Handbücher sind auf der CD-ROM enthalten. Die Handbücher liegen nur zum Teil in deutscher Sprache vor, der restliche Teil ist in englischer Sprache verfügbar. Dem Programmpaket liegt auch ein Bestellschein für die gedruckten Bücher von Microsoft Press bei. Folgende Handbücher können in gedruckter Form erworben werden:

- ▼ *Microsoft Visual C++ 6.0 Programmierhandbuch*: Darin ist eine Beschreibung der Entwicklungsumgebung sowie des Ablaufs beim Entwickeln von Projekten enthalten (Developer Studio, Projekte u.a.).
- ▼ *Microsoft Visual C++ Reference Library*: In der Referenz sind alle Klassen, Funktionen und globalen Variablen in alphabetischer Reihenfolge beschrieben.

1.7.2 Online-Dokumentation

Der gesamte Inhalt der verfügbaren Handbücher ist auf den beiden MSDN-CDs enthalten und dient als Ersatz für die gedruckte Dokumentation. Bei Bedarf können auch einzelne Teile der Dokumentation ausgedruckt werden. Neben einigen Erweiterungen bei den Hilfe-Dateien, die nicht in gedruckter Form vorliegen, können die Beispielprogramme direkt aus der Hilfe auf die Festplatte kopiert werden. Weiterhin existieren komfortable Suchmethoden sowie die Möglichkeit, Themen zu Favoriten hinzuzufügen. Die Online-Dokumentation ist vom Developer Studio aus aufrufbar. Sämtliche Hilfe-Dateien sind im HTML-Format implementiert. Neben einer optischen Aufwertung der Hilfe können direkt Links in das *World Wide Web* integriert werden.

Das Hilfesystem ist in einer hierarchischen Struktur organisiert. In der obersten Ebene sind die *Bücher* enthalten. Innerhalb der Bücher sind *Kapitel* und *Dokumente* enthalten, die auf mehrere Hierarchiestufen verteilt sein können. Die Dokumente enthalten den eigentlichen Text, der selbst wiederum Querverweise, Grafiken, Popups und anderes enthalten kann.

Bedienung

Die Online-Dokumentation ist über mehrere Schnittstellen an die Entwicklungsumgebung angekoppelt. Im Menü ? sind verschiedene Punkte für den Zugriff auf die Dokumentation enthalten. Der Punkt INHALT öffnet das Register *Inhalt* der MSDN-Library. Über den Punkt springt man zum Such-Register. Die Auswahl des Indexes aktiviert den Reiter INDEX, um nach Stichworten zu suchen.

INDEX entspricht der Suche im Index eines Buches. Zuerst wird das Schlüsselwort eingegeben. Dabei werden in der Listbox darunter alle gefundenen Referenzen angezeigt, in der dann die gewünschte Stelle mittels Doppelklick ausgewählt und anschließend im rechten Fenster angezeigt wird.

SUCHEN erlaubt es, eine Volltextsuche in allen Büchern bzw. eine Auswahl aus denselben durchzuführen. Die Vorgehensweise ist dabei die gleiche. Zuerst wird der Begriff eingegeben. Es kann auch nach mehreren Begriffen über logische Verknüpfungen gesucht werden. Über den Button THEMEN AUFLISTEN wird die Suche gestartet. Das Ergebnis der Suche wird in einer Liste dargestellt. Der Suchbegriff kann aus einer beliebigen Kombination von Buchstaben und Ziffern bestehen, jedoch existiert eine Liste mit reservierten Worten (and, or, not, near). Gesucht werden kann nach einem einzelnen Wort, einer Phrase oder einem Ausdruck mit Wildcards (siehe Tabelle 1.9).

Der Menüpunkt HILFE|TASTATURBELEGUNG öffnet ein Fenster, das zu allen verfügbaren Befehlen der Entwicklungsumgebung die Hotkeys anzeigt. Die Zuordnung von Tastenkombinationen zu Befehlen erfolgt im Menü EXTRAS|ANPASSEN|TASTATUR.

Zeichen/Wort	Bedeutung	Beispiel-Eingabe	Suchergebnis
*	Ersatz für beliebige Zeichen	*86	8086, 80386, 80486 ...
?	Ersatz für ein beliebiges Zeichen	80?86	80386, 80486 ...
". "	kennzeichnet eine Phrase	"options dialog"	sucht gesamten Begriff
AND	beide Ausdrücke im selben Punkt enthalten	cwinapp AND mfc	cwinapp und mfc müssen vorhanden sein
OR	einer der beiden Ausdrücke im Suchergebnis enthalten	cwinapp OR mfc	cwinapp oder mfc müssen vorhanden sein
NOT	Ausdruck nicht im Suchergebnis enthalten	cwinapp NOT mfc	cwinapp muß vorhanden sein, mfc darf nicht vorhanden sein
NEAR	die Begriffe müssen innerhalb einer festgelegten Anzahl von Worten aufeinanderfolgen	cwinapp NEAR mfc	mfc darf maximal n Worte von cwinapp entfernt stehen

Tabelle 1.9: Wildcards und Operatoren für Volltextsuche

Der Menüpunkt **UNTERMENGE DEFINIEREN** im Menü **ANSICHT** der MSDN-Library dient zum Anlegen eigener Sammlungen von Ausschnitten aus den Handbüchern. Bei einer Suche kann man sich dann auf diese Teilmenge beschränken. Über **AKTIVE UNTERMENGE** wird festgelegt, welche Teilmenge standardmäßig benutzt wird. Weiterhin enthalten im Menü ? ist die Hilfe zum **TECHNISCHEN SUPPORT**. Über den Punkt **MICROSOFT IM WEB** können direkt einige Seiten im Internet aufgerufen werden. Die Darstellung der Seiten erfolgt dann direkt in der Online-Hilfe.

Im MSDN-Fenster können die Bücher mit einem Mausklick auf das Pluszeichen geöffnet werden. Ist die gesamte Struktur bis zum gewünschten Dokument expandiert, kann dieses mit einem Doppelklick zur Anzeige gebracht werden.



Abbildung 1.34: Symbolleiste der Online-Dokumentation

Die in Abbildung 1.34 dargestellte Symbolleiste dient zum Navigieren in der Online-Hilfe. Der dritte Button dient zum Öffnen des nächsten Dokuments der Dokumentation, der vierte zum Öffnen des vorhergehenden. Bewegt man sich über Querverweise von Dokument zu Dokument, kann mit den Buttons **ZURÜCK** und **VORWÄRTS** der Sprung zum vorhergehenden/nächsten betrachteten Dokument erfolgen. Bei allen Aktionen im Editorfenster und der Toolbar, die andere Dokumente aufrufen, kann der Baum über den Button **SUCHEN** nachgeführt werden.

Kontextsensitive Hilfe

Kontextsensitive Hilfe bedeutet, Informationen zu dem Begriff oder zu der Zeile zu erhalten, in welcher der Cursor in der Entwicklungsumgebung steht, ähnlich den Kontextmenüs. Der Aufruf ist verknüpft mit dem Drücken der Taste **[F1]**, wenn der Cursor in einem Schlüsselwort steht oder sich in einer Zeile mit einer Fehlermeldung befindet. Ähnlich der Suche über **INDEX** wird nach dem Drücken der Taste **[F1]** eine Liste mit allen Vorkommen des gesuchten Begriffs angezeigt. Aus dieser Ergebnisliste kann dann das entsprechende Dokument ausgewählt werden.

In Dialogboxen- auch größeren Dialogen mit Property Sheets- wird über **[F1]** die Hilfe zum aktiven Fenster aufgerufen oder ein kurzes Popup-Fenster geöffnet. Das gleiche Ergebnis wird beim Drücken des meist vorhandenen Hilfe-Buttons erreicht.

Eine weitere Form der kontextsensitiven Hilfe sind die Anzeigen in der Statuszeile. Zu jedem Menüpunkt wird in der Statuszeile ein Kurztext mit der Erklärung angezeigt, sobald sich der Cursor über dem Element befin-

det. Ebenso wird ein Text angezeigt, wenn sich der Mauscursor über einen Button der Toolbar hinwegbewegt. Zusätzlich werden bei kurzem Verweilen des Mauscursors über einen Button sogenannte *Tool Tips* angezeigt. Dabei erscheint am Cursor ein gelbes rechteckiges Fenster, das eine Beschreibung seiner Funktion beinhaltet.

1.7.3 Weitere Informationsquellen

In der Dienstprogramme-Gruppe zu Visual C++ ist ein weiteres Icon enthalten, mit dem man auf die Hilfe von OLE-Werkzeugen zugreifen kann. Im Verzeichnis \PROGRAMME\MICROSOFT VISUAL STUDIO\COMMON\VC98 sind HTML-Dokumente mit einer Vielzahl von Informationen und Fehlerkorrekturen enthalten.

1.7.4 Referenzen auf elektronische Dokumente

In diesem Buch finden sich von Zeit zu Zeit Referenzen auf elektronische Handbücher. Dabei wurde meist das Stichwort und zum Teil der Name des Handbuchs angegeben, in dem der Begriff erklärt wird. Das ist deshalb wichtig, da viele der Begriffe in mehreren Handbüchern zu finden sind. Des weiteren sind Verweise auf Namen von Funktionen enthalten und gegebenenfalls der Titel des Handbuchs. Die jeweiligen Einträge zu den Stichworten können Sie über INDEX oder SUCHEN in der MSDN-Library erreichen.

1.7.5 Quelltexte zur MFC

Ein großer Vorteil der Klassenbibliothek MFC ist es, daß alle Quelltexte im Lieferumfang enthalten sind. Damit wird zum einen das Debuggen in MFC-Funktionen möglich und zum anderen besteht die Möglichkeit, Änderungen vorzunehmen und anschließend die MFC neu zu übersetzen. Nachteile von Änderungen sind die Inkompatibilität mit anderen Systemen oder Programmen sowie die Gefahr, daß sich Fehler einschleichen. Die Quelltexte zur MFC sind im Verzeichnis \PROGRAMME\MICROSOFT VISUAL STUDIO\VC98\MFC\SRC zu finden. Die Dateien bieten auch einen Einblick, wie Profis programmieren, und können ergänzend zur Dokumentation genutzt werden. Einige Dateien der MFC befinden sich im Verzeichnis \PROGRAMME\MICROSOFT VISUAL STUDIO\VC98\MFC\INCLUDE.



Hello – Das erste Programm

2

Kapitelübersicht

2.1	Ein kurzer Abriß in (Visual) C++	70
2.1.1	Begriffe und Definitionen	70
2.1.2	Aufbau eines MFC-Programms	72
2.1.3	Das Message-Map-Konzept	73
2.2	Die Klasse CWinApp	75
2.2.1	Ableiten einer Anwendungsklasse von CWinApp	75
2.2.2	Überlagern der Methode InitApplication	78
2.2.3	Überlagern der Methode ExitInstance	79
2.2.4	Globale Funktionen	79
2.2.5	Das Programm-Icon	81
2.2.6	Der Programmcursor	83
2.2.7	Daten-Member in CWinApp	85
2.2.8	Auswerten der Kommandozeile	85
2.2.9	Multitasking in eigenen Programmen	88
2.3	Die Klasse CWnd	95
2.3.1	Fensterklassen	95
2.3.2	Erzeugen eines neuen Fensters	97
2.3.3	Der Konstruktor von CMainWindow	98
2.3.4	Überlagern der OnPaint-Methode	100
2.3.5	Erzeugen der Message-Map	103
2.3.6	Zerstören eines Fensters	105
2.4	Das Beispiel Schritt für Schritt	106
2.4.1	Das Anlegen des Hello-Projekts	106
2.4.2	Hello – Schritt 1	109
2.4.3	Hello – Schritt 2	111
2.4.4	Hello – Schritt 3	112
2.4.5	Hello – Schritt 4	113

Seit Brian W. Kernighan und Dennis M. Ritchie vor mehr als 15 Jahren mit ihrem Buch »The C-Programming Language« die Ära der Sprache C eingeleitet haben, *muß* jedes Buch zu diesem Thema mit einem »hello world«-Beispiel beginnen. Davon soll auch hier keine Ausnahme gemacht werden: Das Buch erklärt die Grundlagen des Themas anhand eines Programms, das nicht mehr zu tun hat, als den Schriftzug »hello world« auf dem Bildschirm auszugeben.

Das hier verwendete Programm ist zwar nicht mehr so kurz wie sein Urahn von damals, dafür aber in C++ geschrieben, Windows-fähig und objektorientiert. Unverändert ist die Idee, die dahintersteckt: Nach dem ersten Kapitel hat man genug Wissen, um sich ein Bild zu machen – und doch noch so wenig, daß man den Rest des Buches auch noch lesen muß.

Das erste Kapitel dieses Buches gibt eine Einführung in die Programmierung von Windows-Anwendungen mit Microsoft Visual C++. Es erklärt die Verwendung zweier sehr wichtiger Klassen der Foundation-Classes-Library am Beispiel des »hello world«-Programms. Darüber hinaus liefert es eine Menge an Informationen, die im Umfeld der Programmierung unerlässlich sind und den Einstieg in die Arbeit mit Visual C++ erleichtern.

Um nicht immer das »hello world«-Programm schreiben zu müssen, soll das hier verwendete Beispielprogramm zukünftig kurz HELLO heißen.

Das Programm erzeugt nach dem Starten ein Hauptfenster und zeigt darin den Text »hello world« zentriert an. Die Fenster-Funktionen *Maximieren*, *Minimieren*, *Verschieben*, *Größe ändern* und *Beenden* können sowohl mit der Maus als auch über das System-Menü bedient werden. Darüber hinaus besitzt das Programm alle Standard-Eigenschaften von Windows-Anwendungen, z.B. was den Aufbau des Hauptfensters oder das korrekte Verhalten im Zusammenspiel mit anderen Bildschirmfenstern betrifft.

Da in der heutigen Zeit das einfache Ausgeben von Text keinen mehr vom Hocker reißt, wurde HELLO noch etwas erweitert und optisch ansprechender gestaltet.

2.1 Ein kurzer Abriß in (Visual) C++

2.1.1 Begriffe und Definitionen

Ungarische Namenskonvention

Wer schon einmal Windows-Programme geschrieben hat, wird sich wahrscheinlich zunächst über Variablennamen wie *hWndMain*, *lpszFileName* oder *nCount* gewundert haben. Nach dem ersten Schrecken stellt sich dann heraus, daß hinter den monströsen Namen eine Systematik steckt, die das Lesen von Quelltexten erleichtert (wenn man sie denn durchschaut hat). Diese Systematik wird gemeinhin als *ungarische Namenskon-*

vention bezeichnet und geht auf den Programmierer Charles Simonyi zurück, der bei Microsoft für Anwendungsprogramme wie Word oder Multiplan verantwortlich war.

Die ungarische Namenskonvention stellt eine Beziehung zwischen Programmnamen und ihren Eigenschaften her. So steht das Präfix *h* beispielsweise für *Handle*, *lp* für *Long Pointer to String which is Zero Terminated*, also für einen FAR-Zeiger auf eine Null-terminierte Zeichenkette, und *n* für eine *number*, also eine numerische Ganzzahl. Wenn man beim Programmieren die wenigen Regeln der ungarischen Namenskonvention einhält, fällt das Lesen der eigenen Programme hinterher sehr viel leichter.

Sowohl das SDK als auch die MFC verwenden die ungarische Namenskonvention. Zusätzlich gibt es Vereinbarungen über die Benennung von Funktionen, Klassen und Member-Variablen. Tabelle 2.1 gibt eine Übersicht der gebräuchlichsten Präfixe.

Präfix	Bedeutung
b	Boolesche Variable, die nur die Werte TRUE oder FALSE annehmen kann
c	Character-Variable
dw	Double Word, vorzeichenloser 32-Bit-Wert
h	Handle
sz	String Zero, Null-terminierte Zeichenkette
lp	Long Pointer, FAR-Zeiger
p	Pointer, NEAR-Zeiger
l	Long, vorzeichenbehafteter 32-Bit-Wert
w	Word, vorzeichenloser 16-Bit-Wert
pt	Point, eine zweidimensionale Punkt-Datenstruktur
f	Flag, vorzeichenloser 16-Bit-Wert zur Speicherung von Bitflaggen
n	Number, vorzeichenbehafteter 16-Bit-Wert
cb	Count of Bytes, wie n
fn	Function, meist eine Callback-Funktion

Tabelle 2.1: Ungarische Namenskonvention

Funktionsnamen

Die Namen von Funktionen beginnen mit einem Großbuchstaben. Sie bestehen normalerweise aus zwei oder mehr Silben, von denen die erste ein Verb und die zweite ein Substantiv ist. Das Verb beschreibt die Aufgabe der Funktion, und das Substantiv liefert einen Hinweis auf das Ziel (das Objekt) der Aktion. Funktionsnamen mit diesem Aufbau sind beispielsweise *SendMessage*, *GetProfileString* oder *FreeProcInstance*. Des weiteren fällt

sofort auf, daß jedes Teilwort mit einem Großbuchstaben beginnt, um den Namen lesbarer zu machen. Da in C und C++ Groß- und Kleinschreibung signifikant ist, sollte man sich diese Regel unbedingt merken.

Klassennamen

Die Namen von Klassen beginnen in den Microsoft Foundation Classes mit einem großen »C« (wie Class) und sind damit auf den ersten Blick von anderen Programmobjekten zu unterscheiden. Diese Konvention gilt nicht für Instanzen von Klassen; es gibt also die Klasse *CWnd*, eine Instanz aber hat beispielsweise den Namen *MyWnd* (und nicht *CMyWnd*).

Member-Variablen

Member-Variablen einer Klasse besitzen immer das Präfix »m_«, damit sie sofort als solche erkannt werden können. Im Zusammenwirken mit der ungarischen Namenskonvention führt dies zu umständlichen Namen wie *m_hBrushCtlBk* oder *m_pszAppName*. Um eine Übereinstimmung mit der Microsoft-Dokumentation und den MFC-Quellen zu erzielen, werden die Konventionen für Klassen- und Member-Namen auch in diesem Buch verwendet.

2.1.2 Aufbau eines MFC-Programms

Vorgehensweise

Dieses Kapitel beschreibt, wie ein einfaches Programm mit Hilfe der Foundation Classes aufgebaut werden kann. Die Ausführungen basieren auf der Annahme, daß es sinnvoll ist, zuerst die allgemeinen Möglichkeiten einer Programmierumgebung vorzustellen und erst später zu den speziellen Eigenschaften überzugehen – selbst, wenn dadurch zunächst etwas mehr Aufwand betrieben wird.

Es werden zunächst die Grundlagen der Windows-Programmierung dargestellt. Da diese Vorgehensweise der Programmierung zwar immer möglich ist, aber die Möglichkeiten von Visual C++ und des Developer Studios nur beschränkt nutzt, wird ab dem nächsten Kapitel der MFC-Anwendungs-Assistent (AppWizard) zur Programmerstellung verwendet. Die Erläuterung der Dokument-Ansicht-Architektur mit dem kompletten Paradigma der Foundation Classes wird aber erst in späteren Kapiteln erfolgen. Zunächst wird nur die Eigenschaft der einfachen Erstellung eines *Programmgerüsts* genutzt.

Eine Einsparung an Programmieraufwand wird in jedem Fall erreicht. Wenn das Programm zusätzlich noch dokumentorientiert arbeitet, kann sehr viel Zeit eingespart werden. Das heißt, daß es im wesentlichen dazu dient, eine oder mehrere Arten von Dateien auf eine bestimmte Art und Weise zu bearbeiten.

Grundlagen der MFC-Programmierung

Um ein einfaches MFC-Programm zu schreiben, werden wenigstens zwei Klassen benötigt. Die erste stellt ein Anwendungsobjekt zur Verfügung – das Gegenstück zu *WinMain* bei der SDK-Programmierung. Die zweite Klasse hat die Aufgabe, das Hauptfenster der Applikation zu erzeugen, auf dem Bildschirm darzustellen und zu pflegen. Zwei der wesentlichen Bestandteile jedes Windows-Programms sind also bereits in Klassen verborgen: in einer Anwendungsklasse und einer Fensterklasse. Beide Klassen sind in der benötigten Form zwar nicht vorhanden, können jedoch durch Vererbung leicht gewonnen werden.

Die Basisklasse für eine eigene Anwendungsklasse heißt *CWinApp* und wird von den Foundation Classes zur Verfügung gestellt. In dieser Klasse ist das typische Hauptprogramm *WinMain* einer C-Windows-Applikation verborgen. Sie führt die Initialisierung des Programms durch und betreibt die Nachrichtenschleife. Da die Klasse *CWinApp* lediglich einen leeren Programmrumpf enthält und kein Hauptfenster erzeugt, muß jede MFC-Anwendung ihre eigene Anwendungsklasse von *CWinApp* ableiten. Diese erbt dann die volle Funktionalität der Basisklasse und ergänzt sie an den nötigen Stellen durch Überlagern von Methoden. In jedem Visual C++-Programm kann *nur eine* von *CWinApp* abgeleitete Klasse existieren.

Für das Erzeugen von Fenstern sind mehrere Klassen vorhanden. Die Klasse für das Hauptfenster wird von *CFrameWnd* abgeleitet. Auch hier werden die wichtigsten Eigenschaften von der Basisklasse zur Verfügung gestellt, während die nötigen Anpassungen wiederum durch Überlagern von Methoden vorgenommen werden.

Um das derart mit neuen Klassendeklarationen ausgestattete MFC-Programm zum Leben zu erwecken, gibt es nicht mehr viel zu tun. Zuerst werden die erforderlichen Member-Funktionen der abgeleiteten Klassen ausprogrammiert. Statt nun ein Hauptprogramm im herkömmlichen Sinn zu schreiben, wird lediglich eine Instanz der abgeleiteten Anwendungsklasse definiert, also eine globale oder statische Variable angelegt. Der dadurch aufgerufene Konstruktor der Anwendungsklasse führt nacheinander die Initialisierung, die Anzeige des Hauptfensters und den Sprung in die Nachrichtenschleife aus. Das weitere Programmverhalten hängt dann von den eintreffenden Nachrichten und vom Verhalten der überlagerten Methoden ab.

2.1.3 Das Message-Map-Konzept

Windows-Anwendungen kommunizieren miteinander über den Austausch von *Nachrichten*. Ein SDK-Programm verfügt dazu über eine Reihe von Callback-Funktionen, die bei eintreffenden Nachrichten aufgerufen werden. Da jede Callback-Funktion viele verschiedene Nachrichten bearbeitet, tauchen in den Quellen lange, verschachtelte *switch*-Anweisungen auf.

Programme mit extensiven *switch*-Statements haben eine gewisse Ähnlichkeit mit Programmen, in denen Verzweigungen durch GOTOs realisiert werden müssen, und sind daher nicht ganz ungefährlich. Wird beispielsweise ein *break*-Statement vergessen oder an die falsche Stelle gesetzt, so wird die Anwendung instabil und fehlerhaft. Zudem sind die eingehenden Nachrichten stets untypisiert, d. h., die von Windows mitgeschickten Bestandteile *wParam* und *lParam* müssen vor der eigentlichen Verwendung erst einmal in die passenden Typen umgewandelt werden. Daß diese Fehlermöglichkeiten nicht nur theoretischer Natur sind, wird jeder SDK-Programmierer bestätigen: Die fehlerhafte Stelle ist in beiden Fällen meist nur mit großen Schwierigkeiten zu lokalisieren.

Hier geht Visual C++ einen anderen Weg. Anstelle der Nachrichtенbearbeitung in den *case*-Zweigen eines *switch*-Statements erfolgt diese in den Methoden der Klassen. Die Basisklasse eines Windows-Fensters besitzt für jede sinnvolle Nachricht eine Methode, die bei Eintreffen der Nachricht aufgerufen wird. Soll eine abgeleitete Fensterklasse auf eine bestimmte Nachricht anders reagieren, so überlagert sie einfach die zugehörige Methode der Basisklasse. Taucht dann beim Programmablauf die entsprechende Nachricht auf, wird sie an die Methode der abgeleiteten Klasse weitergeleitet. Die typischen *switch*-Anweisungen eines C-Programms gibt es nicht mehr – und ihre Fehler auch nicht.

Weiterhin werden die übergebenen Nachrichten vor Aufruf der Methode korrekt typisiert. Die formalen Parameter der Methode entsprechen dabei den Nachrichtenbestandteilen *wParam* und *lParam* und besitzen die zum Bearbeiten der Nachricht erforderlichen Typen. Daher gehören auch die typbezogenen Probleme der Vergangenheit an.

Im Prinzip könnten die Methoden für den Empfang von Nachrichten als virtuelle Funktionen mit den Standardmitteln eines C++-Compilers realisiert werden. Aus Gründen der Effizienz ging man bei der Entwicklung der Microsoft Foundation Classes jedoch einen anderen Weg und führte das Konzept der *Message-Maps* ein.

Eine abgeleitete Klasse bekommt für jede Nachricht, die sie bearbeiten will, zur Compile-Zeit einen Eintrag in ihre Message-Map. Dadurch wird eine Tabelle aufgebaut, mit deren Hilfe die Verbindung zwischen der Nachricht und der zugehörigen Methode hergestellt wird. Wird nun irgendeine Nachricht an das Fenster gesendet, so wird in der Message-Map der zugehörigen Klasse nach einer passenden Methode gesucht.

Falls eine passende Methode in dieser Klasse nicht definiert wurde, wird die Nachricht in einer bestimmten Reihenfolge an andere Klassen weitergeleitet. Diese Reihenfolge hängt einerseits davon ab, um welche Art von Nachricht es sich handelt, und andererseits, welches Objekt sie empfangen hat. Während die WM_COMMAND-Nachrichten, die durch Auswäh-

len von Menüpunkten oder Dialogboxelementen ausgelöst werden, mit Hilfe eines relativ komplizierten Mechanismus an andere Objekte weitergegeben werden, können alle übrigen Nachrichten entweder in der aktuellen Fensterklasse bearbeitet oder – wenn kein Message-Map-Eintrag zur Verfügung steht – an die Elternklasse weitergegeben werden.

Alle Standard-Windows-Messages verfügen wenigstens in der Klasse *CWnd* (der Basisklasse aller Fenster) über einen passenden Message-Map-Eintrag, so daß sie auch dann richtig behandelt werden, wenn in der abgeleiteten Klasse kein entsprechender Eintrag definiert wurde. Dieser Aufruf einer Methode der Basisklasse entspricht damit sinngemäß dem Aufruf der Funktion *DefWindowProc* bei der herkömmlichen Windows-Programmierung. Handler für WM_COMMAND-Nachrichten sind dagegen normalerweise in den vordefinierten Klassen nicht vorhanden; wenn hier kein passender Eintrag zu finden ist, wird das entsprechende Kommando einfach nicht ausgeführt.

Die Ableitung von Anwendungs- und Fensterklassen und die Kommunikation mit Member-Funktionen per Message-Map gehören zu den wichtigsten Programmierstechniken im Umgang mit den Microsoft Foundation Classes. Darüber hinaus werden später auch Klassen vorgestellt, die direkt (durch das Erzeugen von Instanzen) verwendet werden können.

Nach dieser kurzen Einführung soll nun damit begonnen werden, das »hello world«-Programm zu erläutern.

2.2 Die Klasse *CWinApp*

2.2.1 Ableiten einer Anwendungsklasse von *CWinApp*

Zunächst sollen die Hauptfunktion und die Nachrichtenschleife des Programmes erzeugt werden. Dazu wird aus der vordefinierten Klasse *CWinApp* eine neue Anwendungsklasse *CHelloApp* abgeleitet. *CHelloApp* übernimmt alle Eigenschaften der Basisklasse, mit Ausnahme der Methode *InitInstance*, die durch eine eigene Version überlagert wird. Damit sieht die Klassendeklaration wie folgt aus:

```
class CHelloApp : public CWinApp
{
public :
    BOOL InitInstance ();
};
```

Die Klassendeklaration wird in die Header-Datei *hello.h* geschrieben. Zusätzlich ist darin noch die Deklaration der Hauptfensterklasse enthalten, die weiter unten erklärt wird. Weil nicht grundsätzlich auszuschließen ist, daß Header-Dateien mehrfach eingebunden werden, sollte man die De-

klarationen in einer Header-Datei (Schnittstellendatei) davor schützen. Üblicherweise werden daher alle Deklarationen durch eine *#ifndef*-Präprozessor-Anweisung eingeklammert:

```
#ifndef _HELLO_H_
#define _HELLO_H_ //wenn noch nicht inkludiert, Symbol definieren
...
Hier stehen die Anweisungen
...
#endif
```

Die Frage ist nun, warum `CHelloApp` genau diese Form hat, d.h., warum gerade die Funktion *InitInstance* überlagert wird? Das hängt einerseits mit der Initialisierungstechnik von Windows-Anwendungen zusammen, andererseits damit, daß ein und dasselbe Programm mehrfach gestartet werden kann, daß es also mehrere *Instanzen* des Programms geben kann. Wird ein MFC-Programm gestartet, so laufen eine Reihe von Aktionen nacheinander ab:

1. Da im Hauptprogramm ein globales Objekt der Klasse `CHelloApp` angelegt wurde, wird dessen Konstruktor aufgerufen. Dieser ist mit dem Konstruktor von `CWinApp` identisch (da er nicht überladen wurde) und führt eine Folge von Variablen-Initialisierungen durch. Die wichtigste davon ist die Zuweisung des *this*-Zeigers an eine globale Variable, die dafür sorgt, daß die globale Funktion *AfxGetApp* (s.u.) korrekt arbeitet. Vereinfacht gesprochen stellt der Konstruktor die Verbindung zwischen der *WinMain*-Funktion der Foundation Classes und dem globalen Anwendungsobjekt her.
2. Nun initialisiert Windows sich selbst und ruft die MFC-Version von *WinMain* auf. Diese initialisiert zuerst das MFC-System und ruft dann mit Hilfe von *AfxGetApp* die folgenden Member-Funktionen von `CHelloApp` auf.
3. Falls die erste Instanz des Anwendungsprogramms gestartet wurde, wird die Methode *InitApplication* aufgerufen.
4. Danach wird *InitInstance* aufgerufen, um der aktuellen Instanz die Möglichkeit zur Initialisierung zu geben. Dies geschieht auch in Folgeinstanzen des Programms.
5. Nachdem die Initialisierungen durchgeführt wurden, wird die Methode *Run* aufgerufen. Diese enthält die Nachrichtenschleife, die so lange Nachrichten liest und bearbeitet, bis ein `WM_QUIT` ankommt.
6. Nach dem Ende der Nachrichtenschleife ruft *Run* die Methode *ExitApplication* auf, mit deren Hilfe die Anwendung die Endebehandlung durchführen kann.

Diese Initialisierungen befinden sich in den Dateien WINMAIN.CPP und APPCORE.CPP im Verzeichnis mit den MFC-Quelltexten.

InitInstance ist also genau die richtige Stelle, um Initialisierungen durchzuführen, die beim Start einer Instanz benötigt werden. Hierzu zählt neben anwendungsbezogenen Tätigkeiten (wie etwa dem Vorbelegen von Variablen) vor allem das Erzeugen und Anzeigen des Hauptfensters.

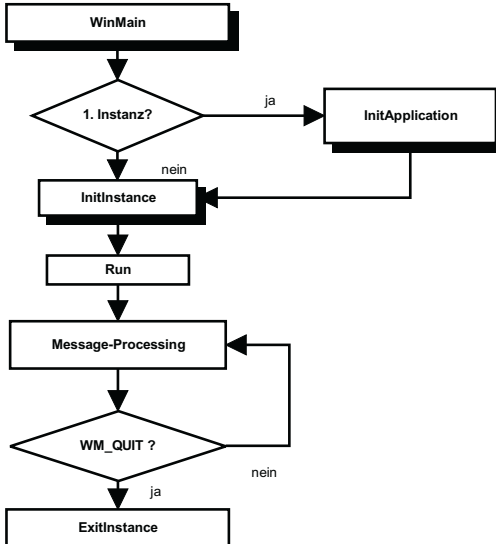


Abbildung 2.1: Ablauf eines Windows-Programms

In herkömmlichen SDK-Programmen wird vor dem Erzeugen des Hauptfensters in der ersten Instanz des Programms üblicherweise eine globale Fensterklasse registriert, aus der alle Instanzen ihr Hauptfenster erzeugen. Der richtige Platz dafür wäre die Methode *InitApplication*. Da die Foundation Classes die wichtigsten Fensterklassen bereits automatisch registrieren, entfällt dieser Schritt bei der MFC-Programmierung normalerweise. Auch das »hello world«-Programm überlagert *InitApplication* nicht. Falls diese Funktion überlagert werden soll, ist zumindest darin die Funktion der Basisklasse aufzurufen, da sonst die Dokument-Templates nicht initialisiert werden.

```

BOOL CHelloApp::InitInstance ()
{
    m_pMainWnd = new CMainWindow();
    m_pMainWnd->ShowWindow(m_nCmdShow);
    return TRUE;
}
  
```

Der boolesche Rückgabewert von *InitInstance* zeigt an, ob die Initialisierung erfolgreich (Rückgabewert TRUE) oder erfolglos (Rückgabewert FALSE) war. Wurde FALSE zurückgegeben, so wird das Programm gleich wieder beendet. Die übrigen Teile von *InitInstance* werden später noch ausführlich erläutert.

Der Platz für *InitInstance* ist die Datei HELLO.CPP. Wie bei allen Dateien, in denen Bestandteile der Microsoft Foundation Classes verwendet werden, muß die Header-Datei AFXWIN.H eingebunden werden. Diese enthält Deklarationen aller MFC-Klassen, Funktionsprototypen, Konstanten, Strukturen usw. und bindet ihrerseits Header-Dateien wie WINDOWS.H und andere ein. Durch AFXWIN.H wird der Quellcode eines MFC-Programms um mehrere Tausend Zeilen aufgebläht! Prinzipiell stört dies zwar nicht, es führt jedoch in der Praxis dazu, daß der C++-Compiler bei MFC-Programmen sehr langsam arbeitet. Wie man das Laufzeitverhalten durch Einsatz von *vorkompilierten Header-Dateien* verbessern kann, wird in einem späteren Abschnitt des Buchs erläutert.

Neben der Implementierung der Methoden befindet sich in HELLO.CPP auch die Definition des globalen Applikationsobjekts:

```
CHelloApp HelloApp;
```

Beim Starten des Programms wird dadurch die oben beschriebene Initialisierungskette in Gang gesetzt (Abbildung 2.1), an deren Ende schließlich der Sprung in die Nachrichtenschleife steht. Man sollte es auf jeden Fall vermeiden, versehentlich ein zweites Applikationsobjekt zu definieren. Dieser Fehler wird nur in der Debug-Version der Microsoft Foundation Classes durch ein ASSERT-Makro abgesichert, in der Release-Version bleibt er hingegen unbemerkt.

2.2.2 Überlagern der Methode *InitApplication*

Windows ist in der Lage, mehrere Instanzen eines Programms zu starten und parallel laufen zu lassen. Dabei können die einzelnen Instanzen auf bestimmte Ressourcen, wie beispielsweise Fensterklassen oder GDI-Objekte, gemeinsam zugreifen. Diese werden üblicherweise von der ersten Instanz erzeugt. Um eine Programminstanz darüber zu informieren, daß sie tatsächlich die erste ist, wird in C-Programmen der Parameter *hPrevInstance* an *WinMain* übergeben. Er ist genau dann NULL, wenn die aktuelle Instanz die erste ist.

Unter den Foundation Classes arbeitet dieser Erkennungsmechanismus etwas eleganter: Nur bei der ersten Instanz eines Programms wird die Methode *InitApplication* von *CWinApp* (bzw. der daraus abgeleiteten Applikationsklasse) aufgerufen. Wird hingegen die zweite, dritte oder eine weitere Instanz gestartet, erfolgt kein Aufruf von *InitApplication*, sondern es wird lediglich *InitInstance* aufgerufen.

```
class CWinApp:
    virtual BOOL InitApplication();
```

Hier sind also all die Tätigkeiten und Initialisierungen zu plazieren, die für sämtliche aufgerufenen Instanzen des Programms gültig sind. Um das praktisch zu realisieren, ist einfach die Methode *InitApplication* der von *CWinApp* abgeleiteten Applikationsklasse zu überlagern und entsprechend auszuprogrammieren. Da HELLO kein besonderes Handling der ersten Instanz benötigt, besitzt *CHelloApp* auch keine *InitApplication*-Methode.

2.2.3 Überlagern der Methode *ExitInstance*

Da MFC-Programme kein explizites Hauptprogramm besitzen, wird auch die Endebehandlung einer Programminstanz anders als in normalen C-Programmen erledigt. Zum Ausführen von Programmcode am Ende der Instanz gibt es in *CWinApp* die Methode *ExitInstance*. Diese wird unmittelbar vor dem Programmende (jedoch nach dem Ende der Nachrichtenschleife) aufgerufen, so daß sie der geeignete Platz für die Ausführung von anwendungsbezogenen Ende-Routinen ist. *ExitInstance* ist die letzte Aktion, bevor die Methode *Run* von *CWinApp* terminiert.

```
class CWinApp:
    virtual int ExitInstance();
```

ExitInstance ist auch dafür verantwortlich, einen definierten Rückgabewert an Windows zu liefern. Die Methode gibt einen Integer-Wert zurück, der von den Microsoft Foundation Classes als Rückgabewert für *WinMain* verwendet wird. Die Basisversion von *ExitInstance* gibt den beim Aufruf an *PostQuitMessage* übergebenen Wert zurück.

2.2.4 Globale Funktionen

Bei der Windows-Programmierung kommt es vor, daß man in einem bestimmten Teil des Programms plötzlich Daten benötigt, die eigentlich nur im Hauptprogramm zur Verfügung stehen. So benötigt beispielsweise die Funktion *MakeProcInstance* den Instanz-Handle der Anwendung und *GetDC* den Handle des Hauptfensters. Um diese Werte zur Verfügung zu stellen, definiert man bei der SDK-Programmierung üblicherweise globale Variablen, die in *WinMain* initialisiert werden.

```
CWinApp *::AfxGetApp ();
LPCTSTR ::AfxGetAppName ();
HINSTANCE ::AfxGetInstanceHandle ();
HINSTANCE ::AfxGetResourceHandle ();
CWnd *::AfxGetMainWnd ();
CWinThread *::AfxGetThread ();
```

Natürlich kann man dies auch bei der MFC-Programmierung machen, es ist jedoch nicht unbedingt erforderlich, denn die Foundation Classes machen dies schon selbst. Dazu wurde eine Reihe von globalen Funktionen implementiert, von denen die gebräuchlichsten kurz erklärt werden sollen:

- ▼ *AfxGetApp*: *AfxGetApp* liefert einen Zeiger auf das globale Applikationsobjekt. Mit Hilfe dieses Zeigers kann beispielsweise das Hauptfenster ermittelt werden, da der Member *m_pMainWnd* darauf zeigt:

```
AfxGetApp ()->m_pMainWnd
```

So kann man an jeder beliebigen Stelle des Programms ohne Verwendung einer globalen Variablen einen Zeiger auf das Hauptfenster-Objekt erhalten.

- ▼ *AfxGetAppName*: Diese Funktion liefert einen Zeiger auf einen nullterminierten String, der den logischen Namen der Applikation enthält. Für das HELLO-Programm wird beispielsweise der String »HELLO« zurückgegeben.
- ▼ *AfxGetInstanceHandle*: Diese Funktion liefert einen Handle auf die aktuelle Instanz des Windows-Programms. Dieser kann z.B. verwendet werden, um Funktionen aufzurufen, die einen formalen Parameter *hInst* besitzen (etwa *MakeProInstance*).
- ▼ *AfxGetResourceHandle*: Diese Funktion liefert einen Handle auf die Instanz des Programms, aus der die Ressourcen geladen werden. Dieser Handle kann z.B. verwendet werden, um auf die Ressourcen des Programms direkt zuzugreifen, beispielsweise mit der Funktion *FindResource*.



Neben den hier beschriebenen Funktionen gibt es noch einige andere mit teilweise sehr interessanten und speziellen Fähigkeiten. Eine genaue Beschreibung dieser Funktionen ist in der Dokumentation zu finden (siehe *global variables*, MFC).

- ▼ *AfxGetMainWnd*: Der Aufruf der Funktion liefert einen Zeiger auf das Hauptfenster zurück und entspricht dem Aufruf von *AfxGetApp* ()->*m_pMainWnd*. Wenn die Applikation ein OLE-Server ist, wird ein Zeiger auf das aktive Hauptfenster zurückgegeben.
- ▼ *AfxGetThread*: Die Funktion wird innerhalb eines Threads aufgerufen, um einen Zeiger auf das aktuelle *CWinThread*-Objekt zu erhalten.

2.2.5 Das Programm-Icon

Die Ressource-Datei

Damit ein Programmfenster nach dem Minimieren nicht mehr für das Neuzeichnen seines (nunmehr sehr kleinen) Bildschirmausschnitts verantwortlich ist, besitzt es in der Regel ein Symbol (Icon), das statt dessen in der Task-Leiste angezeigt wird. Um festzulegen, welches Icon angezeigt werden soll, wird in der Ressource-Datei des Programms eine Icon-Datei unter einem bestimmten Namen eingebunden und bei der Definition der Hauptfensterklasse über diesen Namen aufgerufen.

Bei den Foundation Classes entfällt der zweite Schritt, wenn das Icon den Bezeichner `AFX_IDI_STD_FRAME` hat. Dies ist nämlich der vordefinierte Name des Icons, das in der Fensterklasse einer SDI-Applikation (Single Document Interface: Anwendungen, die nur ein Dokument im Hauptfenster öffnen können) verwendet wird. Wie später noch deutlich werden wird, ist es bei der Programmierung unter Visual C++ in der Regel unnötig, eigene Fensterklassen (im Windows-Sinne) zu registrieren, da alle wesentlichen Fensterklassen bereits von der MFC registriert werden. Aus diesem Grund sind für den MFC-Programmierer auch die Namen der Icons bereits fest vorgegeben – `AFX_IDI_STD_FRAME` für SDI-Anwendungen und `AFX_IDI_STD_MDIFRAME` für MDI-Anwendungen.

Das manuelle Editieren der Ressource-Datei ist nicht notwendig, da das Erstellen und Verändern von Ressourcen in das Developer Studio integriert wurde. Je nach gewähltem Ressource-Typ werden die entsprechenden Editoren aktiviert.

Anlegen einer neuen Icon-Ressource

Um das Programm-Icon einbinden zu können, muß eine Ressource-Datei erzeugt werden, die es enthält. Dazu muß man wie folgt vorgehen:

1. Zuerst wird im Menü EINFÜGEN der Punkt RESSOURCE aufgerufen. Alternativ dazu kann `[Strg][R]` gedrückt werden.
2. In der angezeigten Dialogbox muß der Typ »Icon« ausgewählt und anschließend mit NEU bestätigt werden.
3. Danach wird im Editor-Fenster ein leeres Icon angezeigt, das jetzt mit Hilfe der vorhandenen Zeichentools gestaltet werden kann. Zu dieser Stelle im Ablauf kann man sofort kommen, wenn in der Ressource-Symbolleiste der Button NEUES SYMBOL (NEW ICON) bzw. `[Strg][4]` gedrückt wird. Es stehen die üblichen Funktionen und Werkzeuge Windows-basierter Grafikeditoren zur Verfügung, beispielsweise Pinsel, Linien oder Kreise; es kann in verschiedenen Farben und Strichstärken gezeichnet werden, und es gibt Werkzeuge für Spezialeffekte

wie beispielsweise eine Sprühpistole oder ein Flächenfüller. Der Schlüssel zu den Werkzeugen sind die Symbolleisten *Grafiken* und *Farben*, das eigentliche Zeichnen erfolgt mit der Maus.

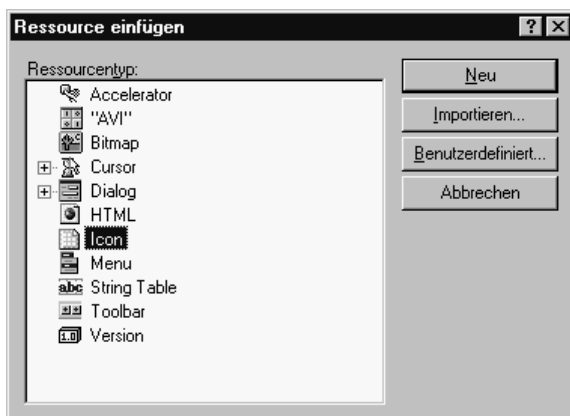


Abbildung 2.2: Anlegen einer neuen Ressource

Nachdem das Editieren abgeschlossen ist, müssen die Eigenschaften des Icons festgelegt werden. Dazu ist mit **[Alt] [←]** oder über den Menüpunkt **ANSICHT|EIGENSCHAFTEN** die Eigenschafts-Dialogbox aufzurufen und der Name des Icons mit **AFX_IDI_STD_FRAME** anzugeben. Über den Push-Pin wird erreicht, daß das Eigenschafts-Fenster immer sichtbar bleibt. Im Feld *Dateiname* kann der Name der Symbol-Datei festgelegt werden.

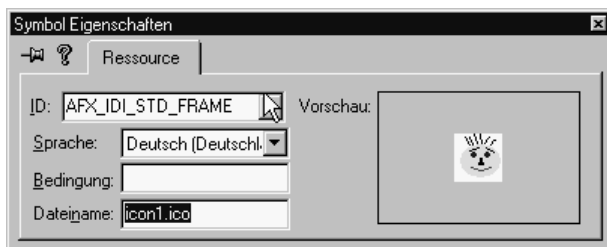


Abbildung 2.3: Festlegen der Eigenschaften des Icons

4. Über **DATEI|SPEICHERN** oder **DATEI|SPEICHERN UNTER** bzw. den Button der Standard-Symbolleiste wird die Ressource-Datei gesichert. Die Datei erhält die Erweiterung **.RC**, was einer Ressource entspricht.

5. Nun müssen die neue RC-Datei sowie die Icon-Ressource geschlossen werden. Im Anschluß daran erfolgt das Einfügen der Ressource-Datei in das Projekt. Dazu wird im Menü PROJEKT der Punkt DEM PROJEKT HINZUFÜGEN|DATEIEN aufgerufen. Als Dateityp wird *Ressource-Dateien* gewählt, und danach wird die zuvor gesicherte Datei ausgewählt. Anschließend steht im *Arbeitsbereich*-Fenster die *Ressourcen*-Ansicht zur Verfügung, mit der alle vorhandenen Ressourcen hierarchisch dargestellt werden. Ein Doppelklick auf den Bezeichner einer Ressource bringt diese in das *Editor*-Fenster.

Für das Anlegen weiterer Ressourcen ist es nicht erforderlich, neue Resource-Dateien anzulegen. Alle Ressourcen können in eine Datei geschrieben werden. Die Darstellung der Ressourcen in der *Ressourcen*-Ansicht des Arbeitsbereiches erlaubt einen schnellen Zugriff auf alle Ressourcen. Auch hier kann durch Drücken der rechten Maustaste ein Kontextmenü geöffnet werden, um beispielsweise den *Eigenschafts*-Dialog zu öffnen. Im Kontextmenü der Resource-Datei können über den Punkt RESSOURCENSYMBOLS alle definierten Symbole angezeigt werden. Die Checkbox »Schreibgeschützte Symbole anzeigen« zeigt alle Symbole an, die über andere Ressourcendateien eingebunden wurden. In erster Linie sind dies die Ressourcen der MFC, die in der Datei AFXRES.H enthalten sind. Darin sind auch die vordefinierten Symbole für die Icons AFX_IDI_STD_FRAME und AFX_IDI_STD_MDIFRAME enthalten.

Nachladen von Symbolen

Neben der Festlegung in der Hauptfensterklasse kann ein Icon auch manuell nachgeladen werden. Die Klasse *CWinApp* verfügt dazu über die Methoden *LoadIcon*, *LoadStandardIcon* und *LoadOEMIcon* zum Nachladen von Symbolen aus der Programmdatei.

Diese Methoden entsprechen den gleichnamigen SDK-Funktionen. Sie lesen jeweils ein Icon aus der Programmdatei und geben einen Handle darauf zurück. Dieser kann dann zum Bearbeiten von Icons verwendet werden, beispielsweise mit einer der Funktionen *DrawIcon* oder *DestroyIcon*:

```
class CWinApp :
    HICON LoadIcon( LPCTSTR lpszResourceName ) const;
    HICON LoadIcon( UINT nIDResource ) const;
    HICON LoadStandardIcon( LPCTSTR lpszIconName ) const;
    HICON LoadOEMIcon( UINT nIDIcon ) const;
```

2.2.6 Der Programmcursor

Nach dem Starten verwendet eine MFC-Anwendung automatisch den in Windows vordefinierten Cursor IDC_ARROW. Dieser wird beim Initialisieren durch Aufruf von *::LoadCursor* in der Hauptfensterklasse registriert,

ohne daß dabei zusätzliches Programmieren erforderlich wäre. Darüber hinaus kann ein alternativer Cursor mit einer der Methoden *LoadCursor*, *LoadStandardCursor* oder *LoadOEMCursor* von *CWinApp* geladen werden.

```
class CWinApp :
    HCURSOR LoadCursor( LPCTSTR lpszResourceName ) const;
    HCURSOR LoadCursor( UINT nIDResource ) const;
    HCURSOR LoadStandardCursor( LPCTSTR lpszCursorName ) const;
    HCURSOR LoadOEMCursor( UINT nIDCursor ) const;
```

Jede der Methoden gibt einen Handle auf den ausgewählten Cursor zurück, der dann in Funktionen, die einen solchen Handle verlangen (z.B. *SetCursor*), verwendet werden kann. Leider ist die Verwendung von *SetCursor* zum Aktivieren eines speziellen Cursors bei der Verwendung der Microsoft Foundation Classes etwas schwierig. Der Aufruf der Funktion schaltet zwar zunächst auf den gewünschten Cursor um, bei der nächsten Mausbewegung ist aber wieder der Pfeil da. Das liegt daran, daß in der Hauptfensterklasse der Microsoft Foundation Classes nicht NULL, sondern IDC_ARROW als Cursor-Handle registriert wird. In diesem Fall reagiert jede Mausbewegung den Pfeilcursor. Eine mögliche Lösung dieses Problems besteht darin, *SetCursor* in der Methode *OnSetCursor* aufzurufen.



```
BOOL CMainWindow::OnSetCursor(CWnd* pWnd, UINT nHitTest, UINT
message )
{
    SetCursor(hCursor1);
    return TRUE;
}
```

OnSetCursor wird bei jeder Mausbewegung aufgerufen und setzt darin den Cursor neu. Anschließend wird TRUE zurückgegeben. Diese Variante funktioniert zwar, der elegantere Weg führt aber über die Registrierung eigener Cursor-Handles. Diese Funktion ist eine der Erweiterungen von HELLO. Die nächste Erweiterung ist das Erzeugen eines anderen Cursors, wenn die linke Maustaste gedrückt ist. Damit ist das Drücken der Maustaste unmittelbar am Bildschirm verfolgbar.

```
void CMainWindow::OnLButtonDown(UINT nFlag, CPoint point)
{
    SetCursor(hCursor2);
    InvalidateRect(NULL, FALSE);
    UpdateWindow();
}
```

Windows sendet die Botschaft `WM_LBUTTONDOWN`, wenn die linke Maustaste gedrückt wird. Diese gilt es auszuwerten und die Funktion *OnLButtonDown* mit entsprechendem Code zu füllen. Darin wird wieder nur ein eigener Cursor aktiviert und anschließend das Fenster aktualisiert. Auch hier gilt, daß bei der kleinsten Mausbewegung der Cursor zurückgesetzt wird.

2.2.7 Daten-Member in *CWinApp*

Die Klasse *CWinApp* enthält einige nützliche Daten-Member.

```
class CWinApp :
    const char* m_pszAppName;
    HINSTANCE m_hInstance;
    HINSTANCE m_hPrevInstance;
    LPTSTR m_lpCmdLine;
    int m_nCmdShow;
    BOOL m_bHelpMode;
    CWnd* m_pActiveWnd;
    const char* m_pszExeName;
    const char* m_pszHelpFilePath;
    const char* m_pszProfileName;
    LPCTSTR m_pszRegistryKey;
class CWinThread:
    CWnd* m_pMainWnd;
```

Die meisten dieser Member entsprechen den gleichnamigen Parametern von *WinMain* und werden bei der Initialisierung einer MFC-Anwendung automatisch gesetzt. Sie haben dieselbe Bedeutung wie bei der normalen SDK-Programmierung und werden deshalb nicht weiter erklärt. Der Member *m_pszAppName* enthält einen Zeiger mit dem Namen der Applikation, der an den Konstruktor von *CWinApp* übergeben wurde.

Eine besondere Stellung nimmt *m_pMainWnd* ein. Hierbei handelt es sich um einen Zeiger auf das Hauptfenster der Applikation. Er wird von den Foundation Classes vor allem dazu gebraucht, das Handling der Nachrichten ordnungsgemäß durchzuführen. Im Unterschied zu den anderen Daten-Membern muß er von der Anwendung selbst gesetzt werden, nachdem sie das Hauptfenster erzeugt hat (s. *InitInstance*). Da *m_pMainWnd* eine *public*-Variable der Klasse *CWnd* ist, wird bei der Beschreibung dieser Klasse noch näher darauf eingegangen.

2.2.8 Auswerten der Kommandozeile

Auch Windows-Programme können Kommandozeilen-Parameter besitzen, selbst wenn die Angabe der Parameter nicht immer einfach ist. So ist es zum Beispiel möglich, Parameter an den Programmnamen anzuhän-

gen, indem der Menüpunkt EIGENSCHAFTEN im Kontextmenü einer Verknüpfung aufgerufen wird, und die Parameter hinter den Programmnamen geschrieben werden. Da Windows-Programme auch aus einer DOS-Box aufgerufen werden können (oder aus Tools wie dem Norton Commander[®]), können dabei ebenfalls Kommandozeilen-Parameter übergeben werden.

Um auf diese Parameter innerhalb eines MFC-Programms zugreifen zu können, besitzt die Klasse *CWinApp* den Daten-Member *m_lpCmdLine*, der durch die *WinMain*-Routine der Foundation Classes gesetzt wird. Hierbei handelt es sich um einen (FAR-) Zeiger auf eine Zeichenkette, welche die komplette Kommandozeile genauso enthält, wie sie vom Benutzer eingegeben wurde. Um die Kommandozeile auszuwerten, ist es nötig, die Zeichenkette von links nach rechts zu durchlaufen, und dabei Sonderzeichen wie Whitespaces, Anführungszeichen oder Slashes geeignet abzuarbeiten.

```

BOOL CMyApp::InitInstance ()
{
    // ...
    if (m_lpCmdLine[0] == '\\0')
    {
        // Create a new (empty) document.
        OnFileNew();
    }
    else
    {
        // Open a file passed as the first command line parameter.
        OpenDocumentFile(m_lpCmdLine);
    } // ...
}

```

Listing: Auswerten der Kommandozeile in *InitInstance*

Verglichen mit der in C üblichen Übergabe der Kommandozeile in einem Zeigerarray, bei der die einzelnen Elemente jeweils als nullterminierte Zeichenkette zur Verfügung stehen, ist der unter Windows verwendete Mechanismus für den Programmierer etwas umständlich zu handhaben. Es ist fast schon einfacher, eine Dialogbox zu erstellen, als die Auswertung einer komplizierten Kommandozeile zu programmieren.

Natürlich haben die Entwickler diesen Punkt nicht etwa vergessen oder ihm aus Bequemlichkeit zu wenig Beachtung geschenkt. Vielmehr gibt es unter Windows ja gar keine Kommandozeile im eigentlichen (DOS-)Sinne mehr. Statt dessen wird jedes nennenswerte Programm als Icon abgelegt und bei Mausklick aufgerufen. Die Voreinstellung bestimmter Optionen, die einst per Parameter und Schalter in der Kommandozeile erfolgte, wird nun im interaktiven Dialog mit dem Benutzer erledigt. Daß man im Pro-

gramm-Manager dennoch eine Kommandozeile vorgesehen hat, sollte eher als Notnagel betrachtet werden, etwa um ausnahmsweise einmal einen Dateinamen zu übergeben oder eine besondere Betriebsart zu aktivieren.

In Visual C++ 6.0 existiert noch eine weitere Möglichkeit, die Kommandozeile auszuwerten: Die Klasse *CCommandLineInfo* dient diesem Zweck. Das funktioniert aber nur, wenn die Parameter in einem fest vorgegebenen Format angegeben werden. In Tabelle 2.2 sind die möglichen Formen des Programmaufrufes aufgelistet. Weitere Informationen dazu sind in der Online-Dokumentation unter dem Stichwort *command line parsing* zu finden.

Kommandozeile	Bedeutung
hello	Aufruf der Applikation HELLO.EXE (erzeugen eines neuen Dokuments)
hello hello.txt	Aufruf der Applikation und öffnen von HELLO.TXT
hello /p hello.txt	Aufruf der Applikation und drucken von HELLO.TXT auf dem Standard-Drucker
hello /pt hello.txt drucker treiber port	Aufruf der Applikation und drucken von HELLO.TXT auf dem angegebenen Drucker

Tabelle 2.2: Kommandozeilen-Auswertung über CCommandLineInfo

Die Auswertung der Kommandozeile erfolgt ebenfalls in der Funktion *InitInstance*. Diese könnte folgendermaßen aussehen:

```

BOOL CHelloApp::InitInstance ()
{
    CCommandLineInfo cmdInfo;
    ParseCommandLine(cmdInfo);
    if (cmdInfo.m_strFileName == '\0')
        OnFileNew(); //neues Dokument anlegen
    else
        OpenDocumentFile(cmdInfo.m_strFileName);
        //Dokument öffnen
    // ...
}

```



Listing: Auswertung der Kommandozeile über CCommandLineInfo

Zusammenfassend kann man sagen, daß bei der manuellen Auswertung von *m_lpCmdLine* auf folgende Punkte geachtet werden muß:

- ▼ Zwischen zwei Parametern können mehrere Leerzeichen auftauchen, diese werden von Windows nicht automatisch entfernt.
- ▼ Schalter, die durch einen Schrägstrich (/) oder ein Minus (-) eingeleitet werden, gehören zu dem vorigen Parameter, wenn sie nicht durch ein Leerzeichen getrennt werden.

- ▼ Anführungszeichen, die dazu dienen, mehrere Worte zu einem Parameter zu gruppieren, müssen manuell interpretiert werden.
- ▼ Platzhalter wie ? und * müssen manuell interpretiert werden.

2.2.9 Multitasking in eigenen Programmen

In einer Multitasking-Umgebung können mehrere Prozesse gleichzeitig ablaufen, so auch in Windows 95 und Windows NT. Während Windows 95 nur zum Teil ein 32-Bit-System ist, wurde Windows NT vollständig als 32-Bit-Betriebssystem entwickelt. Da Windows NT alte 16-Bit-Programme in einer Emulation abarbeitet, können diese unter Umständen langsamer laufen als in Windows 3.x. Windows 3.x als Multitasking-Betriebssystem zu bezeichnen, wäre übertrieben; dennoch gab es auch da schon gewisse Möglichkeiten, mehrere Programme parallel laufen zu lassen.

Windows 95 arbeitet Prozesse nach zwei verschiedenen Methoden ab – dem kooperativen und dem preemptiven Multitasking. Beim kooperativen Multitasking, das bei 16-Bit-Software zur Anwendung kommt, muß die Applikation regelmäßig die Nachrichten-Schlange abfragen, um nicht den Eindruck zu bekommen, daß ein Programm hängt. Jedes 32-Bit-Programm besitzt eine eigene Message-Queue und arbeitet in einem separaten Speicherbereich, so daß hier das preemptive Multitasking verwendet wird, bei dem das Betriebssystem die Verteilung der Prozessorzeit übernimmt.

DOS-Programme arbeiten (bei einem 386er oder höher) im unterbrechenden Multitasking, d. h., sie bekommen vom Scheduler (ein Systemprozeß) in bestimmten Zeitabständen Rechenzeit zugeteilt und werden nach Ablauf dieser Rechenzeit wieder unterbrochen. Da sie einen virtuellen 8086er mit eigenem MSDOS zugeteilt bekommen, merken sie in der Regel noch nicht einmal, daß andere Anwendungen parallel zu ihnen ablaufen.

Bei den 16-Bit-Windows-Programmen gestaltet sich die Sache komplizierter, denn sie müssen sich selbst um die Verwaltung ihrer Rechenzeit kümmern. Genauer gesagt: Bekommt ein 16-Bit-Windows-Task den Prozessor zugeteilt, muß er ihn freiwillig nach einer gewissen Zeit wieder freigeben, damit er einer anderen Task zugeteilt werden kann. Eine aktive 16-Bit-Windows-Task wird nie zugunsten einer anderen Windows-Task unterbrochen, sondern muß dies von sich aus tun. Man bezeichnet dies auch als nicht-unterbrechendes Multitasking.

32-Bit-Windows-Anwendungen erhalten von einem Scheduler Zeitscheiben des Prozessors zugewiesen. Dabei spielt die Priorität bei der Vergabe eine erhebliche Rolle. In regelmäßigen Abständen wird sie neu berechnet und entsprechend dem Ergebnis die Prozessorzeit verteilt, wenn nicht ge-

rade eine MSDOS-Anwendung 100% der Prozessorzeit benötigt. Da sowohl DOS- als auch 16- und 32-Bit-Windows-Programme zur gleichen Zeit laufen können, müssen alle Varianten koordiniert werden.

Die *OnIdle*-Funktion

Um Hintergrundaktivitäten in eigenen MFC-Programmen verwenden zu können, besitzt die Klasse *CWinApp* die Methode *OnIdle*, um rechenzeitintensive Arbeiten auszuführen. *OnIdle* wird innerhalb der Nachrichtenschleife immer dann aufgerufen, wenn die Applikations-Warteschlange leer ist, also gerade keine Nachrichten bearbeitet werden müssen. Innerhalb von *OnIdle* können dann zeitaufwendige Berechnungen in kleinen Häppchen durchgeführt werden (siehe Abbildung 2.4). Diese Methode ist für 16-Bit-Anwendungen zwingend, da sie nicht mehrere Programmfäden (Threads) starten können.

Es ist allerdings wichtig, daß nicht zu viel Rechenzeit auf einmal in Anspruch genommen wird, denn alle anderen Windows-Tasks bleiben inzwischen stehen. Statt dessen sollte die Aufgabe in kleine Teile zerlegt werden, die in aufeinanderfolgenden Aufrufen von *OnIdle* ausgeführt werden. Um hierbei den Foundation Classes nach einem Rücksprung aus *OnIdle* mitzuteilen, daß noch mehr Rechenzeit benötigt wird, besitzt die Funktion einen booleschen Rückgabewert. Ist dieser *TRUE*, so wird erneut *OnIdle* aufgerufen, ist er *FALSE*, wird die Kontrolle an eine andere Task übertragen.

```
class CWinApp:  
    virtual BOOL OnIdle ( long lCount);
```

Die Größe der Einzelaufgaben sollte so gewählt werden, daß andere Anwendungen nicht zu träge reagieren. Als Anhaltspunkt kann dabei eine Rechenzeit von maximal etwa 1/2 bis 1 Sekunde Dauer angesetzt werden. Werden insbesondere interaktive Anwendungen länger unterbrochen, wird das Arbeiten mit ihnen zur Qual. Darüber hinaus ist während der Dauer eines *OnIdle*-Aufrufs natürlich auch die eigene Anwendung unterbrochen. System- oder Benutzernachrichten werden erst wieder bearbeitet, wenn *OnIdle* beendet ist.

Eine Besonderheit von *OnIdle* ist der Parameter *lCount*, der bei jedem Aufruf übergeben wird. Hierin wird die Anzahl der Aufrufe von *OnIdle* seit dem Bearbeiten der letzten Nachricht mitgezählt, so daß die Anwendung in etwa abschätzen kann, wie lange sie ohne Nachrichten im Wartezustand verbracht hat. So kann sie beispielsweise mehrere parallel auszuführende Tätigkeiten priorisieren, indem die wichtigen bei niedrigen Werten von *lCount* und die unwichtigen bei höheren Werten ausgeführt werden.

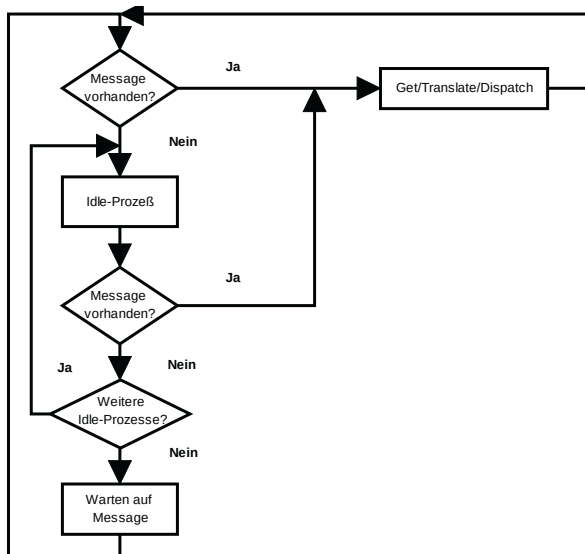


Abbildung 2.4: Auswertung von Messages

Ebenfalls wichtig ist es, in einer eigenen *OnIdle*-Funktion die gleichnamige Methode der Basisklasse *CWinApp* aufzurufen, um das Standard-Idle-Time-Processing nicht abzuhängen. Hier führt die Basisklasse Aufräumarbeiten in den internen Datenstrukturen durch und aktualisiert Userinterface-Objekte. Der Aufruf kann am Anfang oder am Ende von *OnIdle* erfolgen.

Diese theoretischen Ausführungen sollen nun durch ein kleines Beispiel abgerundet werden. Das »hello world«-Programm wird so umgestaltet, daß es anstatt »hello world« die Quadratwurzel von 71 anzeigt. Damit genug zu tun ist, wird diese mit einem langsamen iterativen Verfahren im Hintergrund berechnet. Während der Dauer der Berechnung zeigt der Bildschirm die Nachricht »Ich rechne...«.

Um die Methode *OnIdle* aufzunehmen, ist die Klassendeklaration von *CWinApp* in *HELLO.H* zu verändern:

```
class CHelloApp : public CWinApp
{
public :
    BOOL InitInstance ();
    BOOL OnIdle(long lCount);
};
```

In *HELLO.CPP* wird zunächst die statische Variable *sqrt71* angelegt, damit *OnIdle* das Ergebnis der Berechnung an die Methode *OnPaint* überreichen kann.

```
static double    sqrt71=0.0;
```

OnIdle ist so aufgebaut, daß die iterative Berechnung der Quadratwurzel bei jedem Aufruf um 1000 Rechenschritte vorangebracht wird. Falls das Ergebnis danach noch nicht ermittelt ist, terminiert die Funktion und gibt TRUE zurück, um zu signalisieren, daß noch mehr Rechenzeit benötigt wird. Ist das Ergebnis ermittelt, wird das Hauptfenster als ungültig markiert, neu gezeichnet und schließlich mit dem Rückgabewert FALSE das Ende der Berechnung signalisiert. Der aktuelle Zustand der Berechnung wird jeweils in der lokalen statischen Variablen *val* festgehalten, die zwischen den Aufrufen von *OnIdle* ihren Wert behält.

```
BOOL CHelloApp::OnIdle(long lCount)
{
    static double val=0.0;
    CWinApp::OnIdle(lCount);
    for (int i=0; i<1000; ++i) {
        if (val*val>=71.0) {
            sqrt71 = val;
            m_pMainWnd->InvalidateRect(NULL,TRUE);
            m_pMainWnd->UpdateWindow();
            return FALSE;
        }
        val += 0.000001;
    }
    return TRUE;
}
```

Listing: Schrittweise Berechnung in OnIdle

Nun muß noch die *OnPaint*-Methode des Hauptfensters etwas umgeschrieben werden, um den neuen Anforderungen zu genügen. Sie erkennt am Inhalt der Variablen *sqrt71*, ob die Berechnung noch läuft oder ob das Ergebnis bereits vorliegt.

```
void CMainWindow::OnPaint()
{
    CPaintDC dc(this);
    CRect rect;
    char buf[80];
    if (sqrt71==0.0) {
        sprintf(buf,"Ich rechne...");
    } else {
        sprintf(buf,"sqrt(71)=%lf",sqrt71);
    }
}
```

```
GetClientRect(rect);  
dc.SetTextAlign(TA_BASELINE | TA_CENTER);  
dc.TextOut(rect.right/2, rect.bottom/2, buf, strlen(buf));  
}
```

Listing: Ausgabe der Ergebnisse in OnPaint

Es lohnt sich, mit diesem Beispiel etwas zu experimentieren, um ein Gefühl für die Multitasking-Fähigkeiten von Windows zu bekommen. So kann man sehr leicht feststellen, daß die Bedienung anderer Programme unangenehm wird, wenn man die Rechenzeit je Aufruf von *OnIdle* zu sehr hochschraubt. Darüber hinaus kann man beispielsweise nachprüfen, ob einer Anwendung auch dann Rechenzeit zugeteilt wird, wenn sie als Icon dargestellt wird oder gar nicht sichtbar ist. Läuft hingegen eine DOS-Box mit gesetztem Exklusiv-Flag im Vordergrund, so werden nicht nur die anderen DOS-Boxen, sondern auch alle aktiven Windows-Anwendungen angehalten.

Eine weitere Methode, Hintergrundaktivitäten in einem Programm auszuführen, sind sogenannte Threads oder »Programmfäden«.

Prozesse und Threads

Betriebssysteme wie Windows NT oder Windows 95 unterstützen *Multitasking* und *Multi-Threading*. Multitasking heißt, daß mehrere Applikationen gleichzeitig laufen können, wobei dies auch mehrere Instanzen einer Applikation sein können. Dabei stellt eine Applikation im Sinne des Betriebssystems einen *Prozeß* dar. Diese Funktionalität ist allen MFC-Programmen gegeben. Ab dem Start der zweiten Instanz eines Programms wird nicht mehr *InitApplication*, sondern nur noch *InitInstance* aufgerufen (siehe Kapitel 2.2.1).

Ein Thread ist ein selbständig ablaufender Programmfaden innerhalb eines Prozesses. Jede 32-Bit-Applikation, die gestartet wird, erzeugt mit dem Prozeß den *primären Thread*. Das Ende dieses Threads ist gleichbedeutend mit dem Ende des Prozesses. Um bestimmte zeitaufwendige Aktivitäten eines Programms in den Hintergrund zu verlagern, können weitere Threads erzeugt werden. Der Nutzer braucht also nicht zu warten, bis zum Beispiel eine Grafik geladen ist, sondern kann schon weiterarbeiten. Ein bekanntes Beispiel für einen Thread ist die Rechtschreibkontrolle während der Eingabe in Winword.

Ein Thread wird durch das Objekt *CWinThread* repräsentiert. Die MFC erlaubt es, durch Aufruf der Funktion *AfxBeginThread* einen neuen Thread zu erzeugen. Dieser wird durch Aufruf von *AfxEndThread* innerhalb des Threads beendet. Die MFC unterscheidet *User-Interface-Threads* und *Worker-Threads*. Während Worker-Threads einfach bestimmte Tätigkeiten im Hintergrund ausführen, dienen User-Interface-Threads der Kommunikation mit dem Nutzer.

Nähere Informationen dazu sind unter dem Stichwort *Multithread-Programmierung mit MFC* in der Online-Hilfe zu finden.



```
CWinThread* AfxBeginThread (
    AFX_THREADPROC pfnThreadProc,
    LPVOID pParam, int nPriority = THREAD_PRIORITY_NORMAL,
    UINT nStackSize = 0,
    DWORD dwCreateFlags = 0,
    LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL );

CWinThread* AfxBeginThread(
    CRuntimeClass* pThreadClass,
    int nPriority = THREAD_PRIORITY_NORMAL,
    UINT nStackSize = 0,
    DWORD dwCreateFlags = 0,
    LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL );
```

Die erste Form des Aufrufs erzeugt einen *Worker-Thread* und dient dem Ausführen von Arbeiten im Hintergrund. Die zweite Form entspricht einem *Benutzeroberflächen-Thread* (User-Interface-Thread). Damit wird die Interaktion mit dem Benutzer ermöglicht, unabhängig davon, wie die anderen Programmfäden gerade ausgelastet sind. Im folgenden Abschnitt wird die Berechnung der Wurzel in einen eigenen Worker-Thread verlegt. Dazu muß die Behandlung von *OnIdle* aus der Nachrichtentabelle wieder entfernt werden. Die Berechnungsfunktion als Thread sieht folgendermaßen aus:

```
UINT SqrtThread(LPVOID pParam)
{
    double val=0.0;
    double dTemp = sqrt(71);
    while (val < dTemp)
    {
        val += 0.000001;
    }
    sqrt71 = val;
    bBerechne = FALSE;
    ::PostMessage((HWND)pParam,WM_ENDTHREAD,0,0);
    return 0;
}
```

Listing: Die Wurzelberechnung als Thread

In der Methode *OnLButtonDown* wird die Berechnung gestartet, also mit dem Drücken der linken Maustaste im Client-Bereich der Anwendung. Um zu verhindern, daß durch mehrfaches Drücken der Maustaste die Berechnung mehrfach gestartet wird, wird in einer booleschen Variablen angezeigt, daß die Berechnung läuft.

```

void CHelloApp::OnLButtonDown(long lCount)
{
    HWND xWnd=GetSafeHwnd();
    TRACE("LButtonDown \n");
    SetCursor(hCursor2);
    if (bBerechne) return;
    bBerechne = TRUE;
    sqrt71=0.0;
    CWinThread* pThread = AfxBeginThread(SqrtThread, xWnd,
    THREAD_PRIORITY_NORMAL);
    InvalidateRect(NULL,FALSE);
    UpdateWindow();
}

```

Listing: Starten des Threads mit der Linken Maustaste

Die Variable `bBerechne` dient gleichzeitig der Ausgabe in *OnPaint*. Wenn die globale Variable `sqrt71` Null ist und `bBerechne` wahr, dann läuft gerade der Thread, ansonsten wird der berechnete Wert ausgegeben.

In Abbildung 2.5 ist der Prozeß-Betrachter mit der Untersuchung der Anwendung *HELLO* zu sehen. Im Fenster sind deutlich zwei aktive Threads zu sehen.

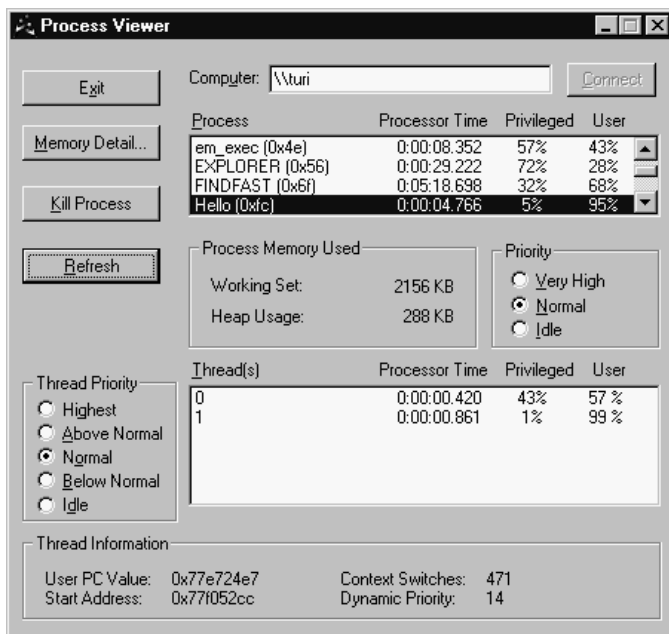


Abbildung 2.5: Hello mit zwei Threads

2.3 Die Klasse *CWnd*

2.3.1 Fensterklassen

Nachdem das Anwendungsobjekt erzeugt wurde, ist die erste Hälfte von HELLO geschafft. Nun ist es an der Zeit, die zweite Hälfte anzugehen und sich mit der Programmierung des Hauptfensters zu beschäftigen.

Um in den Microsoft Foundation Classes ein neues Fenster anzulegen, wird zunächst eine eigene Fensterklasse (im C++-Sinn) abgeleitet. Dazu gibt es eine Reihe vordefinierter Fensterklassen, die sich als Basisklassen eignen. Alle vordefinierten Fensterklassen entstammen letztlich der Klasse *CWnd*, in der ihre gemeinsamen Eigenschaften realisiert sind. Bei der Windows-Programmierung mit den Microsoft Foundation Classes muß man zwischen dem Fensterobjekt (d.h., der von *CWnd* oder einem seiner Nachfolger erzeugten Instanz) und dem Windows-Fenster, das auf dem Bildschirm dargestellt wird, unterscheiden. Ein Fensterobjekt wird durch Aufruf des Konstruktors erzeugt, während das Windows-Fenster eine Windows-interne Datenstruktur ist, die durch Aufruf von *Create* erzeugt wird. Diese Zweiteilung ist zum Verständnis des Anlegens und Zerstörens von Fenstern von Bedeutung und wird später noch ausführlich behandelt.

Im Gegensatz zur traditionellen Windows-Programmierung in C verfügt eine MFC-Fensterklasse nicht über eine sichtbare Fensterfunktion. Statt dessen werden eingehende Nachrichten durch das System der *Nachrichtentabellen* (*Message-Maps*) zu den passenden Methoden der Klasse umgeleitet. Genauer gesagt: Es besteht eine 1:1-Zuordnung zwischen Nachrichten und Methoden, so daß jede eingehende Nachricht in einen korrekt parametrisierten Aufruf der passenden Methode umgewandelt wird. Dabei kann zwischen drei Gruppen von eingehenden Nachrichten unterschieden werden:

1. *Command Messages*: Diese werden gesendet, wenn der Benutzer einen Menüpunkt anklickt, einen Button der Buttonleiste oder eine Beschleunigtaste drückt. Sie entsprechen WM_COMMAND-Messages und werden durch frei definierbare Methoden gepflegt.
2. *Notification Messages*: Diese werden durch Benutzeraktionen in Kindfenstern (z.B. Drücken eines Buttons in einer Dialogbox) erzeugt. Sie entsprechen WM_COMMAND-Messages mit einem bestimmten Benachrichtigungscode und werden durch frei definierbare Methoden gepflegt.
3. *Andere Nachrichten*: Diese werden durch andere System- oder Benutzeraktionen ausgelöst. Sie werden durch vordefinierte Methoden gepflegt, die in der abgeleiteten Klasse überlagert werden können.

In allen Fällen wird die Zuordnung zwischen Nachrichten und Methoden durch Einträge in der *Nachrichtentabelle* der Klasse vorgenommen. Diese Message-Map kann man sich bildlich als Tabelle vorstellen, in der zu jeder für die Anwendung bedeutsamen Nachricht eine Methode eingetragen wird. Beim Auftreten einer Nachricht sorgt dann die globale Fensterfunktion *AfxWndProc* dafür, daß die entsprechende Methode aufgerufen wird.

Methoden, die über Einträge in der Message-Map Fensternachrichten bedienen, verhalten sich ähnlich wie virtuelle Methoden. Sie werden aber nicht mit dem Schlüsselwort *virtual* deklariert, sondern mit dem Makro *afx_msg*. Die Konstruktion der Message-Map ermöglicht es, das Verhalten virtueller Funktionen ohne den ihnen eigenen Speicherbedarf nachzubilden. Falls also eine Windows-Nachricht auftritt, wird in der zugehörigen Klasse nach einem passenden Message-Map-Eintrag gesucht und die dort verzeichnete Methode aufgerufen.

Neben eingehenden Nachrichten gibt es bei der herkömmlichen Windows-Programmierung noch die *Control Messages*, die zur Manipulation von Steuerungen eingesetzt werden. Sie werden beispielsweise gebraucht, um in einem Editfeld den Text zu verändern, eine Listbox mit Daten zu füllen oder einen Radio-Button zu markieren. Bei Verwendung der Foundation Classes sind sie in den Methoden der Klassen für die Steuerungen verborgen und brauchen daher in der Regel nicht explizit gesendet zu werden.

Message-Maps können nicht nur in von *CWnd* abgeleiteten Fensterklassen enthalten sein, sondern auch in anderen MFC-Klassen. Entscheidendes Kriterium ist, ob die entsprechende Klasse von *CCmdTarget* abgeleitet wurde. Diese Klasse besitzt mit der Methode *OnCmdMsg* das Instrument, um WM_COMMAND-Nachrichten und Nachrichten von Benutzerschnittstellen zu den vorgesehenen Klassen umzuleiten (siehe Kapitel: »Das Message-Map-Konzept der MFC«). Tatsächlich ist *CCmdTarget* nicht nur Vaterklasse von *CWnd*, sondern auch von *CWinApp* und einigen anderen Klassen.

Auf diese Weise kann man bei der Entwicklung eines MFC-Programms den Message-Map-Eintrag in der Klasse plazieren, die aus Sicht der Programmsystematik am besten dafür geeignet ist. Erlaubt ist diese Vorgehensweise aber nur für Methoden, die auf ON_COMMAND-Nachrichten reagieren, also auf Nachrichten, die durch Benutzerschnittstellenobjekte wie Menüpunkte oder Steuerungen einer Dialogbox ausgelöst werden.

2.3.2 Erzeugen eines neuen Fensters

Ableiten einer Fensterklasse

Zunächst wird eine neue Fensterklasse *CMainWindow* von *CFrameWnd* abgeleitet und in *HELLO.H* deklariert:

```
class CMainWindow: public CFrameWnd
{
public:
    CMainWindow();
};
```

CFrameWnd ist unmittelbar von *CWnd* abgeleitet und besitzt die Eigenschaften eines überlappenden bzw. Popup-Fensters. Daher kann *CFrameWnd* in den meisten Anwendungen zur Realisierung des Hauptfensters benutzt werden. Lediglich Anwendungen, die nach dem MDI-Standard (MDI = Multiple Document Interface) arbeiten, verwenden in der Regel eine andere Basisfensterklasse. Darüber hinaus eignet sich *CFrameWnd* dafür, weitere überlappende Programmfenster zu generieren.

Die Unterschiede zwischen *CMainWindow* und *CFrameWnd* liegen im Konstruktor *CMainWindow* und der Methode *OnPaint*. Der Konstruktor wird überlagert, um darin durch Aufrufen von *Create* ein neues Windows-Fenster zu erzeugen. *OnPaint* ist die Methode, welche auf die WM_PAINT-Nachrichten reagiert und dafür sorgt, daß der »hello world«-Text geschrieben wird. Beide Methoden werden in *HELLO.CPP* implementiert.

Erzeugen einer Message-Map

Wichtiger Bestandteil der Klassendeklaration von *CMainWindow* ist das Makro *DECLARE_MESSAGE_MAP*. Es dient dazu, den Aufbau der Message-Map in der Deklaration der abgeleiteten Klasse vorzubereiten. Jede abgeleitete Fensterklasse, die mit Windows über Message-Maps kommunizieren will, muß in ihrer Deklaration dieses Makro verwenden. Dabei spielt es keine Rolle, ob das Makro am Anfang oder am Ende der Klassendeklaration untergebracht wird, es sollte nur nicht mitten in einem *public*-, *protected*- oder *private*-Bereich eingefügt werden. Da es selbst einen *private*- und *protected*-Bereich definiert, kann es die Sichtbarkeit der eigenen Member-Variablen beeinflussen, wenn nicht unmittelbar danach wieder eines der Schlüsselwörter *private*, *protected* oder *static* folgt.

Jede Methode, die über einen Eintrag in der Message-Map eine Windows-Nachricht verarbeiten soll, muß – wie *OnPaint* – mit dem Makro *afx_msg* deklariert werden. Zwar wird dieses Makro in der aktuellen Implementierung der Foundation Classes in Leerzeichen umgewandelt (und könnte daher ebenso gut weggelassen werden), in späteren Versionen (oder bei der Portierung auf andere Umgebungen) könnte *afx_msg* aber durchaus eine reale Bedeutung haben und sollte daher vor einer Message-Map-Methode immer angegeben werden.

Die Makros `DECLARE_MESSAGE_MAP` und `afx_msg` befinden sich in der Include-Datei `\PROGRAMME\MICROSOFT VISUAL STUDIO\VC98\MFC\INCLUDE\AFXWIN.H`, ebenso wie die meisten anderen MFC-Deklarationen.

2.3.3 Der Konstruktor von *CMainWindow*

Um aus der neu definierten Klasse *CMainWindow* ein Fensterobjekt erzeugen zu können, wird der Konstruktor überlagert. Er ruft dazu die Methode *Create* der Basisklasse *CFrameWnd* auf, um das Windows-Fenster zu erzeugen:

```
CMainWindow::CMainWindow()
{
    Create(        NULL,
                  "Hello World",
                  WS_OVERLAPPEDWINDOW,
                  rectDefault,
                  NULL,
                  NULL);
}
```

Listing: Erzeugen des Hauptfensters

Jede der vordefinierten Fensterklassen verfügt über eine spezielle Variante von *Create*. Aufgabe dieser Methode ist es, ein zur Klasse passendes Windows-Fenster anzulegen, denn der Konstruktor selbst erzeugt nur die internen Datenstrukturen der Instanz. Die Methode *CFrameWnd::Create* besitzt acht Parameter, welche die Eigenschaften des Fensters festlegen.

```
class CFrameWnd:
BOOL Create( LPCTSTR lpszClassName,
             LPCTSTR lpszWindowName,
             DWORD dwStyle=WS_OVERLAPPEDWINDOW,
             const RECT& rect = rectDefault,
             CWnd* pParentWnd = NULL,
             LPCTSTR lpszMenuName = NULL,
             DWORD dwExStyle = 0,
             CCreateContext *pContext = NULL );
```

Die Bedeutung der Parameter wird in Tabelle 2.3 erklärt. Da die letzten sechs Parameter Default-Parameter sind, kann man die Funktion bereits mit zwei Parametern, nämlich dem Namen der Fensterklasse und dem Titel des Fensters, aufrufen. Der Anschaulichkeit wegen wurden in *CMainWindow::CMainWindow()* jedoch sechs Parameter aufgeführt.

Parameter	Bedeutung
lpszClassName	Zeigt auf einen nullterminierten String, der den Klassennamen enthält. Dies kann ein vordefinierter Klassenname oder ein mit <i>AfxRegisterWndClass</i> definierter Klassenname sein. Falls NULL angegeben wird, verwendet die Funktion die vordefinierte Klasse mit den am besten passenden Attributen.
lpszWindowName	Zeigt auf einen nullterminierten String, der in der Titelleiste des Fensters angezeigt wird.
dwStyle	Spezifiziert die Fensterattribute. Hierbei handelt es sich um das erste Default-Argument: wird es nicht angegeben, so nimmt es den Wert <i>WS_OVERLAPPEDWINDOW</i> an (siehe Windows-Styles in der Class Library Reference).
rect	Dieses Rechteck spezifiziert die Position und Größe des angelegten Fensters. Wird hier der Wert <i>rectDefault</i> angegeben; so verwendet Windows seine eigene Voreinstellung. Das »hello world«-Programm verwendet <i>rectDefault</i> , um sich nicht weiter um die anfängliche Größe des Hauptfensters kümmern zu müssen.
pParentWnd	Ein CWnd-Zeiger auf das Vaterfenster. Dieser sollte NULL sein (und ist es auch als Vorgabe), wenn es sich um das Hauptfenster einer Anwendung handelt.
lpszMenuName	Zeigt auf einen nullterminierten String, der den Namen des Hauptmenüs aus der Resource-Datei angibt. Da »hello world« kein Hauptmenü enthält, steht hier NULL.
dwExStyle	Erweiterte Fensterattribute; werden in der Regel nicht benötigt.
pContext	Ein Zeiger auf ein <i>CCreateContext</i> -Objekt; wird in der Regel nicht benötigt.
Rückgabewert	<i>CFrameWnd::Create</i> gibt TRUE zurück, wenn die Funktion erfolgreich war, andernfalls FALSE.

Tabelle 2.3: Die Parameter von *CFrameWnd::Create*

Sollen von einer Fensterklasse keine weiteren Klassen mehr abgeleitet werden, kann der Aufruf von *Create* auch im Konstruktor erfolgen, andernfalls ist dies nicht erlaubt. In diesem Fall würde nämlich der abgeleitete Konstruktor (durch Aufruf von *Create* im Konstruktor der Basisklasse) bereits das Fenster erzeugen, bevor die Message-Map der eigenen Klasse fertig ist. Dies hätte zur Folge, daß die initialen Nachrichten *WM_CREATE* und *WM_PAINT* von den Methoden der Basisklasse bearbeitet würden. Um dieses Problem zu umgehen, sollte bei Fensterklassen, die zur weiteren Ableitung dienen, eine *Create*-Methode realisiert und so der zweigeteilte Erzeugungsprozeß beibehalten werden.

Im Gegensatz zur SDK-Programmierung braucht man sich beim Programmieren einer MFC-Anwendung in der Regel nicht mehr um das Registrieren einer Windows-Fensterklasse zu kümmern. Dies wird für alle relevanten Fenstertypen während der Initialisierung der Microsoft Foundation Classes erledigt. Daraus folgt allerdings, daß die in der Fensterklasse festgelegten Eigenschaften beim Anlegen des Fensters mit *Create* nicht mehr beeinflußt werden können. Probleme kann es dabei mit dem Cursor, dem Icon oder der Hintergrundfarbe geben:

1. Als Programmcursor wird `IDC_ARROW` verwendet, dies ist der normale Pfeil. Durch diese Vorgabe wird es schwierig, den Cursor während des Programmlaufs zu verändern. Schaltet man den Cursor nämlich mit `SetCursor(LoadCursor(...))` um, so ist das nur ein temporärer Erfolg. Bei der kleinsten Mausbewegung aktiviert Windows nämlich wieder den Pfeil. Um das zu verhindern, kann man entweder eigene Fensterklassen registrieren, die Foundation Classes ändern und recompilieren oder `SetCursor` in der Methode `OnSetCursor` aufrufen. Geht es nur darum, den Eieruhrcursor zu erzeugen, kann man dazu die Methoden `BeginWaitCursor` und `EndWaitCursor` der Klasse `CcmdTarget` verwenden.

```
class CcmdTarget:
    void BeginWaitCursor();
    void EndWaitCursor();
```

2. Das Programm-Icon wird bei Hauptfenstern und MDI-Kindfenstern durch die Konstante `AFX_IDI_STD_FRAME` festgelegt, bei MDI-Hauptfenstern durch die Konstante `AFX_IDI_STD_MDIFRAME`. Beide sind in der Schnittstellen-Datei `\PROGRAMME \MICROSOFT VISUAL STUDIO\VC\MFC\INCLUDE\AFXRES.H` deklariert. `AFXRES.H` sollte daher in jede Ressource-Datei einer MFC-Anwendung eingebunden werden, in `AFXWIN.H` wird sie automatisch eingebunden. Falls in der Ressource-Datei keine Icons mit diesem Namen definiert werden, verwenden die Microsoft Foundation Classes automatisch das Standard-Icon `IDI_APPLICATION`.
3. Der Programmhintergrund wird nur bei SDI-Hauptfenstern und MDI-Kindfenstern auf `COLOR_WINDOW+1`, also auf die Systemhintergrundfarbe, gesetzt. Bei den anderen Fensterklassen wird keine Hintergrundfarbe vorgegeben, so daß die Anwendung selbst für das Update des Fensterhintergrundes sorgen muß oder der Default-Fenster-Prozedur überträgt.

2.3.4 Überlagern der *OnPaint-Methode*

Zum jetzigen Zeitpunkt wäre die »hello world«-Anwendung schon lauffähig, würde allerdings noch keinen Text in der Client-Area ausgeben. Dazu ist es nötig, die Methode `OnPaint` zu überlagern, die beim Auftreten einer `WM_PAINT`-Nachricht aufgerufen wird.

```
void CMainWindow::OnPaint()
{
    CPaintDC      dc(this);
    CRect          rect;
    TRACE("OnPaint\n");
```

```

GetClientRect(rect);
dc.SetTextAlign(TA_BASELINE | TA_CENTER);
dc.TextOut(rect.right/2,rect.bottom/2,"Hello World",11);
}

```

Listing: Textausgabe in OnPaint

Der Device-Kontext

Bei der Windows-Programmierung ist zum Zeichnen im Arbeitsbereich eines Fensters immer der Zugriff auf einen *Device-Kontext* erforderlich. Dazu wird in SDK-Programmen vor dem Zeichnen zunächst die Funktion *BeginPaint* aufgerufen, um einen Handle auf den Device-Kontext zu beschaffen, und danach *EndPaint*, um ihn wieder freizugeben. Eine typische Fehlerquelle ist es dabei, das *EndPaint* zu vergessen, und dadurch bei jedem Aufruf wertvolle Systemressourcen zu verbrauchen.

Um dieses Problem auszuschalten, gibt es in MFC-Programmen die Klasse *CPaintDC*. Sie ist speziell dafür gedacht, in der *OnPaint*-Methode den Device-Kontext zu beschaffen. Dazu muß nur am Anfang der Methode ein *CPaintDC*-Objekt angelegt und am Ende wieder zerstört werden. Am einfachsten geht dies durch die Deklaration einer lokalen Variablen vom Typ *CPaintDC*. Der Konstruktor der Klasse ruft *BeginPaint* auf, um einen Device-Kontext zu erhalten, während der Destruktor diesen mit *EndPaint* automatisch wieder freigibt.

```

class CPaintDC : public CDC {
public:
    CPaintDC(CWnd* pWnd); throw( CResourceException );
    virtual ~CPaintDC();
    PAINTSTRUCT m_ps;
protected:
    HWND m_hWnd;
};

```

Der Konstruktor von *CPaintDC* erwartet ein Argument, nämlich einen Zeiger auf das Fensterobjekt, für das er benötigt wird. Da er in der *OnPaint*-Methode aufgerufen wird, reicht es aus, den *this*-Zeiger der aktuellen Fensterklasse zu übergeben.

CPaintDC ist von der Klasse *CDC* abgeleitet, welche die Schnittstelle zum GDI ist: *CDC* stellt beispielsweise Methoden zur Verfügung, um mit GDI-Objekten zu arbeiten, im Arbeitsbereich zu zeichnen oder auf die Attribute des Device-Kontexts zuzugreifen. Anders als *CDC* besitzt *CPaintDC* zusätzlich den Member *m_ps* zum Zugriff auf die *PAINTSTRUCT*-Struktur, die beim Aufruf von *BeginPaint* mit Informationen über den Arbeitsbe-

reich gefüllt wird. Diese ist identisch mit der Struktur, die auch bei der SDK-Programmierung benötigt wird (siehe `PAINTSTRUCT` in der Online-Dokumentation).

Ausgeben des Textes

Der Rest von `OnPaint` ist normale Windows-Programmierung, angepaßt an die Microsoft Foundation Classes. Zunächst wird mit `GetClientRect` ein Rechteck-Objekt vom Typ `CRect` gefüllt, das die Ausdehnung des Arbeitsbereichs angibt.

```
class CRect {
public :
    int l;
    int t;
    int r;
    int b;
}
```

Die Klasse `CRect` zur Darstellung von Rechtecken ist der `RECT`-Struktur sehr ähnlich. Sie besitzt dieselben Member-Variablen wie `RECT`, zusätzlich aber eine ganze Reihe von Methoden für die unterschiedlichsten Operationen auf Rechtecken. Ein `CRect`-Rechteck besteht immer aus zwei Punkten, der linken oberen und der rechten unteren Ecke. Diese werden durch die Koordinaten *left*, *top*, *right* und *bottom* definiert.

Die Koordinaten müssen positive Ganzzahlen sein und im Wertebereich einer Integer-Variablen liegen, und es muß sichergestellt werden, daß der Punkt (*left*, *top*) nicht rechts unter dem Punkt (*right*, *bottom*) liegt, daß also *right* $\geq$ *left* und *bottom* $\geq$ *top* gilt. Um die praktische Anwendbarkeit von `CRect` zu erhöhen, wurde es mit einigen Typ-Konvertern und Konstruktoren ausgestattet. Dadurch kann ein `CRect`-Objekt auch dann als Parameter an eine Funktion übergeben werden, wenn ein `RECT` oder `LPCRECT` vorgesehen ist.

Mit `SetTextAlign` wird nun festgelegt, daß der Text zentriert und seine Grundlinie auf den angegebenen Punkt ausgerichtet werden soll. Um besser beobachten zu können, wann der Client-Bereich neu gezeichnet wird, verwendet das Programm in Step 3 den Zähler `cnt` zur Veränderung der Schriftfarbe. Er wird nach dem Neuzeichnen inkrementiert, so daß durch einen Aufruf von `SetTextColor` bei jedem Aufruf von `OnPaint` je eine andere der Farben Schwarz, Rot, Grün oder Blau aktiviert wird.

Schließlich wird der im Puffer gespeicherte Text mit Hilfe von `TextOut` ausgegeben. Da der angegebene Punkt genau in der Mitte der Client-Area liegt, wird der Text im Fenster zentriert ausgegeben.

Die benötigten Routinen zur Bestimmung des Fensters sowie zur Textausgabe sind in der folgenden Aufstellung enthalten.

```
class CWnd:
    void GetClientRect ( LPRECT lpRect ) const;

class CDC:
    UINT SetTextAlign ( UINT nFlags );
    virtual COLORREF SetTextColor( COLORREF crColor );
    virtual BOOL TextOut( int x, int y, LPCTSTR lpszString,
                        int nCount );
    BOOL TextOut( int x, int y, const CString& str );
```

2.3.5 Erzeugen der Message-Map

Durch die bisherige Definition und Implementierung der Methode *OnPaint* in *CMainWindow* wurde noch keine Verbindung zu der Nachricht *WM_PAINT* hergestellt, sie würde daher an die Klasse *CWnd* weitergeleitet. Der Grund dafür ist, daß *OnPaint* keine echte virtuelle Funktion ist (sie wurde nicht mit dem Schlüsselwort *virtual* definiert, sondern mit dem Makro *afx_msg*), sondern erst mit Hilfe des Message-Map-Mechanismusses zu einer Methode gemacht werden muß, die sich wie eine virtuelle Funktion verhält.

Um die Message-Map zu konstruieren, muß sie zunächst in der abgeleiteten Fensterklasse mit Hilfe des Makros *DECLARE_MESSAGE_MAP* deklariert werden. Dies wurde bereits in *HELLO.H* getan. Zusätzlich muß nun jede Nachricht, durch die eine Methode aufgerufen werden soll, manuell in die Message-Map eingetragen werden. Dies geschieht zweckmäßigerweise in der Nähe der Implementierung der neuen Fensterklasse, hier also in der Datei *HELLO.CPP*.

```
BEGIN_MESSAGE_MAP(CMainWindow,CFrameWnd)
    ON_WM_PAINT()
END_MESSAGE_MAP()
```

Am Anfang der Deklaration muß das Makro *BEGIN_MESSAGE_MAP* aufgerufen werden. Es erwartet zwei Parameter: den Namen der neuen Fensterklasse und den Namen ihrer Basisklasse. Hierdurch wird die Verbindung zwischen Basisklasse und abgeleiteter Klasse hergestellt, mit der später bei Auftreten einer Nachricht nach einer passenden Methode gesucht wird.

Nun wird für jede Methode, die einer Nachricht zugeordnet werden soll, ein Eintrag in die Message-Map gemacht. Hierfür steht eine Reihe von Makros zur Verfügung, von denen die wichtigsten in drei Gruppen unterteilt werden können:

1. Makros für WM_COMMAND-Nachrichten, die entweder durch Menüselektionen oder Tastendrucke des Benutzers verursacht wurden. Sie haben die Form:

```
ON_COMMAND(id, memberFxn)
```

Dabei wird bei Auftreten einer WM_COMMAND-Nachricht mit der Kennung *id* die Methode *memberFxn* aufgerufen. Diese muß die folgende Form haben:

```
afx_msg void memberFxn();
```

2. In die zweite Gruppe gehören die Makros für Notifikationscodes von Kindfenstern. Beispiel eines solchen Makros könnte sein:

```
ON_CBN_DBLCLK(id, memberFxn)
```

Hiermit wird eine Verbindung zwischen der ON_COMMAND-Nachricht einer Combobox mit der Nummer *id* und dem Notifikationscode CBN_DBLCLK (Doppelklick aufgetreten) und der Methode *memberFxn* hergestellt. Letzere ist wie folgt zu deklarieren:

```
afx_msg void memberFxn();
```

Diese Art von Makros und ihre Behandlung wird im Zusammenhang mit den Themen *Steuerungen* und *Dialoge* in diesem Buch genauer beschrieben.

3. Zuletzt wären da noch die Makros für allgemeine Windows-Nachrichten. Hiermit werden (fast) alle anderen Windows-Nachrichten den geeigneten Methoden zugeordnet. Beispiel eines solchen Makros ist:

```
ON_WM_PAINT()
```

Hiermit wird eine Verbindung zwischen der Nachricht WM_PAINT und der Methode *OnPaint* der Fensterklasse hergestellt. *OnPaint* muß immer die folgende Form haben:

```
afx_msg void OnPaint();
```

Kennzeichen der Makros dieser dritten Gruppe ist es, daß der Name der aufzurufenden Methode nicht mehr frei vorgegeben werden darf, sondern durch das Makro selbst bestimmt wird. Deshalb ist es auch nicht mehr nötig, diesen beim Aufruf des Makros anzugeben – vielmehr sind die meisten Makros dieser Gruppe parameterlos.

Es gibt mehr als hundert verschiedene Makros zum Erzeugen von Einträgen in der Message-Map. Die meisten davon sind selbsterklärend, weil ihre Namen einem allgemeinen Schema untergeordnet sind: Es ist der

Name der zugehörigen Windows-Nachricht mit dem Präfix `ON_`. Wichtig ist, daß die angegebene Member-Funktion den syntaktischen Regeln, die das Makro erfordert, genügt.

Am Ende der Deklaration der Message-Map wird das Makro `END_MESSAGE_MAP` aufgerufen. Danach dürfen keine weiteren Einträge in die Message-Map eingefügt werden.

Das Verständnis der Arbeit mit Messages ist wichtig für das Verständnis der Windows-Programmierung an sich. Wenn Sie sich andere Programme oder Programmiersprachen wie MS Access oder Visual Basic ansehen, werden Sie dieses Prinzip wiederfinden.



Fehler bei der Deklaration der Message-Map

Bei der Konstruktion der Message-Map können vielerlei Fehler passieren, die mehr oder weniger schwierig zu finden sind. Da das System der Message-Maps nicht direkter Bestandteil der Sprache ist, sondern mit Hilfe von Standard-Eigenschaften des C++-Compilers bzw. seines Präprozessors realisiert wurde, gibt es beim Übersetzen des Programms nicht immer sinnvolle Meldungen, wenn bei der Deklaration der Message-Map ein Fehler gemacht wurde. Immerhin gibt es aber in den meisten Fällen wenigstens einen Compilerfehler, der einen Hinweis darauf gibt, daß das Problem in der Message-Map zu suchen ist. Nur wenn ein Eintrag völlig vergessen wurde, gibt es überhaupt keine Fehlermeldung, und das Problem kann erst zur Laufzeit des Programms bemerkt werden.

2.3.6 Zerstören eines Fensters

Das Zerstören eines Fensters ist in MFC-Programmen etwas schwieriger als in SDK-Programmen, da außer dem Windows-Fenster das Fenster-Objekt zerstört werden muß. Der eigentliche Auslöser zum Schließen des Fensters ist die Methode *DestroyWindow*. Sie kann entweder direkt von der Anwendung aufgerufen, oder aber durch eine Benutzeraktion ausgelöst werden, welche die Nachricht `WM_CLOSE` erzeugt. Die Standardreaktion auf `WM_CLOSE` besteht darin, *DestroyWindow* aufzurufen, um damit das Fenster zu schließen. Dieser Vorgang läuft wie folgt ab:

1. Die Anwendung erhält die Nachricht `WM_CLOSE`. Dies wird meist dadurch ausgelöst, daß der Benutzer `[Alt]` `[F4]` gedrückt oder im Systemmenü den Punkt SCHLIESSEN ausgewählt hat. Dadurch wird die Methode *OnClose* von *CWnd* aufgerufen. An dieser Stelle könnte das Programm beim Benutzer nachfragen (einfach *OnClose* in der abgeleiteten Klasse neu definieren), ob er das Fenster wirklich schließen will, ob noch Daten zu sichern sind oder ähnliches. Soll das Fenster wirklich geschlossen werden, ruft das Programm die Methode *DestroyWindow* auf.

2. *DestroyWindow* zerstört das Windows-Fenster und sendet zwei Nachrichten an die Anwendung. Zunächst wird ihr mit *WM_DESTROY* mitgeteilt, daß ihr Fenster zerstört wurde. Falls es sich um das Hauptfenster handelte, ruft die Anwendung nun die Funktion *PostQuitMessage* (s.u.) auf. Danach wird ihr die Nachricht *WM_NCDESTROY* gesendet, um den Non-Client-Bereich zu zerstören. Hier gilt es aufzupassen: Falls das zerstörte Fenster von der Klasse *CFrameWnd* abgeleitet wurde, ruft *OnNcDestroy* zum Schluß *delete this* auf, um das Programmobjekt zu löschen! Bei Fensterklassen, die nicht von *CFrameWnd* abgeleitet wurden, unterbleibt dies. Hieraus ergeben sich einige Konsequenzen bezüglich der Speicherklassse von Fensterobjekten und deren Zerstörung, die unten näher erläutert werden.
3. Falls *PostQuitMessage* aufgerufen wurde, sendet Windows die Nachricht *WM_QUIT* an die Anwendung. Dies führt dazu, daß die Nachrichtenschleife beendet wird, und die Methoden *ExitInstance* und *AfxWinTerm* aufgerufen werden.

```
class Cwnd:
    afx_msg void OnClose ();
    afx_msg void OnDestroy ();
    afx_msg void OnNcDestroy ();
    virtual BOOL DestroyWindow ();
global:
    void ::PostQuitMessage ( nExitCode );
```

2.4 Das Beispiel Schritt für Schritt

Im folgenden Abschnitt lernen Sie, wie Sie Ihr erstes Programm mit Visual C++ schreiben. Alles, was Sie dazu wissen müssen, wurde in den vorangegangenen Abschnitten erläutert. Das Beispielprogramm wird Schritt für Schritt erweitert. Sie erhalten dafür jeweils eine Aufgabe, deren Lösung im folgenden Abschnitt beschrieben wird. Zuerst wenden wir uns jedoch dem Erzeugen des Projekts zu.

2.4.1 Das Anlegen des Hello-Projekts

Unser erstes Programm soll ein Minimum an Ballast von der MFC enthalten. Allerdings soll die Klassenbibliothek Verwendung finden, um die Arbeitsweise eines Windows-Programms mit der MFC zu verstehen. Deshalb verwenden wir ein Projekt vom Typ *Win32 Application*.

Zuerst wird ein neues Projekt vom Typ *Win32 Application* angelegt. Schauen Sie sich dazu nochmals den Abschnitt »Projekte« an. Rufen Sie im Menü *DATEI* den Punkt *NEU* auf. Wählen Sie die Seite *PROJEKTE*. Geben Sie dem Projekt den Namen »Hello« und achten Sie auf das Projektverzeichnis. Drücken Sie anschließend die *OK*-Taste, um das Projekt anzulegen (siehe Abbildung 2.6).

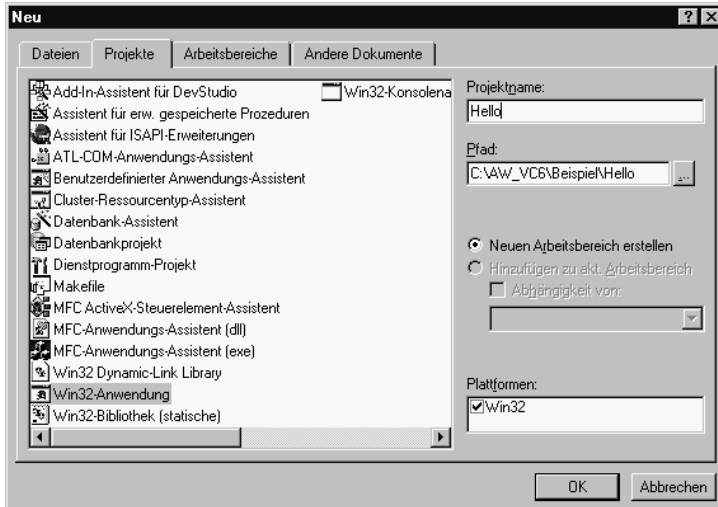


Abbildung 2.6: Das Anlegen des Hello-Projekts

In der Version 6 sind jetzt auch für die Win32-Anwendung verschiedene Typen vordefiniert (siehe Abbildung 2.7). Für unsere Zwecke ist der erste Typ, *ein leeres Projekt*, genau das Richtige. Beide anderen Typen klingen zwar verlockend, entsprechen aber nicht den oben gestellten Forderungen. Die *einfache Win32-Anwendung* erzeugt eine Quelldatei mit dem Aufruf von der API-Funktion *WinMain* ohne Funktionalität. Variante drei, die *typische »Hallo Welt!«-Anwendung*, basiert auch auf dem Win32-API und nicht auf der MFC.

Nachdem Sie dieses Projekt angelegt haben, ist es im Developer Studio geöffnet. Jedoch sind im Arbeitsbereich keinerlei Klassen oder Dateien vorhanden. Für unser Projekt benötigen wir zumindest eine Quell- und eine Header-Datei (Implementierungs- und Schnittstellendatei). Legen Sie diese über DATEI|NEU mit dem Namen »Hello« an (Abbildung 2.8).

Da das Kontrollkästchen »Dem Projekt hinzufügen« aktiviert ist, werden die so erzeugten Dateien sofort in die Make-Datei des Projekts aufgenommen. Sie können nun mit dem Editieren der Quelltexte beginnen.



Abbildung 2.7: Projekttypen einer Win32-Anwendung

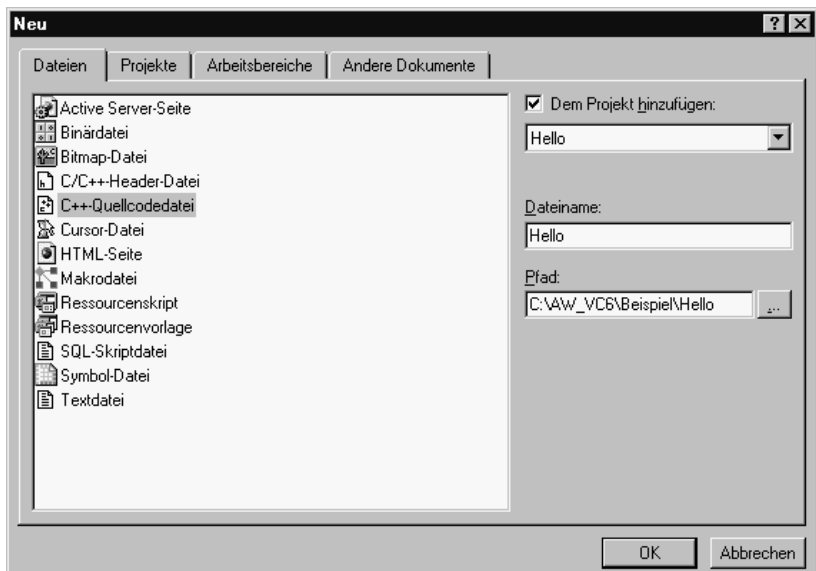


Abbildung 2.8: Anlegen von Quell- und Headerdateien

Legen Sie eine Klasse *CHelloApp* an, die von *CWinApp* abgeleitet wird. Die Methode *InitInstance* muß in unserer Anwendung überschrieben werden. Weiterhin wird eine Fensterklasse benötigt, die von *CFrameWnd* abgeleitet wird. Sie soll *CMainWindow* heißen und nur einen Konstruktor enthalten. Schützen Sie die Header-Datei vor Mehrfach-Einbindung. In der Methode *InitInstance* wird das Hauptfenster erzeugt und angezeigt.



Bevor Sie die Anwendung übersetzen und starten können, muß in den Einstellungen des Projekts noch eine Änderung vorgenommen werden. Wir müssen dem Projekt mitteilen, daß die MFC verwendet werden soll. Dazu müssen zum einen die Datei *AFXWIN.H* inkludiert und zum anderen die MFC eingebunden werden. Rufen Sie die Projekteinstellungen über das Menü PROJEKT|EINSTELLUNGEN oder **Alt** **F7** auf. Auf der Seite *Allgemein* wird in der Listbox *Microsoft Foundation Classes* der Punkt »MFC in einer gemeinsam genutzten DLL verwenden« aktiviert und mit OK bestätigt.

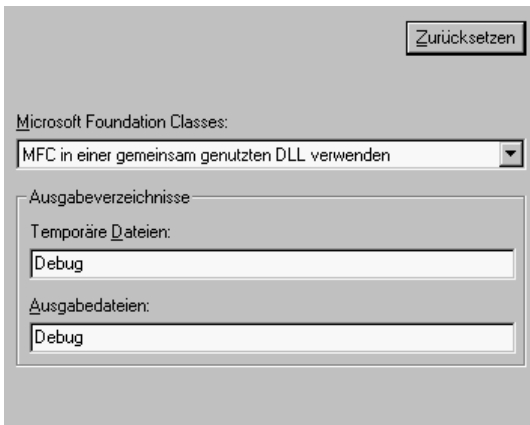


Abbildung 2.9: Einstellungen für die MFC im Projekt vornehmen

2.4.2 Hello – Schritt 1

Im folgenden Quelltext ist die Datei *Hello.H* dargestellt. Darin sind die Deklaration der Applikationsklasse *CHelloApp* und der Hauptfensterklasse *CMainWindow* enthalten.

```
//Mehrfach-Inkludierung verhindern
#ifdef _HELLO_H_
#define _HELLO_H_
//Applikationsklasse
class CHelloApp : public CWinApp
{
```

```

public :
    BOOL InitInstance();
};
//Hauptrahmenklasse
class CMainWindow : public CFrameWnd
{
public :
    CMainWindow();           //Konstruktor
};
#endif

```

Die Klasse *CMainWindow* erhält lediglich einen Konstruktor, in dem das Fenster mit der Methode *Create* der Klasse *CFrameWnd* angelegt wird. Der Name des Fensters wird im zweiten Parameter von *Create* angegeben. Da alle anderen Parameter Standardvereinbarungen besitzen, genügt die Angabe der ersten beiden Parameter. In der Methode *InitInstance* wird dann eine neue Instanz der Klasse *CMainWindow* angelegt und der Member-Variablen *m_pMainWnd* von *CWinApp* zugewiesen. Anschließend wird die Methode *ShowWindow* aufgerufen, um das Fenster anzuzeigen. Die eigentliche Anwendung wird durch das Anlegen einer Variablen vom Typ *CHelloApp*, also einer Instanz von *CWinApp*, erzeugt. Damit ist die erste Anwendung komplett und kann erstellt und gestartet werden.

```

#include <afxwin.h>
#include "hello.h"
static CHelloApp HelloApp;
/*****
/*      CHelloApp Member-Funktionen      */
*****/
BOOL CHelloApp::InitInstance()
{
    m_pMainWnd = new CMainWindow();
    m_pMainWnd->ShowWindow(m_nCmdShow);
    return TRUE;
}
/*****
/*      CMainWindow Member-Funktionen      */
*****/
CMainWindow::CMainWindow()
{
    Create(NULL, "Hello World",
           WS_OVERLAPPEDWINDOW,
           rectDefault, NULL, NULL);
}

```

Erweitern Sie das Programm aus Schritt 1 so, daß in der Mitte des Fensters der Text »Hello World« ausgegeben wird.



2.4.3 Hello – Schritt 2

Entsprechend der Aufgabe soll ein Text im Hauptfenster des Programms ausgegeben werden. *OnPaint* ist die Methode, welche auf die WM_PAINT-Nachrichten von Windows reagiert und dafür sorgt, daß das Fenster neu gezeichnet wird. Alles, was in diesem Fenster ausgegeben werden soll, muß in der Methode *OnPaint* kodiert werden. Das Schreiben (genauer gesagt: das Überschreiben der virtuellen Funktion der Basisklasse) der Methode ist der erste Punkt, man muß dann aber auch dafür sorgen, daß die eigene *OnPaint*-Methode aufgerufen wird. Dazu muß die Nachricht WM_PAINT auf die Methode *OnPaint* umgeleitet werden. Hier kommt die Message-Map zum Einsatz. In der Klassendefinition von *CMainWindow* wird das Makro DECLARE_MESSAGE_MAP eingefügt, um eigene Nachrichtenbehandlungsroutinen zu integrieren.

```
class CMainWindow: public CFrameWnd
{
public:
    CMainWindow();
    afx_msg void OnPaint();
    DECLARE_MESSAGE_MAP()
};
```

Bei der Implementierung kommen dann die Makros BEGIN_MESSAGE_MAP und END_MESSAGE_MAP zum Einsatz. Zwischen diesen beiden Makros werden alle Nachrichten geschrieben, auf die im Programm reagiert werden soll.

```
BEGIN_MESSAGE_MAP(CMainWindow, CFrameWnd)
    ON_WM_PAINT()
END_MESSAGE_MAP()
```

In unserem Fall ist es nur die WM_PAINT-Nachricht. Die zugehörige *OnPaint*-Methode wird im Anschluß implementiert.

```
void CMainWindow::OnPaint()
{
    CPaintDC          dc(this);
    CRect             rect;
    TRACE("OnPaint\n");
    GetClientRect(rect);
    dc.SetTextAlign(TA_BASELINE | TA_CENTER);
    dc.TextOut(rect.right/2, rect.bottom/2, "Hello World", 11);
}
```

Hier werden alle oben beschriebenen Funktionen benötigt: `SetTextAlign`, um die Ausrichtung des Textes festzulegen und `TextOut`, um den Text in der Mitte des Fensters, dessen Größe mit `GetClientRect` ermittelt wurde, auszugeben. Der dazu benötigte Device Context wird als lokale Variable angelegt und damit bei Verlassen der Methode `OnPaint` auch automatisch wieder freigegeben.

Nach dem Erstellen der Anwendung sollte der Text in der Mitte des Fensters ausgegeben werden. Wenn Sie die Größe des Fensters ändern, bleibt der Text weiterhin in der Mitte. Das liegt daran, daß bei jeder Veränderung des Fensters die Methode `OnPaint` aufgerufen und somit immer die richtige Größe des Fensters ermittelt wird.



Als weiterer Zusatz wurde auf den auf der CD mitgelieferten Programmen in die Funktionen der Message-Map jeweils das TRACE-Makro eingefügt. Als Parameter wird eine Zeichenkette übergeben. Diese kann, wie bei der Funktion `printf`, Parameter enthalten. Immer dann, wenn die Applikation auf ein TRACE-Makro trifft, wird im Debug-Fenster der Entwicklungsumgebung der entsprechende Text ausgegeben, vorausgesetzt, das Programm wurde im Debug-Modus übersetzt und gestartet. Diese Arbeitsweise ermöglicht es, den Programmablauf zu verfolgen und auch Fehler zu finden. In unserem Fall werden Sie sehen, wann und wie oft die `OnPaint`-Methode aufgerufen wird. Man sollte sich deshalb nicht scheuen, diese Makros an geeigneter Stelle einzuflechten.



Im nächsten Schritt wird es farbig. Der Text soll auf Druck der linken Maustaste in vier wechselnden Farben ausgegeben werden.

2.4.4 Hello – Schritt 3

Diese Erweiterung sollte Ihnen leichtfallen. Sie müssen nur in der Message-Map einen weiteren Eintrag hinzufügen, um auf die Nachricht `WM_LBUTTONDOWN` zu reagieren, die gesendet wird, sobald die linke Maustaste gedrückt wird. Die zugehörige Methode `OnLButtonDown` muß daraufhin implementiert werden. In dieser Methode werden nur zwei Funktionen aufgerufen: `InvalidateRect`, um das Fenster als ungültig zu deklarieren und `UpdateWindow`, was die Nachricht `WM_PAINT` auslöst.

```
void CMainWindow::OnLButtonDown(UINT nFlag,CPoint point)
{
    InvalidateRect(NULL,FALSE);
    UpdateWindow();
}
```

Um den Verwaltungsaufwand innerhalb von Windows so gering wie möglich zu halten, werden nur die Fenster oder Teile davon aktualisiert, deren Neuausgabe wirklich nötig ist. Ein verdecktes Fenster muß beispielsweise

nicht neu gezeichnet werden. Dafür ist die Methode *InvalidateRect* zuständig. Wo kommt aber nun die Farbe ins Spiel? Da alle Ausgaben in der Methode *OnPaint* erfolgen, kann auch nur dort die Farbe festgelegt werden. Die Methode *OnLButtonDown* bewirkt nur einen vorzeitigen Aufruf von *OnPaint*. Innerhalb von *OnPaint* sollte deshalb als erstes eine Variable deklariert werden, die zählt, wie oft WM_PAINT versandt wurde. Um den Wert der lokalen Variablen bei nacheinanderfolgenden Aufrufen von *OnPaint* beizubehalten, muß sie statisch deklariert werden.

```
void CMainWindow::OnPaint()
{
    CPaintDC dc(this);
    CRect rect;
    static int cnt = 0;
    GetClientRect(rect);
    dc.SetTextAlign(TA_BASELINE | TA_CENTER);
    switch (cnt) {
        case 0 : dc.SetTextColor(RGB(0,0,0)); break;
        case 1 : dc.SetTextColor(RGB(255,0,0)); break;
        case 2 : dc.SetTextColor(RGB(0,255,0)); break;
        case 3 : dc.SetTextColor(RGB(0,0,255)); break;
    }
    dc.TextOut(rect.right/2,rect.bottom/2,
               "Hello World",11);
    cnt = (cnt+1)%4;
}
```

Listing: Textausgabe in verschiedenen Farben

Der Trick besteht nur darin, abhängig vom Zählerstand die Farbe mit der Methode *SetTextColor* zu setzen. Die Farbe wird hier als RGB-Wert angegeben. Zum Schluß wird der Zähler erhöht und mit Modulo 4 dividiert.

Erweitern Sie das Programm so, daß es ein Symbol (Programmicon) und einen neuen Cursor erhält. Der Cursor soll sich beim Drücken der linken Maustaste ändern.



2.4.5 Hello – Schritt 4

Die Lösung dieser Aufgabe wurde schon im Kapitel »Anlegen einer neuen Icon-Ressource« erläutert. Verfahren Sie in der dort beschriebenen Reihenfolge. Sie können auch die Cursor und das Symbol von der CD in Ihr Projekt einfügen.

Das Setzen des neuen Cursors erledigen Sie am einfachsten in der Methode *OnSetCursor*, die von der Nachricht WM_SETCURSOR aufgerufen wird.

```
BOOL CMainWindow::OnSetCursor(CWnd* pWnd,  
                               UINT nHitTest,  
                               UINT message )  
{  
    SetCursor(hCursor1);  
    return TRUE;  
}
```

Darin setzen Sie einfach den aktuellen Cursor neu. Die neuen Cursor müssen zuvor aus der Ressourcdatei geladen werden. Das erfolgt im Konstruktor der Fensterklasse mit der Methode *LoadCursor*.

```
CMainWindow::CMainWindow()  
{  
    Create(NULL, "Hello World",  
           WS_OVERLAPPEDWINDOW,  
           rectDefault, NULL, NULL);  
    hCursor2 = AfxGetApp()->LoadCursor(IDC_HAMMER2);  
    hCursor1 = AfxGetApp()->LoadCursor(IDC_HAMMER1);  
}
```

Damit haben Sie Ihr erstes Programm geschrieben und schon um eine Reihe von Funktionen erweitert.



Auf der CD befinden sich zwei weitere Fassungen von HELLO, die sich mit der Programmierung von Hintergrundaktivitäten beschäftigen. Die zugehörigen Erklärungen finden Sie im Kapitel »Multitasking in eigenen Programmen«.



Sie können HELLO auch noch um andere Funktionen erweitern. Versuchen Sie doch einmal, Grafikelemente wie Linien, Kreise oder Rechtecke auszugeben – und das vielleicht auf Mausklick. Informationen zum Ausgeben grafischer Objekte finden Sie in der Online-Hilfe zur Klasse *CDC*.



Mit den Assistenten
zur Anwendung

TEIL II



Die Assistenten

3

3.1	Der Anwendungs-Assistent	118
3.1.1	Generieren eines Projekts	119
3.1.2	Vom Anwendungs-Assistenten erzeugte Dateien	126
3.1.3	Test des neuen Programms	128
3.1.4	Neuerungen in InitInstance	128
3.1.5	Initialisierungen in CMainFrame	131
3.2	Der Klassen-Assistent	132
3.2.1	Anlegen von Klassen, Member-Funktionen und Member-Variablen	132
3.2.2	Assistentenleiste	135
3.2.3	Anlegen von Funktionen und Variablen über die Klassen-Ansicht	137

3.1 Der Anwendungs-Assistent

Nachdem Sie schon den Umgang mit Projekten gelernt haben, geht es nun darum, mit Hilfe des Anwendungs-Assistenten ein neues Projekt anzulegen. Wir beschäftigen uns hier ausschließlich mit dem Assistenten zum Anlegen von EXE-Dateien – also ausführbaren Windows-Programmen.

Der Anwendungs-Assistent ist fester Bestandteil des Developer Studios. Mit Hilfe dieses Zauberers werden komplette Visual C++-Projekte generiert. Diese sind sofort lauffähig, ohne auch nur eine Zeile Quelltext von Hand zu schreiben. Es werden alle nötigen Quell-, Header-, Ressource- und Projekt-Dateien inklusive der Make-Datei erzeugt.

Damit wird die Grundfunktionalität eines Windows-Programms mit diversen Zusätzen wie Online-Hilfe, Statuszeile oder Symbolleiste erzeugt. Mittels einiger weiterer Optionen können Datenbank- oder OLE-Funktionalität hinzugefügt werden. Doch damit noch nicht genug: Die Sammlung von *Komponenten und Steuerelementen* hält vorgefertigte Programm-Module bereit, die bei Bedarf nur noch eingefügt werden müssen. Dies erspart nicht nur Zeit beim Programmieren, sondern erhöht auch die Sicherheit des Programms. Die vorgefertigten Standard-Komponenten sind im allgemeinen bereits getestet und somit fehlerfrei. Sie können diese natürlich auch um eigene Komponenten ergänzen.

Einer Vielzahl von Vorteilen stehen aber auch einige wenige Nachteile gegenüber: Das vom Anwendungs-Assistenten erzeugte *Programmgerüst* ist zwar durch Optionen in weiten Grenzen variierbar, aber es besitzt einen festen Rahmen. Dieser Rahmen wird durch die *Dokument-Ansicht-Architektur* (auch Document-View) beschrieben. Das heißt, eine Applikation besitzt entweder ein Dokument zur Datenhaltung (SDI) und eine oder mehrere Views, die die Daten in verschiedenen Ansichten darstellen, oder mehrere Dokumente (MDI) mit ebenfalls einer oder mehreren Ansichten. Wenn aber ein Programm wie zum Beispiel »hello world« dieses aufwendige Programmgerüst gar nicht benötigt, muß auf das Generieren der Quelldateien verzichtet werden. Das trifft natürlich auch auf Programme zu, die keine Fenster besitzen. In der Version 6 von Visual C++ wurde der Anwendungs-Assistent nochmals um einige Möglichkeiten ergänzt, von denen Sie eine in den folgenden Kapiteln kennenlernen. Zu den Neuerungen zählen auch Anwendungen im Web- oder Explorer-Stil.



Eine weitere Art der Anwendung wird später noch beschrieben, und zwar eine dialogbasierende Applikation, die ebenfalls mit dem Anwendungs-Assistenten erzeugt werden kann. Im Beispielpogramm zu diesem Abschnitt wird nicht auf das dokumentorientierte Arbeiten eingegangen. Wir benutzen nur die Ansicht zur Ausgabe von Informationen während der Laufzeit des Programms. Die Erklärungen zur Dokument-Ansicht-Architektur erfolgen später.

Ein großer Vorteil der Verwendung des Anwendungs-Assistenten ist die Vereinheitlichung von Quelltexten. Wer kennt nicht das Problem, ein altes Programm erweitern oder verändern zu müssen, das vielleicht noch von einem anderen Programmierer geschrieben wurde. Die Einarbeitung in das Projekt benötigt sehr viel Zeit, da jeder seinen eigenen Stil hat. Kommentare sind meist nur unzureichend im Programm vorhanden. Da die vom Anwendungs-Assistenten erzeugten Programme alle gleich aufgebaut sind, fällt es leichter, sich darin zurechtzufinden. Wurden zusätzlich noch allgemeingültige Namenskonventionen benutzt, wird ein Programm noch leichter lesbar.

Alle vom Anwendungs-Assistenten erzeugten Programme machen von der MFC Gebrauch. Auch alle damit verbundenen Projekt-Einstellungen werden automatisch vorgenommen.

Dieser Teil wird ebenfalls von einem Beispielprogramm begleitet. Wir haben es *SuchText* genannt. Die Aufgabe dieses Programms ist es, in ASCII-Dateien nach Text zu suchen und diesen anzuzeigen. Nun wird das zwar von jedem Textverarbeitungsprogramm erledigt; unser Programm kann aber mit Dateien beliebiger Größe arbeiten. Erreicht wird das dadurch, daß nicht zuerst die gesamte Datei eingelesen wird, sondern immer nur ein Teil davon. Der gefundene Text wird im Fenster ausgegeben. Alle Möglichkeiten des Programms werden Schritt für Schritt in den folgenden Abschnitten erläutert.



3.1.1 Generieren eines Projekts

Der Aufruf des Anwendungs-Assistenten erfolgt im Menü DATEI über den Punkt *NEU|PROJEKTE*. Die Vorgehensweise entspricht derjenigen beim Anlegen eines neuen Projektes (Arbeitsbereich). In der Liste der verfügbaren Projekttypen wird allerdings nicht die Win32 Anwendung, sondern *MFC-Anwendungs-Assistent (exe)* gewählt. Es müssen der Projektname und das Projektverzeichnis eingegeben werden. Dabei wird automatisch der Name des Projekts auch für das Verzeichnis vorgeschlagen. Der Verzeichnisname kann aber nachträglich noch verändert werden. Da der Name des Projekts zum Teil auch für Dateinamen und Bezeichnungen von Klassen benutzt wird, sollte dieser sinnvoll gewählt werden. Schließlich kann noch gewählt werden, ob ein neuer Arbeitsbereich erstellt werden soll oder dieser zum aktiven Projekt hinzugefügt wird.

Der Anwendungs-Assistent macht von langen Dateinamen Gebrauch.

Die im folgenden dargestellte Vorgehensweise zum Anlegen einer neuen Anwendung können Sie gleich nachvollziehen. Wir benutzen das hier erzeugte Projekt für unser Beispielprogramm *SuchText*.



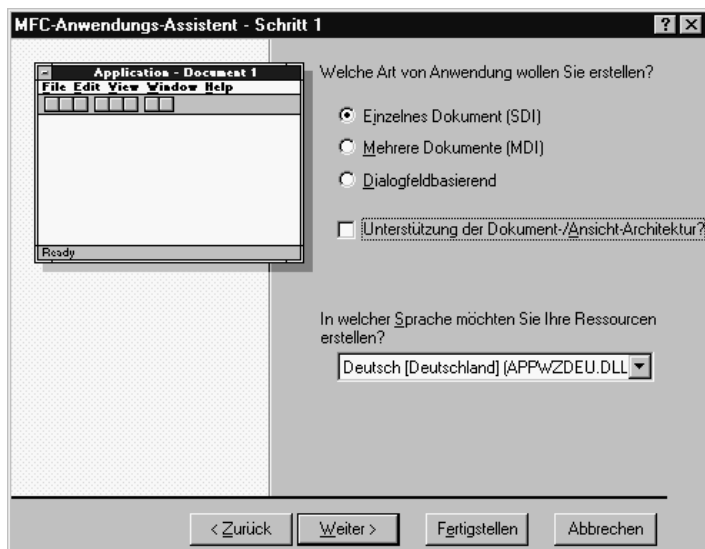


Abbildung 3.1: Schritt 1 des Anwendungs-Assistenten

Nach dem Drücken der Schaltfläche OK wird man Schritt für Schritt durch den Anwendungs-Assistenten beim Erstellen des Programmgerüsts begleitet. Im ersten Schritt muß angegeben werden, welcher Typ der Applikation erzeugt werden soll und welche Sprache für die Ressourcen benutzt wird. Über die Schaltflächen ZURÜCK und WEITER wird im Anwendungs-Assistenten ein Schritt vor- oder zurückgegangen. FERTIGSTELLEN beendet den Anwendungs-Assistenten mit dem Generieren der Projektdateien. Über ? erhält man eine Erklärung der Parameter im jeweiligen Schritt des Anwendungs-Assistenten. Nach dem Drücken der ?-Schaltfläche kann mit der Maus das Element angewählt werden, zu dem jeweils Hilfe gewünscht wird.

Wir wählen im ersten Schritt den Typ *Einzelnes Dokument* (SDI), da in unserem Programm kein Dokument und nur eine Ansicht benötigt werden. Zusätzlich verzichten wir auf die Dokument-Ansicht-Architektur. Diese wird in Teil vier erläutert. Durch Deaktivieren des Kontrollkästchens *Unterstützung der Dokument-/Ansicht-Architektur?* wird diese nicht in das Programmgerüst eingefügt. Des weiteren können Sie in diesem Dialog die Ressourcen-Sprache der MFC festlegen.

In Schritt 2 wird die Datenbank-Unterstützung der späteren Applikation festgelegt (Abbildung 3.2). Da diese nicht benötigt wird, wird *keine* gewählt und mit Schritt 3 fortgefahren (Abbildung 3.3). Ansonsten können hier alle Datenbanken eingebunden werden, die eine ODBC-Schnittstelle besitzen. Interessant ist hier die Einbindung von MS Access-Datenbanken.

Diese können über die DAO-Schnittstelle (Data Access Objects) direkt gelesen und geschrieben werden, was Geschwindigkeitsvorteile gegenüber ODBC bringt. Über die Schaltfläche DATENQUELLE kann hier schon die zu benutzende Datenbank festgelegt werden.

Die Programmierung einer Datenbank-Anwendung wird im letzten Teil dieses Buches erläutert.

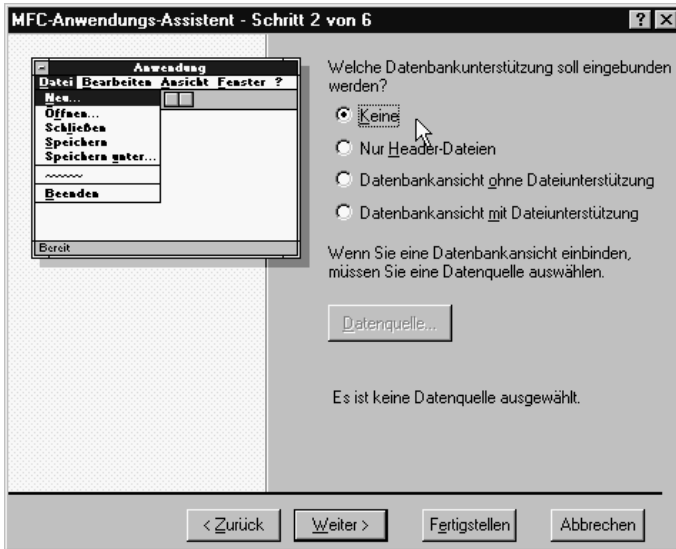


Abbildung 3.2: Schritt 2 des Anwendungs-Assistenten

Für Schritt 3 wird auf weiterführende Literatur verwiesen. Hier wird eingestellt, welche OLE-Funktionalität die Anwendung haben soll. Wichtig für uns ist hier nur das Kontrollkästchen *ActiveX-Steuerelemente*. Sicher haben Sie schon in Zusammenhang mit dem Internet von ActiveX-Controls gehört. Diese sind aber auch für eigene Anwendungen interessant, und das vor allem deshalb, da sie die Nachfolger von VBX- und OCX-Controls sind. Soll also ein ActiveX-Control in der eigenen Applikation verwendet werden, muß dieses Kontrollkästchen aktiviert werden, was schon standardmäßig erfolgt. Durch diesen Punkt wird in den Quelltext eine zusätzliche Header-Datei eingefügt, die den Quelltext aufbläht und den Erstellungsvorgang verlangsamt. Sind Sie sicher, daß Sie keines dieser Steuerelemente benötigen, deaktivieren Sie dieses Kontrollkästchen.

Sollte sich herausstellen, daß Sie doch ActiveX-Controls benötigen, müssen Sie nicht etwa das Projekt neu anlegen. Es muß nur folgende Zeile in die Datei STDAFX.H eingetragen werden:

```
#include <afxdisp.h>
```



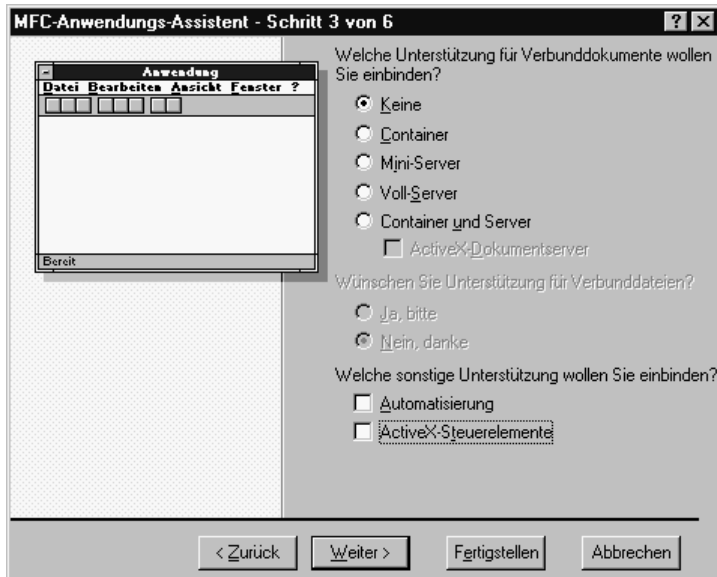


Abbildung 3.3: Schritt 3 des Anwendungs-Assistenten

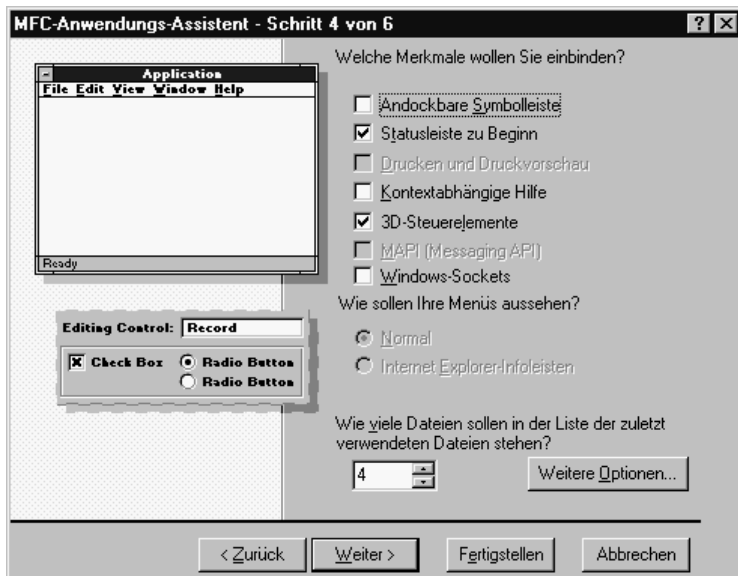


Abbildung 3.4: Schritt 4 des Anwendungs-Assistenten

Im vierten Schritt werden das Aussehen und der Inhalt der Applikation genauer bestimmt (Abbildung 3.4).

- ▼ *Andockbare Symbolleiste (Docking toolbar)*: Vom Anwendungs-Assistenten wird eine Symbolleiste erzeugt, die Buttons für Anlegen, Öffnen und Speichern eines Dokuments, Ausschneiden/Kopieren/Einfügen, Druck und Hilfe enthält. Die Symbolleiste kann mit Hilfe der Maus an eine andere Position innerhalb der Applikation bewegt werden (wie beispielsweise in Winword).
- ▼ *Statusleiste zu Beginn (Initial status bar)*: Die Anwendung erhält eine Statuszeile. Diese enthält standardmäßig Anzeigen für Num Lock, Caps Lock und Scroll Lock sowie ein Feld zur Anzeige der Hilfetexte, wenn der Cursor im Menü bewegt wird. Wenn diese Option aktiviert ist, wird ein zusätzlicher Menüpunkt eingefügt, der es erlaubt, Symbolleiste und Statuszeile auszublenden.
- ▼ *Drucken und Druckvorschau (Printing and print preview)*: Es wird zusätzlicher Code eingebunden, der das Drucken (über die Klasse CView) und eine Druckvorschau ermöglicht. Zum Aufruf werden die entsprechenden Menüpunkte in das DATEI-Menü eingebunden.
- ▼ *Kontextabhängige Hilfe (Context-sensitive Help)*: Der Punkt HILFE wird in das Menü eingefügt und ermöglicht den Aufbau einer kontextsensitiven Hilfe. Dabei ist der Hilfetext für alle vom Anwendungs-Assistenten generierten Menüs bereits eingetragen.
- ▼ *3D-Steuerelemente (3D controls)*: Gibt an, daß für Dialogboxen und Ansichten die dreidimensionalen Effekte (Controls) verwendet werden sollen.
- ▼ *MAPI (Messaging API)*: Mit Hilfe dieser API ist es möglich, Mail-Nachrichten zu erstellen und zu versenden.
- ▼ *Windows Sockets*: Windows Sockets ermöglichen die Nutzung des TCP/IP-Netzwerkprotokolls in der Anwendung.
- ▼ *Aussehen der Menüs*: Hierbei geht es eigentlich um das Aussehen der Symbolleisten – offensichtlich ein Fehler beim Übersetzen. Die Auswahl wird auch erst dann möglich, wenn Symbolleisten aktiviert sind. Diese können das traditionelle Aussehen haben oder an den Internet Explorer 4 angelehnt sein. Die zweite Variante ermöglicht es, auf einfache Weise weitere Standard-Steuerelemente in die Symbolleiste einzubinden.

Der Punkt *Wie viele Dateien sollen in der Liste der zuletzt verwendeten Dateien stehen?* ermöglicht es, die Einstellung der Zahl der Einträge der zuletzt geöffneten Dateien zu ändern. In Winword kann diese Zahl auch in EXTRAS|OPTIONEN geändert werden.



In Schritt 4 des Anwendungs-Assistenten dürfen nicht alle Optionen deaktiviert werden, da es sonst vorkommen kann, daß die Anwendung mit einer Schutzverletzung schon beim Start beendet wird. Leider ließ sich nicht genau lokalisieren, wann das der Fall ist. Hier gilt es, auf ein Service Pack von Microsoft zu warten, das den Fehler behebt.

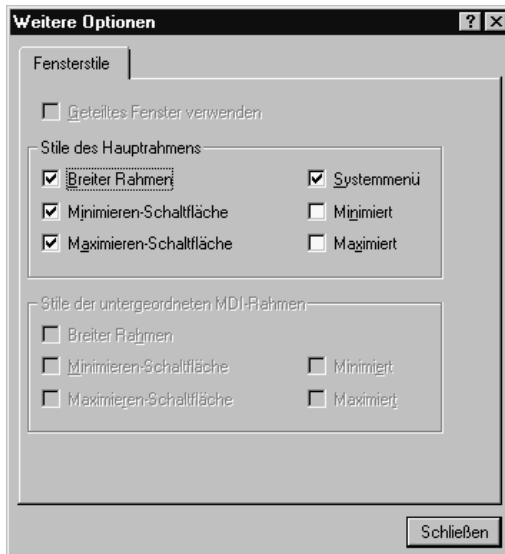


Abbildung 3.5: Einstellen weiterer Optionen in Schritt 4

Die Schaltfläche WEITERE OPTIONEN bietet noch weitere Einstellungen, zum Beispiel, ob ein System-Menü vorhanden sein, soll oder ob das Fenster beim Start der Applikation maximiert angezeigt werden soll (siehe Abbildung 3.5).



Stellen Sie für das Beispielprogramm alle Optionen so ein, wie in Abbildung 3.4 dargestellt.

In Schritt 5 wird festgelegt, ob Kommentare in den Quelltext eingefügt werden sollen (diese beginnen meist mit ZU ERLEDIGEN: (TO DO:)) und beschreiben, an welcher Stelle eigener Programmcode eingefügt werden kann) und wie die MFC-Klassenbibliothek verwendet werden soll: entweder als DLL oder als Bestandteil des Programms. Die Verwendung der DLL verkürzt den Link-Vorgang und verkleinert das Programm. Allerdings muß sichergestellt sein, daß die MFC-DLL auf dem Zielsystem vorhanden ist oder mitgeliefert wird. Des weiteren legen Sie hier fest, ob Sie eine Anwendung im Explorer-Stil oder eine Anwendung im herkömmlichen Windows-Stil erzeugen wollen (siehe Abbildung 3.6).

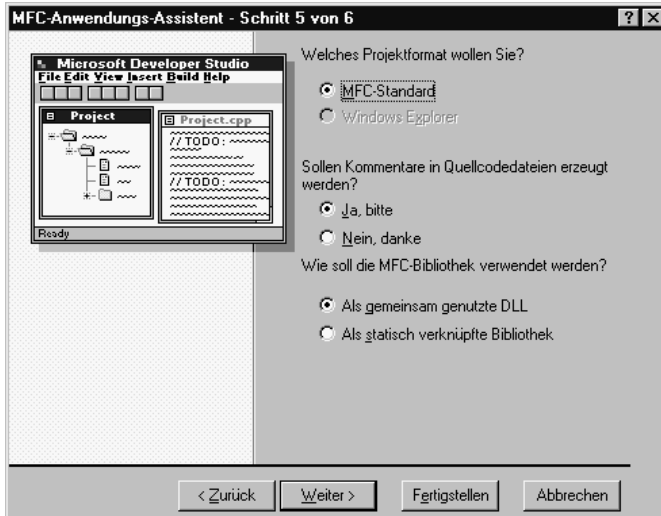


Abbildung 3.6: Schritt 5 des Anwendungs-Assistenten

Der sechste und letzte Schritt zeigt alle Klassen-Namen und Dateinamen an, die vom Anwendungs-Assistenten erzeugt werden. An dieser Stelle können noch Änderungen der Dateinamen und, bei SDI- und MDI-Anwendung mit Dokument-Ansicht-Architektur, der Ansichts-Basisklasse vorgenommen werden (siehe Abbildung 3.7). Abbildung 3.8 zeigt die geöffnete Liste zur Wahl der Ansichtsklasse.

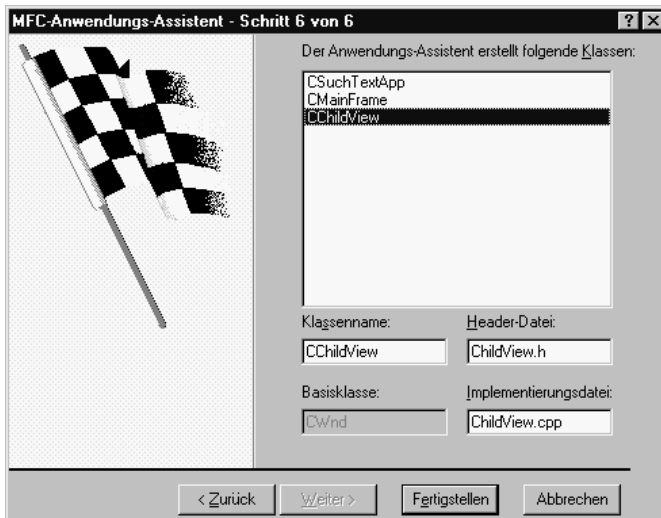


Abbildung 3.7: Schritt 6 des Anwendungs-Assistenten

Jetzt kann nur noch zurückgegangen oder mit FERTIGSTELLEN das Projekt angelegt werden. In der folgenden Dialogbox wird noch einmal ein Report angezeigt, welche Klassen und Dateien denn nun erzeugt werden. Sobald der OK-Button gedrückt wird, werden die Dateien generiert, und der neue Arbeitsbereich wird geöffnet.



Legen Sie zum Erlernen des Umgangs mit dem Anwendungs-Assistenten weitere Projekte an, und variieren Sie in den Einstellungen. Interessant sind vor allem die Möglichkeiten in Schritt 3. Damit können wichtige Teile einer Windows-Anwendung implementiert werden, ohne eine einzige Zeile Quelltext zu schreiben. Erzeugen Sie auch einmal eine MDI-Anwendung, und vergleichen Sie die Unterschiede bezüglich ihres Aussehens. (Auf SDI und MDI wird in weiteren Abschnitten noch genauer eingegangen.)



Abbildung 3.8: Auswahl der Ansichtsklasse einer SDI-Anwendung



Schreiben Sie einen eigenen Texteditor ohne eine Zeile Quelltext. Unter Verwendung der Dokument-Ansicht-Architektur in Schritt 1 und der Basisklasse der Ansicht als *CEditView* in Schritt 6 wird das problemlos erreicht. Wenn Sie Druck und Seitenansicht aktiviert haben, können Sie die verfaßten Texte auch sofort ausdrucken.

Wie Sie sehen, bietet der Anwendungs-Assistent eine Vielzahl von Möglichkeiten, um Programme bzw. Programmgerüste automatisch zu erzeugen. Dies bedeutet eine enorme Zeiteinsparung für den Programmierer, der sich nicht mehr mit diesen wiederkehrenden Problemen befassen muß.

3.1.2 Vom Anwendungs-Assistenten erzeugte Dateien

Die Bedeutung und der Inhalt der Dateien, die vom Anwendungs-Assistenten erzeugt werden, sollen im folgenden Abschnitt kurz beschrieben werden.



Noch ein wichtiger Hinweis: Nachdem ein Projekt mit dem Anwendungs-Assistenten generiert wurde, befinden sich an einigen Stellen zusätzliche Kommentare im Programm. Diese dürfen weder entfernt noch verändert werden. Auch die durch die Kommentare eingeschlossenen Programmzei-

len dürfen nicht verändert werden. Es handelt sich hierbei um Erkennungszeichen des *Klassen-Assistenten*, der Klassen und Member-Funktionen erzeugen kann.

```
BEGIN_MESSAGE_MAP(CSuchTextApp, CWinApp)
    //({AFX_MSG_MAP(CSuchTextApp)
    ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
    // HINWEIS - Hier werden Mapping-Makros vom
    Klassen-Assistenten eingefügt und entfernt.
    // Innerhalb dieser generierten
    Quelltextabschnitte NICHTS VERÄNDERN!
    //})AFX_MSG_MAP
END_MESSAGE_MAP()
```

In Tabelle 3.1 sind die vom Anwendungs-Assistenten erzeugten Dateien sowie deren Bedeutung für das Programmgerüst aufgeführt. Für eine genaue Erklärung der Arbeitsweise von Dokument und Ansicht sei auf einen späteren Abschnitt dieses Buches verwiesen.

Dateiname (.H, .CPP)	Inhalt
MainFrm.h MainFrm.cpp	Enthält die von CFrameWnd abgeleitete Klasse, die zur Erzeugung des Hauptfensters dient.
SuchText.h SuchText.cpp	In diesen Dateien ist die von CWinApp abgeleitete Applikationsklasse enthalten. Zusätzlich ist noch der »Info über ...« -Dialog integriert.
SuchText.rc Resource.h	Dies ist die Ressource-Datei zum Projekt mit allen bereits vorhandenen wichtigen Ressourcen wie Menü, Icon, Beschleuniger-Tasten, Info-Dialog und Zeichenketten. In Resource.h sind die zugehörigen symbolischen Konstanten definiert. Weitere Ressourcen (Symbol, Bitmap ...) sind im Unterverzeichnis RES abgelegt.
SuchTextDoc.h SuchTextDoc.cpp	Das Dokument mit der von CDocument abgeleiteten Klasse (nur bei Verwendung der Dokument-Ansicht-Architektur).
ChildView.h ChildView.cpp	Die Ansichts-Klasse, die im Hauptfenster liegt.
StdAfx.h StdAfx.cpp	Enthält die Anweisungen zum Einbinden der Header-Dateien der MFC. Aus diesen wird die vorkompilierte Header-Datei <i>SuchText.pch</i> angelegt, um den Erstellungsvorgang zu beschleunigen.
SuchText.clw	Binärdatei mit Informationen für den Klassen-Assistenten.
ReadMe.txt	Eine Textdatei mit Informationen zu den angelegten Dateien ähnlich wie in dieser Tabelle beschrieben.

Tabelle 3.1: Vom Anwendungs-Assistenten erzeugte Dateien

Zusätzlich generiert der Anwendungs-Assistent auch noch die Projektdatensatz-Datei Make-File (.DSP) und den Arbeitsbereich (.DSW). Wird die Dokument-Ansicht-Architektur benutzt, wird die Ansichts-Klasse in einer eigenen Datei (SuchTextView.*) abgelegt.

Die vom Anwendungs-Assistenten angelegten Dateien und Klassen stellen das Programmgerüst dar. Somit steht dem Übersetzen und Starten der neuen Windows-Applikation nichts mehr im Wege.

3.1.3 Test des neuen Programms

Nachdem das Projekt erzeugt wurde, kann es sofort übersetzt und gestartet werden. Dazu muß nur die Schaltfläche ERSTELLEN (Taste **[F7]** oder Menü ERSTELLEN|XX.EXE ERSTELLEN) gedrückt werden. Zuvor sollte noch entschieden werden, ob die Debug- oder die Release-Version erzeugt werden soll. Da wir hier noch am Anfang der Entwicklung des Programms stehen und Debugging (siehe Kapitel 8.2) zur Fehlersuche unumgänglich ist, kann die Standardeinstellung (Debug-Version) beibehalten werden.

Nach dem Erstellungs-Vorgang kann die fertige EXE-Datei gestartet werden (**[Strg] [F5]** oder ERSTELLEN|AUSFÜHREN VON XX.EXE). Da für das *Such-Text*-Programm viele Optionen ausgeschaltet wurden, besitzt die Applikation nur ein Menü und stellt eine minimale Windows-Anwendung dar.



Sie können das Programm auch sofort ausführen. Wenn die EXE-Datei nicht existiert oder veraltet ist, wird vom Developer Studio nachgefragt, ob diese vorher neu erstellt werden soll.

Damit ist schon alles zum Anwendungs-Assistenten gesagt. Er ist ein Hilfsmittel zum Anlegen von Applikationen und befreit den Nutzer von Standard-Arbeiten. Mit einem einheitlichen Programmgerüst läßt sich die Struktur eines Programms leichter erkennen. Wichtig für das Verständnis des erzeugten Programmgerüsts ist auch, was sich in welcher Datei befindet. Sehen Sie sich in Ruhe noch einmal alle Quellcode-Dateien an.

3.1.4 Neuerungen in InitInstance

Wie bereits in Kapitel 2.2.1 erläutert, dient die überlagerte Methode *InitInstance* von *CWinApp* zum Anlegen und Anzeigen des Hauptfensters. Dazu wird eine Instanz einer von *CFrameWnd* abgeleiteten Klasse angelegt. Der Anwendungs-Assistent legt diese Klasse mit der Bezeichnung *CMainFrame* in den Dateien *MainFrm.h* und *MainFrm.cpp* an. Die Methode *InitInstance* ist in *SuchText.cpp* implementiert.

```
BOOL CSuchTextApp::InitInstance()
{
    // Standardinitialisierung
    // Wenn Sie diese Funktionen nicht nutzen und
    // die Größe Ihrer fertigen
    // ausführbaren Datei reduzieren wollen,
    // sollten Sie die nachfolgenden
    // spezifischen Initialisierungsroutinen, die Sie
    // nicht benötigen, entfernen.
```



```

#ifdef _AFXDLL
    Enable3dControls(); // Diese Funktion bei
                        // Verwendung von MFC in gemeinsam genutzten
                        // DLLs aufrufen
#else
    Enable3dControlsStatic(); // Diese Funktion bei statischen
                             // MFC-Anbindungen aufrufen
#endif
// Ändern des Registrierungsschlüssels, unter dem unsere
// Einstellungen gespeichert sind.
// ZU ERLEDIGEN: Sie sollten dieser Zeichenfolge einen
// geeigneten Inhalt geben
// wie z.B. den Namen Ihrer Firma oder Organisation.
SetRegistryKey(_T("Local AppWizard-Generated Applications"));
// Dieser Code erstellt ein neues Rahmenfensterobjekt und setzt
dies
// dann als das Hauptfensterobjekt der Anwendung, um das
Hauptfenster
// zu erstellen.
CMainFrame* pFrame = new CMainFrame;
m_pMainWnd = pFrame;
// Rahmen mit Ressourcen erstellen und laden
pFrame->LoadFrame(IDR_MAINFRAME,
                 WS_OVERLAPPEDWINDOW | FWS_ADDTOTITLE, NULL, NULL);
// Das einzige Fenster ist initialisiert und kann jetzt angezeigt
und
// aktualisiert werden.
pFrame->ShowWindow(SW_SHOW);
pFrame->UpdateWindow();
return TRUE;
}

```

Listing: InitInstance in SuchText.cpp

Am Anfang ist ein Hinweis enthalten, der dazu auffordert, alle nicht benötigten Teile zu entfernen, um die Größe der EXE-Datei zu verringern. Der Aufruf von *Enable3dControls* schaltet die dreidimensionale Darstellung der Steuerungen von Dialogen ein. Diese Zeilen sind nur vorhanden, wenn der Punkt *3D Steuerelemente* im Anwendungs-Assistenten, Schritt 4, markiert wurde.

Die Funktion *LoadStdProfileSettings()* (nur bei Verwendung der Dokument-Ansicht-Architektur enthalten) dient zum Laden der Liste der zuletzt bearbeiteten Dateien (MRU – most recently used files) aus der Standard-INI-Datei der Applikation.

```
[Recent File List]
File1=C:\MSC\SuchText\Resource.h
File2=C:\MSC\SuchText\Browse.cpp
```

Listing: Ausschnitt aus der INI-Datei

Wenn Sie mit der Registrierungsdatei arbeiten wollen, sollten Sie den Aufruf von *SetRegistryKey* beachten. Dabei wird ein neuer Schlüssel in die Registrierung eingetragen. Der angegebene String sollte der Name der Firma oder etwas ähnlich Treffendes sein. Hier muß nicht etwa der Programmname eingetragen werden. Abgelegt werden die Einträge dann unter HKEY_CURRENT_USER\Software\<Firma>\<Programm>\<Abschnitt>\<Wert>. Wurde die Funktion aufgerufen, werden alle Aufrufe von *Get/SetProfileString* und *Get/SetProfileInt* nicht mehr mit der INI-Datei abgewickelt, sondern an die Registrierung weitergeleitet (Abbildung 3.9).

Diese Liste wird in das DATEI-Menü eingeblendet. Die Anzahl der Dateien ist standardmäßig auf vier festgelegt. Wenn die Anzahl davon abweicht (vgl. Anwendungs-Assistent, Schritt 4), dann wird hier die Zahl angegeben. Null wiederum bedeutet, daß diese Liste nicht geladen wird. Dieses Beispiel zeigt, wie die Arbeit des Programmiers durch die MFC erleichtert wird.

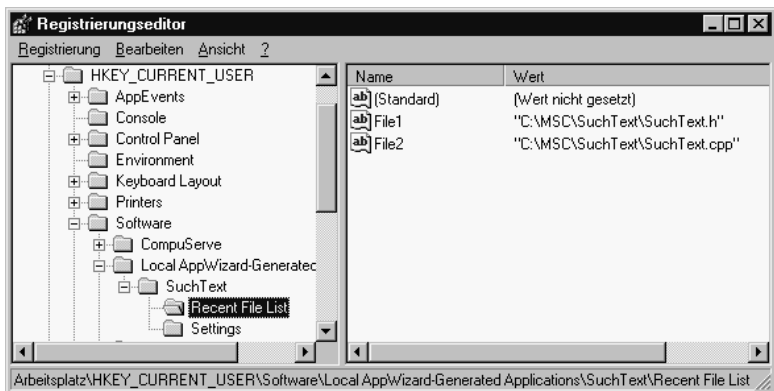


Abbildung 3.9: Eintragen von Programm-Einstellungen in die Registrierung

Die Klasse *CMainFrame*, die von *CFrameWnd* abgeleitet ist, bildet das Hauptfenster der Anwendung. Zuerst wird eine Instanz von *CMainFrame* erzeugt und der Zeiger dem Hauptfenster der Anwendung zugewiesen. In der Referenz zur Klassenbibliothek sind zur Klasse *CFrameWnd* drei Wege zum Erzeugen des Hauptfensters beschrieben: Der erste führt über die bekannte Funktion *Create*, der zweite über *LoadFrame* und der dritte Weg über sogenannte Document Templates.

Die Anwendung von Document Templates wird im Kapitel 12.4 genauer erläutert. Die Ressource-ID (IDR_MAINFRAME) wird beim Laden des Menüs, des Programmsymbols, der Beschleunigertasten und einer String-Tabelle benutzt.

Die weiteren Zeilen in *InitInstance* sind bekannt. Das Hauptfenster wird angezeigt und mit *UpdateWindow* aktualisiert. *UpdateWindow* sendet die WM_PAINT Nachricht.

3.1.5 Initialisierungen in *CMainFrame*

Im vorhergehenden Kapitel wurde dargestellt, daß *CMainFrame*, eine von *CFrameWnd* abgeleitete Klasse, das Hauptfenster der Anwendung bildet. Mit Hilfe von *LoadFrame* wird das Fenster dynamisch erzeugt. Bevor das erfolgt, werden noch zwei Funktionen der Klasse *CMainFrame* aufgerufen: *PreCreateWindow* und *OnCreate*. *PreCreateWindow* wird kurz vor dem Anzeigen des Fensters aufgerufen und dient zum Setzen von Styles sowie zum Registrieren des Fensters.

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    if( !CFrameWnd::PreCreateWindow(cs) )
        return FALSE;
    // ZU ERLEDIGEN: Ändern Sie hier die Fensterklasse oder das
    // Erscheinungsbild, indem Sie CREATESTRUCT cs modifizieren.
    cs.dwExStyle &= ~WS_EX_CLIENTEDGE;
    cs.lpszClass = AfxRegisterWndClass(0);
    return TRUE;
}
```

Listing: *PreCreateWindow* in *CMainFrame*

Interessanter ist die Funktion *CMainFrame::OnCreate*. In ihr erfolgen die Initialisierungen für das Hauptrahmenfenster und das Einbetten des Kindfensters der Klasse *CChildView*.

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;
    // create a view to occupy the client area of the frame
    if (!m_wndView.Create(NULL, NULL, AFX_WS_DEFAULT_VIEW,
        CRect(0, 0, 0, 0), this, AFX_IDW_PANE_FIRST, NULL))
    {
        TRACE0("Failed to create view window\n");
        return -1;
    }
}
```

```

    if (!m_wndStatusBar.Create(this) ||
        !m_wndStatusBar.SetIndicators(indicators,
            sizeof(indicators)/sizeof(UINT)))
    {
        TRACE0("Statusleiste konnte nicht erstellt werden\n");
        return -1;        // Fehler bei Erstellung
    }
    return 0;
}

```

Listing: Initialisierungen in der Methode OnCreate

Die Member-Variable `m_wndView` ist eine Instanz der Klasse `CChildView`. `CChildView` wiederum ist von `CWnd` abgeleitet. In der Methode `CChildView::PreCreateWindow` wurden dem Fenster einige Eigenschaften zugewiesen und die Registrierung durchgeführt. Dieses Fenster wird am Anfang der Methode `OnCreate` mit `m_wndView.Create` erzeugt. Die Größe wird dabei so angepaßt, daß es das Hauptrahmenfenster vollständig ausfüllt. In diesem Fenster erfolgen später auch die Ausgaben des Programms. Dies erfolgt in der Klasse `CChildView` in der Methode `OnPaint`. Die Erläuterungen dazu finden Sie in den folgenden Kapiteln.

Des weiteren wird in `OnCreate` noch die Statuszeile erzeugt. Je nach den Einstellungen des Anwendungs-Assistenten können hier weitere Initialisierungen, wie zum Beispiel das Anlegen einer Symbolleiste, erfolgen.

3.2 Der Klassen-Assistent

3.2.1 Anlegen von Klassen, Member-Funktionen und Member-Variablen

Bisher wurden die Einträge der Message-Map oder Member-Variablen der Klassen durch Editieren des Quelltextes von Hand vorgenommen. Dabei mußte besonders bei Funktionen, die über die Message-Map aufgerufen werden, die Funktion zusätzlich angelegt werden. Um auch hier eine Arbeitserleichterung zu erreichen, existieren im Developer Studio einige Möglichkeiten; wichtigster Bestandteil ist wiederum ein Assistent, der *Klassen-Assistent*.

Der Klassen-Assistent erzeugt Prototypen von Klassen und Funktionen, legt Member-Variablen an und stellt die Verknüpfung zwischen Ressource und Quelltext her. Wohlgemerkt, es werden nur die Funktionsrümpfe erzeugt – die Funktionen zum Leben zu erwecken, ist weiterhin Sache des Programmierers.

Der Aufruf des Klassen-Assistenten erfolgt über das Menü **ANSICHT|KLASSEN-ASSISTENT** oder mit der Tastenkombination **[Strg][W]**. Damit wird das in Abbildung 3.10 dargestellte Fenster auf den Bildschirm gebracht.

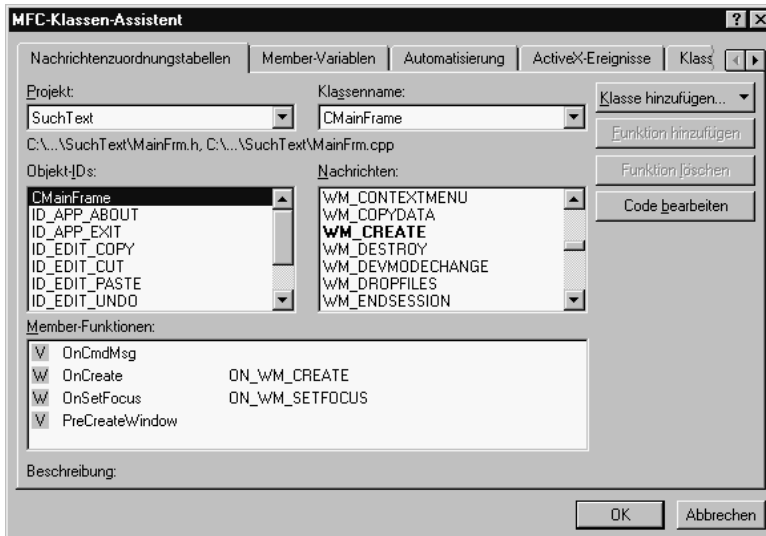


Abbildung 3.10: Der Klassen-Assistent

Der Klassen-Assistent ist wie ein Karteikasten mit Reitern aufgebaut. Jede der Seiten hat spezifische Aufgaben.

Die Seite *Nachrichtenzuordnungstabellen* dient zur Verknüpfung von Windows-Nachrichten mit Methoden der Klassen. Folgende Arbeitsschritte sind notwendig:

1. Im Feld *Klassenname* wird die Klasse ausgewählt, in der die Nachricht behandelt werden soll. Voreingestellt ist immer die Klasse der im Editor aktiven Datei.
2. In der Liste der *Objekt IDs* ist entweder die entsprechende ID oder der Name der Klasse zu wählen, abhängig davon, ob die WM_COMMAND-Nachrichten von Ressourcen (z.B. Menüs) abgefangen werden sollen oder ob eine andere Nachricht der Klasse (z.B. WM_PAINT) behandelt werden soll.
3. In der Listbox *Nachrichten* ist die gewünschte Nachricht auszuwählen. Es werden immer nur diejenigen Nachrichten angezeigt, die in der jeweiligen Klasse oder über die ID zu handhaben sind. Für Resource-IDs sind das immer nur COMMAND- und UPDATE_COMMAND_UI-Nachrichten. Für Klassen kann ggf. noch der *Filter für Nachrichten* der Klasse auf der Seite *Klassen-Info* geändert werden.
4. Jetzt kann die Schaltfläche FUNKTION HINZUFÜGEN gedrückt werden, mit dem der Funktionsrumpf und die Deklaration in den Quelltext eingefügt werden.

5. Weitere Funktionen können sofort angelegt werden. Ist das nicht nötig, kann durch Drücken von CODE BEARBEITEN direkt zu der neu eingefügten Funktion im Quelltext gesprungen werden.

Der Button FUNKTION LÖSCHEN dient dazu, eine Zuweisung einer Nachricht wieder aufzuheben. Dabei wird vom Klassen-Assistenten zwar der Eintrag in der Message-Map und die Deklaration im Header-File entfernt, für das Löschen der Funktion ist der Programmierer jedoch selbst zuständig. Das hat den Vorteil, daß die Implementation einer Funktion nicht verlorengeht.

An dieser Stelle soll noch einmal darauf hingewiesen werden, daß innerhalb der Kommentare des Klassen-Assistenten keine manuelle Bearbeitung erfolgen darf und diese Kommentare nicht gelöscht werden dürfen, da dies die Arbeit mit dem Klassen-Assistenten unmöglich macht!

Der Klassen-Assistent erlaubt es auch, neue Klassen anzulegen (Schaltfläche KLASSE HINZUFÜGEN|NEU). Darauf wird im Verlauf des Buches an entsprechender Stelle nochmals eingegangen.

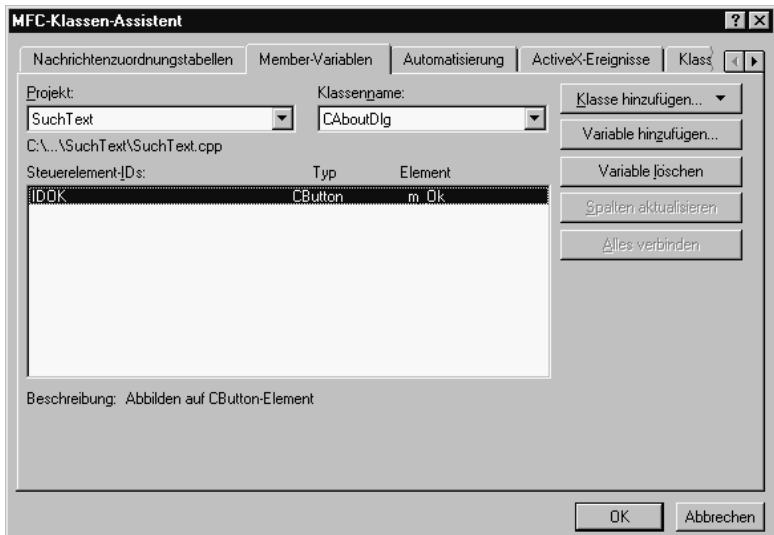


Abbildung 3.11: Member-Variablen im Klassen-Assistenten

In Abbildung 3.11 ist die Dialogseite zum Anlegen von *Member-Variablen* dargestellt. Auf dieser Seite werden zu Elementen eines Dialogs (z.B. zu einer Schaltfläche) die entsprechenden Variablen angelegt. Dabei kann unter *Kategorie* noch unterschieden werden, ob eine einfache Variable oder die Instanz der Klasse der entsprechenden Steuerung angelegt wird (siehe

Abbildung 3.12). Eine Instanz der Klasse hat den Vorteil, daß die Steuerung zur Laufzeit verändert werden kann (z.B., indem eine Schaltfläche deaktiviert wird). In den Feldern *Kategorie* und *Variablentyp* werden kontextabhängig nur die möglichen Elemente angezeigt.

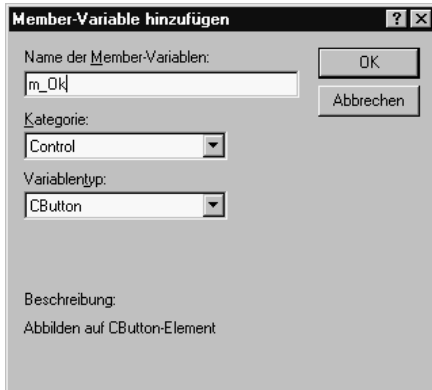


Abbildung 3.12: Anlegen einer neuen Member-Variablen

Auf die OLE-spezifischen Einstellungsmöglichkeiten soll in diesem Buch nicht eingegangen werden.

Auf der Register-Seite *Klassen-Info* können noch einmal Informationen zu einer Klasse angezeigt werden. Dazu gehören beispielsweise die Dateinamen der Header- und Implementierungsdatei sowie der Name der Basis-Klasse (Abbildung 3.13). Unter *Weitere Optionen* kann noch der Filter für die Nachrichten der Klasse eingestellt werden. Dieser wird entsprechend dem Typ der Klasse voreingestellt. Sollte es erforderlich sein, eine Nachricht zu behandeln, die nicht in der Liste enthalten ist, kann der Filter geändert werden, so daß zusätzliche Nachrichten beispielsweise von übergeordneten Fenstern angezeigt werden.

Der Aufruf des Klassen-Assistenten kann auch direkt nach dem Anlegen einer Ressource, zum Beispiel eines Menüpunktes, erfolgen. Dann ist auf seiten der Nachrichtenzuordnungstabelle der entsprechende Eintrag der ID bereits markiert.

3.2.2 Assistentenleiste

Eine vereinfachte, aber nicht weniger leistungsfähige Form des Klassen-Assistenten ist dessen Assistentenleiste (Abbildung 3.14). Das Ein- und Ausblenden kann über das Kontextmenü für Symbolleisten oder über EXTRAS|ANPASSEN|SYMBOLLEISTEN erfolgen. Der Inhalt der Leiste wird der aktuellen Cursorposition im Editorfenster angepaßt.

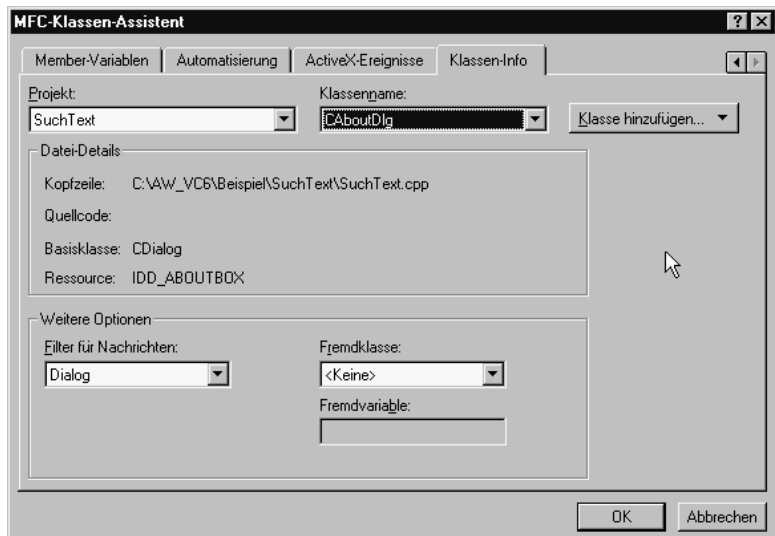


Abbildung 3.13: Informationen zu den Klassen anzeigen



Abbildung 3.14: Die Assistentenleiste

In der linken Drop-down-Liste sind alle C++-Klassen des Projekts enthalten, in der mittleren Liste kann ein Filter eingestellt werden, und rechts werden alle C++-Elemente dem Filter entsprechend angezeigt. So kann man durch Auswahl von Klasse und Element direkt zu einer Methode springen. In der Filter-Liste sind alle der Klasse bekannten Ressource-IDs enthalten. Wird eine ID ausgewählt, werden in der Element-Liste die zur jeweiligen ID möglichen Nachrichten angezeigt. Ist bereits eine Message-Handler-Routine geschrieben, so wird die Message fett dargestellt und bei deren Selektion zur entsprechenden Stelle im Quelltext gesprungen. Andernfalls kann für die gewählte Message eine neue Funktion angelegt werden. Umgekehrt wird die Assistentenleiste beim Blättern im Quelltext immer aktualisiert. Im rechten Teil der Symbolleiste befinden sich noch eine Schaltfläche und eine Drop-down-Liste. Die Schaltfläche, die ein veränderliches Symbol ihrem Zustand entsprechend enthält (siehe Tabelle 3.2), dient zum Ausführen einer Standard-Aktion. Soll eine andere Aktion ausgelöst werden, kann diese über die Liste ausgewählt werden (siehe Abbildung 3.15). Die Standard-Aktion ist fett dargestellt und ebenfalls kontextsensitiv, d.h., sie ändert die Aktion der eingestellten Klasse, dem Filter und dem Element entsprechend. Auch die Menüpunkte passen sich dem Kontext an.

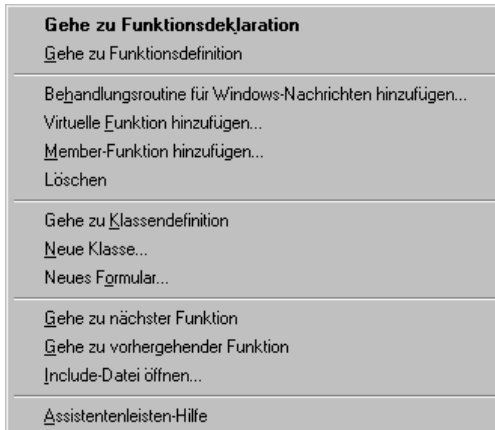


Abbildung 3.15: Aktionen der Assistentenleiste

Wie man sieht, lässt sich auf diese Weise schnell auf eine Reihe von Funktionen zugreifen, die sonst nur über den Klassen-Assistenten oder den Arbeitsbereich erreichbar sind. Damit wurde eine weitere einfache Bedienungsmöglichkeit der Entwicklungsumgebung geschaffen, an die man sich im Laufe der Zeit schnell gewöhnen kann.

Symbol	Bedeutung
	Assistentenleiste ist aktiv und überwacht aktuelle Cursorposition.
	Assistentenleiste ist aktiv, kann aber Cursorposition nicht überwachen (Sie befinden sich z. B. gerade in der Online-Hilfe).
	Assistentenleiste ist nicht aktiv.

Tabelle 3.2: Symbolbedeutung der Schaltfläche STANDARDAKTION

3.2.3 Anlegen von Funktionen und Variablen über die Klassen-Ansicht

Wenn es darum geht, Member-Variablen in der Header-Datei einer Klasse anzulegen oder Member-Funktionen zu erstellen, die nicht mit Nachrichten verbunden werden, hilft das Kontextmenü des Klassen-Assistenten im *Arbeitsbereich*-Fenster (Abbildung 3.16). Wird die rechte Maustaste über dem Namen einer Klasse gedrückt, kann man eine Reihe von Funktionen ausführen lassen:

- ▼ Anlegen von Member-Funktionen, Member-Variablen und virtuellen Funktionen zur Klasse
- ▼ Anlegen von Funktionen für Nachrichten (Abbildung 3.18)

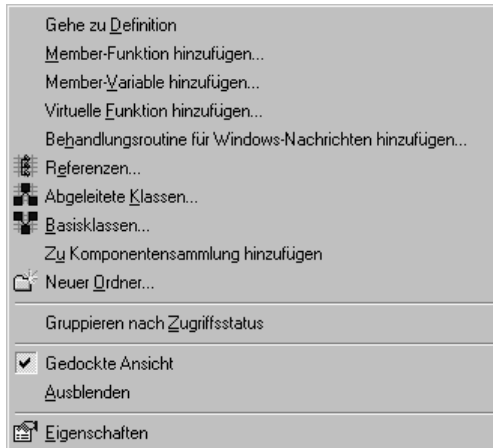


Abbildung 3.16: Kontextmenü einer Klasse im ClassView

- ▼ Funktionen des Klassen-Browsers wie zum Beispiel die *Referenzen* einer Funktion anzeigen
- ▼ Hinzufügen einer Klasse als neues Modul zur Komponentensammlung

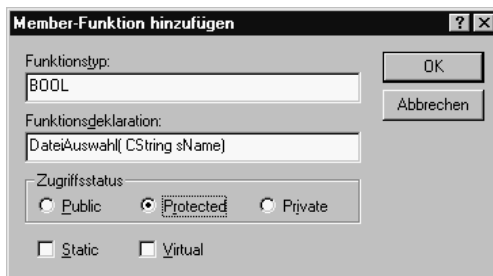


Abbildung 3.17: Hinzufügen von Member-Funktionen

Abbildung 3.17 zeigt den Dialog zum Hinzufügen einer Member-Funktion. Es müssen der Typ der Funktion sowie deren Name angegeben werden. Zusätzlich können die Attribute *virtual* und *static* ergänzend deklariert werden. Über *Zugriffsstatus* wird der Sichtbarkeitsbereich der Funktion festgelegt.

Ähnlich verhält es sich bei Variablen. Dabei müssen Typ und Name sowie der Sichtbarkeitsbereich in der Klasse angegeben werden.

Das Fenster zum Hinzufügen von Nachrichtenbehandlungsfunktionen ist quasi ein Ausschnitt aus dem Klassen-Assistenten. Für die ausgewählte Klasse werden entweder Routinen für die gesamte Klasse oder für einzelne

Objekte, deren ID angezeigt wird, angelegt. Je nach Auswahl werden die möglichen Nachrichten in eine Liste aufgenommen. Durch Auswahl einer Nachricht aus dieser Liste kann über die Schaltfläche BEHANDLUNGSRoutine HINZUFÜGEN eine Funktion in der Nachrichtentabelle erzeugt werden.

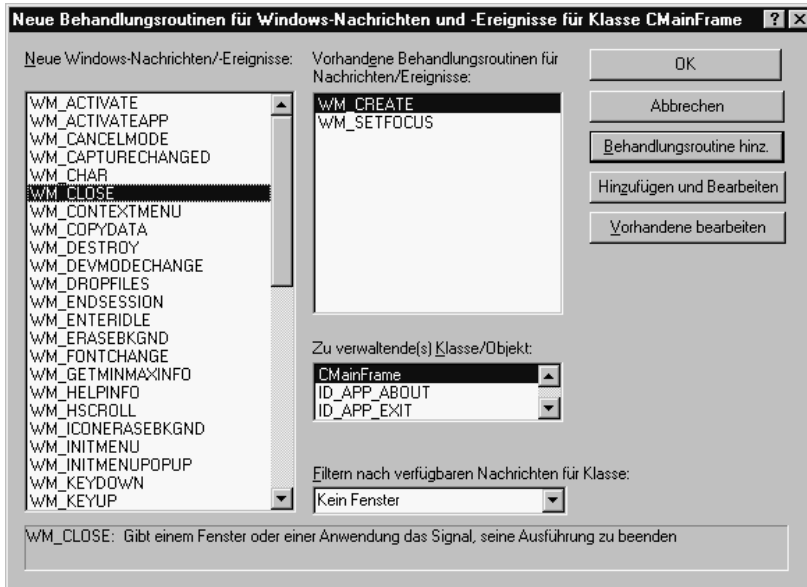


Abbildung 3.18: Hinzufügen von Nachrichtenbehandlungsroutinen

HINZUFÜGEN UND BEARBEITEN fügt die neue Funktion ein und verzweigt sofort zum Quelltext, um sie bearbeiten zu können. Über VORHANDENE BEARBEITEN wird zum Quelltext der Routine gesprungen, die in der Liste der vorhandenen Zuordnungen selektiert worden ist. Des weiteren ist es möglich, für die angezeigten Nachrichten einen Filter zu wählen.



Ressourcen

4

Kapitelüberblick

4.1	Menüs	142
4.1.1	Das Hauptmenü	142
4.1.2	Bedienung des Menü-Editors	143
4.2	Verändern von Menüpunkten	153
4.2.1	Die Klasse CMenu	154
4.2.2	Temporäre Objekte	155
4.2.3	Menüpunkte aktivieren und deaktivieren	156
4.2.4	Menüpunkte markieren	159
4.3	Dynamisch erzeugte Menüs	161
4.3.1	Menüpunkte zur Laufzeit löschen oder einfügen	161
4.3.2	Erzeugen eines Kontext-Menüs	165
4.3.3	Verändern des Systemmenüs	169
4.4	Beschleunigertasten	171
4.4.1	Anlegen einer Beschleuniger-Ressource	172
4.4.2	Integration einer Beschleuniger-Ressource	173
4.4.3	Anlegen versteckter Beschleuniger	174

Die Ressourcen eines Windows-Programms sind die optische Schnittstelle zum Benutzer des Programms. Deshalb ist es wichtig, diese entsprechend den »Standards« so zu entwerfen, daß sich der Benutzer leicht zurechtfindet. Voraussetzung zum Entwurf der Benutzerschnittstelle ist ein komfortabler Ressourcen-Editor. Dieser ist in das Developer Studio bereits integriert – Sie müssen kein gesondertes Programm aufrufen. Ein Doppelklick mit der Maus auf eine Ressource auf der Ressourcen-Seite des *Arbeitsbereich*-Fensters genügt, um die Ressource im entsprechenden Editor für die Bearbeitung bereitzustellen.

4.1 Menüs

4.1.1 Das Hauptmenü

Nahezu jede Windows-Anwendung besitzt ein Hauptmenü. Dies ist nicht nur eine allgemein übliche Konvention, sondern vor allem deshalb sehr sinnvoll, weil es dem ungeübten Anwender den Umgang mit dem Programm erleichtert. Menüs sind – ebenso wie beispielsweise Symbole – Ressourcen und werden daher über die *RC-Datei* des Programms eingebunden. Während es bei der SDK-Programmierung noch erforderlich war, die Menü-Ressource per Hand zu editieren, ist dies in Visual C++ nicht mehr nötig, denn der in das Developer Studio integrierte Menü-Editor ermöglicht das Bearbeiten von Menüs.

Nicht immer ist es angebracht, ein Menü als Ressource einzubinden. Da der Aufbau des Menüs in diesem Fall bereits zur Compile-Zeit feststehen muß, ist eine solche Vorgehensweise manchmal nicht flexibel genug. Glücklicherweise gibt es neben der Möglichkeit, Menüs als Ressource einzubinden, auch die Möglichkeit, sie mit Hilfe der Klasse *CMenu* zur Laufzeit zu bearbeiten, und sie über Methoden dieser Klasse zu erzeugen, anzuzeigen oder zu verändern.

Die Beschreibung der Klasse *CMenu* wird im Abschnitt »Dynamisch erzeugte Menüs« behandelt. Die nachfolgenden Ausführungen beschäftigen sich zunächst mit dem Erzeugen von Menü-Ressourcen.

Die Menü-Ressource von *SuchText* ist in der Datei *SuchText.RC* untergebracht und hat nach dem Anlegen mit dem Anwendungs-Assistenten folgenden Aufbau:

```
IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&Datei"
    BEGIN
        MENUITEM "&Beenden",                                ID_APP_EXIT
    END
    POPUP "&Bearbeiten"
```

```

BEGIN
    MENUITEM "&Rückgängig\tStrg+Z",          ID_EDIT_UNDO
    MENUITEM SEPARATOR
    MENUITEM "&Ausschneiden\tStrg+X",        ID_EDIT_CUT
    MENUITEM "&Kopieren\tStrg+C",            ID_EDIT_COPY
    MENUITEM "E&infügen\tStrg+V",           ID_EDIT_PASTE
END
POPUP "&Ansicht"
BEGIN
    MENUITEM "S&tatusleiste",                ID_VIEW_STATUS_BAR
END
POPUP "&?"
BEGIN
    MENUITEM "Inf&o über SuchText...",      ID_APP_ABOUT
END
END

```

Listing: Ausschnitt aus der Ressource-Datei

Um diese Menü-Ressource zu erstellen, gibt es nichts weiter zu tun, als das *SuchText*-Projekt zu erzeugen. In der Ressource-Datei ist die Menü-Struktur bereits enthalten. Sie können sich die RC-Datei mit jedem Texteditor ansehen. Wenn Sie versuchen, die RC-Datei mit dem Developer Studio zu öffnen, wird Ihnen nicht der Quelltext angezeigt, sondern es werden die Ressourcen interpretiert und gemeinsam mit den Bezeichnern in einer Strukturansicht dargestellt.

Sie können die Ressource-Datei auch im Developer Studio als Text öffnen. Dazu müssen Sie beim Öffnen der RC-Datei im Kombinationsfeld *Öffnen als* den Punkt *Text* einstellen.



Die Syntax einer Menü-Ressource ist so leicht zu lesen, daß offensichtlich ist, welche Bedeutung Schlüsselwörter wie CHECKED, POPUP oder GRAYED haben.

4.1.2 Bedienung des Menü-Editors

Anlegen eines neuen Menüs

Um ein neues Menü in der Ressource-Datei anzulegen, muß entweder die Schaltfläche NEUES MENÜ der Ressourcen-Symboleiste gedrückt oder über EINFÜGEN|RESSOURCE der Punkt MENÜ ausgewählt werden. Nun wird das Menü angelegt. Im Editor-Fenster wird der Menü-Editor aktiviert und das noch leere Menü angezeigt. Alternativ kann auch die Tastenkombination Strg 2 benutzt werden.

Standardmäßig hat ein neues Menü den Namen IDR_MENU1. Dieser ist zwar durchaus praktikabel, für das Hauptmenü sollte jedoch der Name IDR_MAINFRAME vergeben werden. Um diesen Bezeichner zuzuweisen, müssen die Eigenschaften des Menüs geändert werden. Falls die Ressourcen-Ansicht im *Arbeitsbereich*-Fenster gewählt ist, kann dies dort erfolgen. Das Eigenschaften-Fenster wird entweder über das Kontextmenü von IDR_MENU1 oder mit der Tastenkombination **Alt**  aufgerufen (Abbildung 4.1).

Das Eigenschaften-Fenster dient aber nicht nur dazu, die globalen Eigenschaften des Menüs zu verändern, sondern wird auch später bei den einzelnen Menüpunkten noch gebraucht. Damit es nicht nach jeder Aktion wieder geschlossen wird, kann man es mit dem Pin, der sich links oben befindet, befestigen: Jeder Druck auf den Button schaltet zwischen fest und losgelöst um.



Abbildung 4.1: Eigenschaften des Hauptmenüs

Editieren von Menüpunkten

Ein leeres Menü besteht aus einer symbolisierten Menüleiste mit einem leeren Rahmen in der linken oberen Ecke. In diesen Rahmen kann nun einfach die Beschriftung des Menüpunkts, also beispielsweise »&Schriftart«, eingegeben werden. Nach dem Drücken von  wird darunter ein neuer Rahmen geöffnet, der den ersten Menüpunkt aus dem gerade angelegten Menü symbolisieren soll. Auch hier kann jetzt wieder die gewünschte Beschriftung eingegeben werden.



Ändern Sie das Hauptmenü so, daß folgende Menüstruktur entsteht:

- ▼ Datei
 - ▼ Öffnen
 - ▼ Schließen
 - ▼ Suchen
 - ▼ Beenden

▼ Ansicht

- ▼ Kurze Menüs
- ▼ Anfang
- ▼ Ende
- ▼ Seite vor
- ▼ Seite zurück

▼ Schrift

- ▼ System
- ▼ Ansi
- ▼ SansSerif 10
- ▼ SansSerif 24
- ▼ Invertiert

▼ Farbe

- ▼ Schwarz
- ▼ Weiß
- ▼ Blau
- ▼ Grün

Soll nachträglich ein Menüpunkt geändert werden, wird diese einfach mit den Cursortasten oder der Maus angewählt, und die Änderungen können im *Eigenschaften*-Fenster eingegeben werden. Soll das nächste Menü erfaßt werden, genügt es, den leeren Rahmen rechts neben den bisherigen Menüs anzuwählen und analog vorzugehen. Durch Anfassen und Ziehen können einzelne Menüpunkte oder ganze Menüs an eine andere Position verschoben werden.

Die Beschriftung eines Menüpunkts bestimmt in gewisser Weise sein Aussehen und sein Verhalten. So haben insbesondere die beiden Zeichen »&« und »\t« eine besondere Bedeutung. Wird ein & im Beschriftungstext platziert, bekommt der nachfolgende Buchstabe bei der tatsächlichen Darstellung einen Unterstrich, und das Menü erhält darüber hinaus die Eigenschaft, daß dieser Menüpunkt nach dem Öffnen des Menüs durch Drücken des unterstrichenen Buchstabens aufgerufen werden kann. Wird der Unterstrich nicht auf einen einzelnen Menüpunkt, sondern auf dem Namen des kompletten Menüs angewendet, wie beispielsweise bei »&Datei«, so kann das Menü durch Drücken der Tastenkombination Alt+Buchstabe aufgeklappt werden. Soll der Menüttext ein & enthalten, ist dieses durch && zu kodieren.

Die Bedeutung der Zeichenkombination »\t« ist etwas anders. Wird diese Zeichenkette innerhalb der Beschriftung verwendet, so wird sie zur Laufzeit als Tabulator interpretiert, mit dem der rechts daneben befindliche Teil spaltenlinear ausgerichtet werden kann. Diese Möglichkeit wird vor allem dazu genutzt, um hinter der eigentlichen Beschriftung auf eine zugehörige Beschleunigtaste zu verweisen. Ein Beispiel dafür ist `[Strg] [O]` hinter dem Menüpunkt DATEI|ÖFFNEN. Der Tabulator allein reicht allerdings noch nicht aus, um tatsächlich einen Beschleuniger (Hotkey) zu definieren: Wie das gemacht wird, erklärt der Abschnitt »Beschleunigertasten«.

Eigenschaften von Menüpunkten

Bisher wurde lediglich gezeigt, wie man die Beschriftungen der Menüpunkte ändert. Darüber hinaus besitzt jeder Menüpunkt noch weitere Attribute, beispielsweise einen Bezeichner oder gewisse Sondereigenschaften. Diese Attribute können entweder unmittelbar nach dem Anlegen eines neuen Menüpunkts oder auch später erfaßt werden. Es sind lediglich die erforderlichen Einträge im *Eigenschaften*-Fenster vorzunehmen.

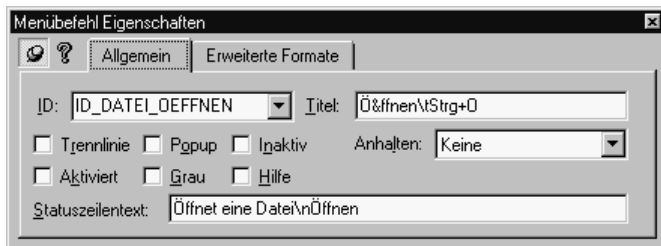


Abbildung 4.2: Eigenschaften von Menüpunkten

Zunächst einmal gilt es, den Bezeichner des Menüs festzulegen. Dazu ist im *Eigenschaften*-Fenster im Feld *ID* der gewünschte Bezeichner anzugeben (siehe Abbildung 4.2), für den Menüpunkt ÖFFNEN mit dem Bezeichner `ID_DATEI_OEFFNEN`. Dieser wird automatisch in der Form `ID_menüname_menüpunkt` vorgegeben. Es ist jedoch zu beachten, daß Umlaute weggelassen werden. Hier muß anschließend von Hand eingegriffen werden. Diese ID wird auch für den Text der Statuszeile verwendet, der in der *Zeichenfolgentabelle* der Ressource-Datei eingetragen wird. Sie können den Bezeichner aber auch bedenkenlos ändern, da der Wert entscheidend ist.

Anschließend können weitere Eigenschaften festgelegt werden. Zum Beispiel ist dies die Option *Grau*, die dafür sorgt, daß der Menüpunkt grau dargestellt wird und vom Anwender nicht ausgewählt werden kann. Andere Optionen sind *Trennlinie*, *Aktiviert* und *Popup*. Mit *Aktiviert* wird fest-

gelegt, daß der Menüpunkt ein kleines Häkchen erhält. *Trennlinie* erzeugt einen Menüpunkt, der nur aus einer horizontalen Trennlinie besteht (zwischen Suchen und Beenden), und mit *Popup* wird angegeben, daß der Menüpunkt der Name eines Untermenüs sein soll. Dabei kann ein Menüpunkt auch weitere Untermenüs enthalten.

IDs sind Bezeichner, für die mit einem *#define*-Statement in der Datei RESOURCE.H ein Wert festgelegt wird. Diese Zuordnung läßt sich anzeigen und auch editieren. Wählen Sie dazu im Menü ANSICHT den Punkt RESSOURCENSYMBOLE an (Abbildung 4.3). Dort sehen Sie alle verwendeten Bezeichner mit den zugeordneten Werten. Zusätzlich zeigt ein Haken an, daß der Bezeichner auch tatsächlich im Projekt verwendet wird.

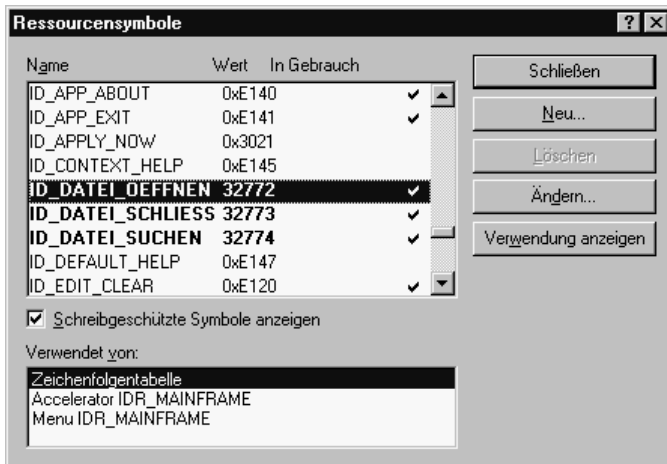


Abbildung 4.3: Editieren der Ressourcensymbole

Alle Bezeichner, die fett dargestellt sind, wurden im Projekt definiert. Alle anderen sind schreibgeschützt und werden nur angezeigt, wenn das Kontrollkästchen *Schreibgeschützte Symbole anzeigen* aktiviert ist. Über den Button NEU können neue IDs erstellt werden und LÖSCHEN erlaubt es, Bezeichner zu löschen, wenn diese nicht schreibgeschützt und nicht verwendet sind. Wo die Bezeichner verwendet werden, ist in der Liste *Verwendet von* zu erfahren. Durch Drücken von VERWENDUNG ANZEIGEN kann direkt zu dem in der Liste gewählten Ressourcentyp gesprungen werden.

Alle schreibgeschützten Ressource-IDs stammen aus inkludierten Dateien. Diese kann man sich über den Menüpunkt ANSICHT|RESSOURCEN-INCLUDES anzeigen lassen (siehe Abbildung 4.4). Die meisten dieser IDs stammen aus der Datei AFXRES.H, der Ressource-Datei der MFC. Des weiteren wird noch ermittelt, welche Bezeichner zur Compile-Zeit vergeben wurden.

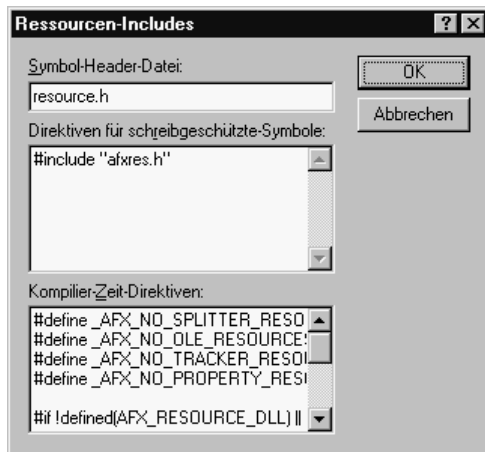


Abbildung 4.4: Anzeigen der Include-Dateien für Ressourcen

Nachdem das Editieren des Menüs beendet ist, sollte es gespeichert werden. Beim Speichern der Ressource-Datei wird zusätzlich eine Datei RESOURCE.H erstellt, die alle symbolischen Namen enthält, die als Bezeichner von Menüpunkten (und anderen Elementen) angegeben wurden. RESOURCE.H wird sowohl von SUCHTEXT.RC als auch von SUCHTEXT.CPP benötigt. Beide Dateien sollten nicht von Hand editiert werden, kann es doch dadurch zu Inkonsistenzen kommen, da die Ressourcen auch noch in einer APS-Datei verwaltet werden.



Falls es doch einmal nötig sein sollte, die RC-Datei von Hand zu ändern, muß auf jeden Fall die APS-Datei gelöscht werden, da sonst die manuellen Änderungen anschließend überschrieben werden würden.

Der nächste Schritt besteht darin, die Menü-Ressource mit dem Hauptfenster zu verbinden. Wie das gemacht wird, erfahren Sie im nächsten Abschnitt.



Weitere Informationen zur Bedienung des Menü-Editors finden sich in der Visual C++-Online-Dokumentation unter dem Stichwort »Menü-Editor«.

Integration einer Menü-Ressource

Um das in der Ressource-Datei deklarierte Menü in das Programm einzubinden, gibt es verschiedene Varianten. Die erste besteht darin, den Namen des Menüs im Aufruf der *Create*-Funktion der Hauptfensterklasse anzugeben. Dazu wird als sechster Parameter ein FAR-Zeiger auf eine nullterminierte Zeichenkette übergeben, die den Namen des Menüs enthält. Da in unserem Fall der Bezeichner als *int*-Konstante angegeben

wurde (IDR_MAINFRAME wird in RESOURCE.H per #define definiert), muß er mit dem Makro MAKEINTRESOURCE umgewandelt werden. MAKEINTRESOURCE erhält als Parameter den numerischen Bezeichner aus der Ressource-Datei und liefert eine Zeichenkette, die mit dem Windows-Ressourcen-Management kompatibel ist und den korrekten Namen generiert.

```
CMainWindow::CMainWindow()
{
    Create(NULL, "Hello World",
        WS_OVERLAPPEDWINDOW,
        rectDefault, NULL,
        MAKEINTRESOURCE(IDR_MAINFRAME));
}
```

Listing: Integration des Menüs über Create

Falls das Hauptfenster kein Menü haben soll, übergibt man diesem Parameter NULL. Diese Variante kann im »hello world«-Programm Verwendung finden, wo das Hauptfenster mit *Create* angelegt wird.

Wurde das Programm mit dem Anwendungs-Assistenten erstellt, wird ein Programmgerüst nach der Dokument-Ansicht-Philosophie aufgebaut. Dabei wird die Erstellung des Hauptfensters in den Template-Klassen gekapselt, so daß die Funktion *Create* nicht explizit aufgerufen wird. Statt dessen wird eine Instanz einer Dokument-Template-Klasse angelegt, der als erster Parameter der Bezeichner einer Ressource übergeben wird. Wenn der Anwendungs-Assistent verwendet wird, ist es der Bezeichner IDR_MAINFRAME. Dieser wird dann unter anderem zum Erzeugen des Hauptmenüs verwendet.

```
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
    IDR_MAINFRAME,
    RUNTIME_CLASS(CSuchTextDoc),
    RUNTIME_CLASS(CMainFrame),
    RUNTIME_CLASS(CSuchTextView));
```

Listing: Menü-Ressource im Dokument

Wurde die Dokument-Ansicht-Architektur im Anwendungs-Assistenten nicht verwendet, erfolgt die Angabe der Ressource-ID im Aufruf von LoadFrame, bei dem das Hauptfenster dynamisch erzeugt wird.

```
pFrame->LoadFrame(IDR_MAINFRAME,
    WS_OVERLAPPEDWINDOW | FWS_ADDTOTITLE, NULL, NULL);
```

Der Aufruf eines Menüpunktes löst zur Laufzeit des Programms eine WM_COMMAND-Nachricht aus, die im Parameter *wParam* den Bezeichner des Menüpunktes aus der Ressource-Datei enthält. Damit das MFC-Programm diese Nachricht empfangen kann, muß es für die zugehörige Fensterklasse eine Message-Map deklarieren und die Nachricht mit einer Methode verbinden. Dabei hilft uns der Klassen-Assistent in den verschiedenen Erscheinungsformen.

Es wird durch wiederholten Aufruf von ON_COMMAND zu jeder erwarteten WM_COMMAND-Nachricht ein Eintrag an die Message-Map angehängt. Dabei erwartet ON_COMMAND als ersten Parameter den Bezeichner der Menü-Ressource und als zweiten Parameter den Bezeichner der Methode, die diese Nachricht bearbeiten soll:

```
ON_COMMAND(<id>, <memberFxn>)
afx_msg void memberFxn( );
```

Am einfachsten arbeitet man so, daß im Menü im Ressourcen-Editor der gewünschte Menüpunkt selektiert wird. Anschließend ruft man durch Drücken von Strg W den Klassen-Assistenten auf. Die korrekte ID des Menüpunktes ist schon ausgewählt (Abbildung 4.5). In der Liste der Nachrichten wird COMMAND gewählt und durch Drücken der Schaltfläche FUNKTION HINZUFÜGEN die Nachrichtenbehandlungsfunktion eingefügt. Zu beachten ist noch, daß die ID für den Menüpunkt sowohl in der Hauptrahmenklasse *CMainFrame* als auch in der Ansicht *CSuchTextView* verfügbar ist. Wir geben unsere Texte später über die Ansicht aus und wählen deshalb die Ansicht-Klasse.

Als Ergebnis der oben beschriebenen Verfahrensweise erhalten wir einen Eintrag in der Message-Map der Ansichts-Klasse und die Deklaration und Definition der zugehörigen Funktion. Nun muß nur noch Code innerhalb der Funktion implementiert werden, um diese mit Leben zu füllen.

```
BEGIN_MESSAGE_MAP(CChildView, CFrameWnd)
    ///{AFX_MSG_MAP(CMainFrame)
        ON_COMMAND(ID_DATEI_OEFFNEN, OnDateiOeffnen)
    ///}AFX_MSG_MAP
END_MESSAGE_MAP()
void CMainFrame::OnDateiOeffnen()
{
    // TODO: Code für Befehlsbehandlungsroutine hier einfügen
}
```

Listing: Message-Map-Eintrag und Funktion für Datei/Öffnen

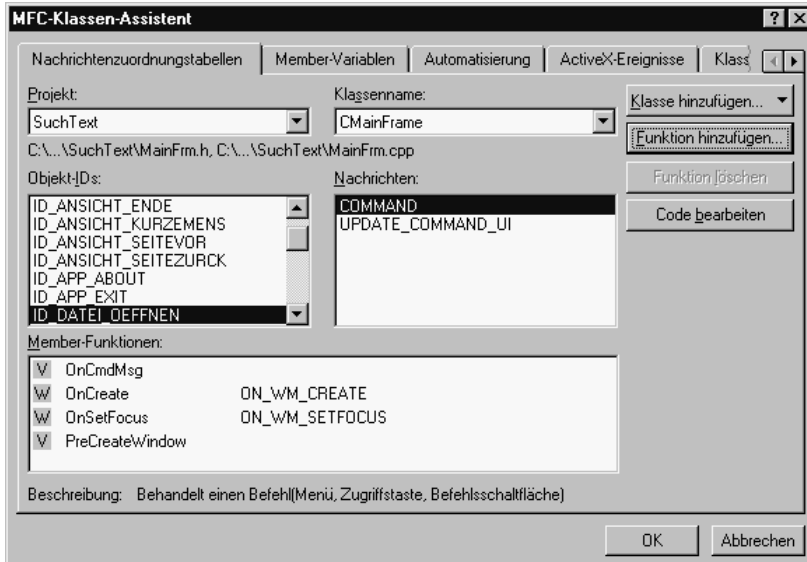


Abbildung 4.5: Verknüpfen von Menüpunkten mit Member-Funktionen

Immer, wenn Sie mit dem Klassen-Assistenten eine Nachrichtenbehandlungsroutine über FUNKTION HINZUFÜGEN anlegen, werden Sie in einem Dialog nach dem Namen der Funktion gefragt (siehe Abbildung 4.6). Der Name der Funktion beginnt immer mit `On`, gefolgt von Menü und Menüpunkt. Sie können hier den Funktionsnamen nach eigenen Wünschen festlegen. Es ist auch möglich, für mehrere Menüpunkte ein und dieselbe Funktion anzugeben. Zusätzlich wird im Dialog noch die Nachricht und die Ressource-ID angegeben.

In *SuchText* gibt es eine ganze Reihe von Menüpunkten, auf die das Programm reagieren muß. Die Deklaration der Message-Map ist daher recht lang. Um die Bezeichner der Menüpunkte bei der Deklaration der Message-Map verwenden zu können, muß die generierte Header-Datei `RESOURCE.H` direkt oder indirekt über eine andere Header-Datei eingebunden werden. In unserem Fall wird dies vom Anwendungs-Assistenten in `SUCHTEXT.H` vorgenommen.

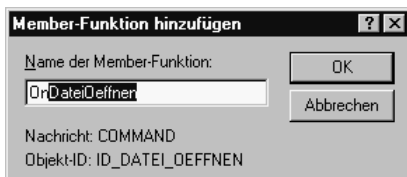


Abbildung 4.6: Anlegen einer Nachrichtenbehandlungsroutine

```

BEGIN_MESSAGE_MAP(CChildView, CWnd)
    //{AFX_MSG_MAP(CChildView)
    ON_COMMAND(ID_DATEI_OEFFNEN, OnDateiOeffnen)
    ON_COMMAND(ID_DATEI_SCHLIESSEN, OnDateiSchliessen)
    ON_COMMAND(ID_DATEI_SUCHEN, OnDateiSuchen)
    ON_COMMAND(ID_ANSICHT_ANFANG, OnAnsichtAnfang)
    ON_COMMAND(ID_ANSICHT_ENDE, OnAnsichtEnde)
    ON_COMMAND(ID_ANSICHT_KURZEMENS, OnAnsichtKurzemenues)
    ON_COMMAND(ID_ANSICHT_SEITEVOR, OnAnsichtSeitevor)
    ON_COMMAND(ID_ANSICHT_SEITEZURCK, OnAnsichtSeitezurueck)
    ON_COMMAND(ID_SCHRIFT_ANSI, OnSchrift)
    ON_COMMAND(ID_SCHRIFT_FARBE_BLAU, OnSchriftFarbe)
    ON_COMMAND(ID_SCHRIFT_FARBE_GRUEN, OnSchriftFarbe)
    ON_COMMAND(ID_SCHRIFT_FARBE_SCHWARZ, OnSchriftFarbe)
    ON_COMMAND(ID_SCHRIFT_FARBE_WEISS, OnSchriftFarbe)
    ON_COMMAND(ID_SCHRIFT_INVERTIERT, OnSchriftInvertiert)
    ON_UPDATE_COMMAND_UI(ID_SCHRIFT_INVERTIERT,
OnUpdateSchriftInvertiert)
    ON_UPDATE_COMMAND_UI(ID_ANSICHT_KURZEMENS, OnUpdateKurzemenues)
    ON_COMMAND(ID_SCHRIFT_SANSERIF10, OnSchrift)
    ON_COMMAND(ID_SCHRIFT_SANSERIF24, OnSchrift)
    ON_COMMAND(ID_SCHRIFT_SYSTEM, OnSchrift)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

```

Listing: Die Message-Map in SuchText

Jeder der Menübezeichner aus der Ressource-Datei wird dabei mit einer Methode der Klasse *CSuchTextView* verbunden, beispielsweise `ID_DATEI_OEFFNEN` mit `OnDateiOeffnen` oder `ID_ANSICHT_ANFANG` mit `OnAnsichtAnfang`. Dies führt im laufenden Programm dazu, daß durch das Auswählen eines Menüpunkts jeweils die zugehörige Methode aufgerufen wird. In den Methoden sind die Schritte zur Bearbeitung der zugehörigen Menüfunktion realisiert. Um die Beziehung zu den einzelnen Menüpunkten zu verdeutlichen, haben die Methoden ähnliche Namen wie die Menübezeichner: Nach dem Präfix *On* folgt der Menübezeichner in typischer Methoden-Namenskonvention, d.h., mit eingebetteten Großbuchstaben. Diese Vorgehensweise ist zwar nicht zwingend nötig, aber wegen der besseren Lesbarkeit durchaus empfehlenswert.



Zu jedem Menüpunkt muß in der abgeleiteten Klasse eine Methode deklariert und mit einem geeigneten `ON_COMMAND`-Message-Map-Eintrag die Verbindung dazu hergestellt werden. Falls die Verbindung zwischen einem Menüpunkt, der in der Ressource-Datei definiert wurde, und einer Methode noch nicht hergestellt wurde, bleibt der Menüpunkt inaktiv: Das

Programm reagiert nicht auf Versuche des Benutzers, diesen Menüpunkt auszuwählen. Achten Sie darauf, daß die Funktionen in der Klasse *CChildView* und nicht in *CMainFrame* angelegt werden, da nur dort die Methode *OnPaint* verfügbar ist.

Zwar ist diese Vorgehensweise, Menübezeichner mit Methoden zu verbinden, sehr einfach zu verstehen und zu handhaben; in der Praxis ist es aber oft zu umständlich, für jeden einzelnen Menüpunkt eine eigene Methode zu schreiben. Vielmehr ähneln sich manche Menüpunkte so sehr, daß es angebracht ist, eine gemeinsame Methode zu verwenden, in der dann der Menübezeichner als Parameter auftaucht. Leider ist dies in den Microsoft Foundation Classes nicht explizit vorgesehen, mit einem kleinen Trick kann man sich aber behelfen.

In *SUCHTEXT.CPP* werden die Menübezeichner *ID_SCHWARZ* bis *ID_WEISS* auf die Methode *OnSchriftFarbe* umgeleitet. Da *OnSchriftFarbe* (wie alle *ON_COMMAND*-Methoden) parameterlos ist, erfährt es bei einem Aufruf zunächst nicht, welcher der fünf Menüpunkte tatsächlich ausgewählt wurde. Um dies herauszubekommen, gibt es die Funktion *GetCurrentMessage*.

```
class Cwnd:
    static const MSG* PASCAL GetCurrentMessage( );
```

GetCurrentMessage liefert einen Zeiger auf eine *MSG*-Struktur, in der genau die Nachricht steht, die gerade bearbeitet wird. Damit können über die Member *wParam* und *lParam* der Struktur zusätzliche Informationen beschafft werden. Da der Menübezeichner bei einer *WM_COMMAND*-Nachricht in *wParam* steht, kann die Methode mit Hilfe des Wertes *GetCurrentMessage()->wParam* die entsprechenden Unterscheidungen durchführen. In *OnSchriftFarbe* sieht dies dann beispielsweise so aus:

```
void CChildView::OnSchriftFarbe()
{
    nFarbe = GetCurrentMessage()->wParam;
    InvalidateRect(NULL,TRUE);
    //Neuzeichnen
    UpdateWindow();
}
```

Nähere Erklärungen zu dieser Methode folgen weiter unten.

4.2 Verändern von Menüpunkten

Die Menüs und Untermenüs eines Windows-Programms sind nicht nur frei definierbar, sondern können auch zur Laufzeit verändert werden. Dabei ist zwischen dem Verändern eines gesamten Menüs und dem einzelner

Menüpunkte zu unterscheiden. Während gesamte Menüs im nächsten Thema abgehandelt werden, sollen hier zwei wichtige Modifikationsmöglichkeiten einzelner Menüpunkte erklärt werden:

- ▼ das Aktivieren oder Deaktivieren eines Menüpunkts sowie das Darstellen nicht aktiver Menüpunkte durch grauen Text
- ▼ das Markieren eines Menüpunkts mit einem kleinen Häkchen sowie das Entfernen dieser Markierung.

4.2.1 Die Klasse *CMenu*

Die Microsoft Foundation Classes Library stellt für den Zugriff auf Menüs die spezielle Klasse *CMenu* zur Verfügung. Diese kapselt die Windows-Struktur *HMENU* und stellt die Zugriffe darauf bereit. Ein *CMenu*-Objekt ist in der Lage, ein einzelnes Menü eines Programms inklusive der zugehörigen Menüpunkte zu repräsentieren. So kann beispielsweise das Hauptmenü von *SuchText* durch ein *CMenu*-Objekt bearbeitet werden, dasselbe gilt für das *SYSTEM*- oder *DATEI*-Menü.

Um einen Menüpunkt verändern zu können, braucht man also erst einmal ein *CMenu*-Objekt, welches das gewünschte Menü repräsentiert. Um *CMenu*-Objekte aus bereits bestehenden Menüs zu erzeugen (*SuchText* besitzt ja bereits ein Menüsystem, das beim Anlegen des Hauptfensters aus der Ressource-Datei geladen wurde), gibt es die Methoden *GetMenu* und *GetSubMenu*.

```
class CWnd:  
    CMenu* GetMenu( ) const;  
class CMenu:  
    CMenu *GetSubMenu( int nPos ) const;
```

GetMenu ist Methode der Klasse *CWnd* und liefert einen *CMenu*-Zeiger auf das Hauptmenü des Fenster-Objekts. Mit *GetSubMenu* kann dagegen ein Zeiger auf *CMenu*-Objekte für Untermenüs beschafft werden. Durch den folgenden Programmausschnitt wird ein Zeiger auf ein *CMenu*-Objekt für das Hauptmenü des aktuellen Fensters beschafft:

```
CMenu *menu;  
menu = GetMenu();
```

Um nun auf Untermenüs zuzugreifen, wird die Methode *GetSubMenu* aufgerufen. *GetSubMenu* erwartet als Parameter den Index des gewünschten Untermenüs, wobei der erste Menüpunkt den Index 0 hat. Falls an dieser Stelle kein Untermenü zu finden ist (sondern lediglich ein normaler Menüpunkt steht), ist der Rückgabewert *NULL*.

Um beispielsweise ein *CMenu*-Objekt für das DATEI-Menü von *SuchText* zu beschaffen, geht man folgendermaßen vor:

```
CMenu *menu;  
menu = GetMenu()->GetSubMenu(0);
```

Nach dem Aufruf repräsentiert die Variable `menu` also das DATEI-Menü, weil es den Positionsindex 0 unter allen Menüpunkten des Hauptmenüs hat. Würde das Dateimenü an dritter Position (also Positionsindex 2) ein weiteres Untermenü besitzen, könnte man dazu mit

```
CMenu *menu;  
menu = (GetMenu()->GetSubMenu(0))->GetSubMenu(2);
```

ebenfalls ein *CMenu*-Objekt gewinnen. Man sollte sich nicht dadurch verwirren lassen, daß das Hauptmenü seine Menüpunkte horizontal auf dem Bildschirm anordnet, während alle anderen Menüs vertikal ausgerichtet sind; beide Arten von Menüs sind auf der Ebene der Programmierung gleichberechtigt – *GetMenu* beschafft aber immer ein Objekt für das horizontale Hauptmenü.

4.2.2 Temporäre Objekte

Beide Methoden, sowohl *GetMenu* als auch *GetSubMenu*, liefern einen Zeiger auf ein *temporäres CMenu*-Objekt, das während des Aufrufs angelegt wurde. Solche Zeiger auf temporäre Objekte tauchen bei der Programmierung mit den Foundation Classes recht häufig auf; man sollte sie *nicht zu lange* speichern, weil das zugeordnete Objekt vor dem Verarbeiten der nächsten Nachricht wahrscheinlich wieder zerstört wird. In der Microsoft-Dokumentation einer solchen Methode (siehe *GetMenu* in der On-line-Dokumentation) steht immer die Warnung: »The returned pointer may be temporary and should not be stored for later use«.

Was heißt das nun genau? Die aufgerufene Methode sieht sich veranlaßt, einen Zeiger auf ein Objekt liefern zu müssen, das möglicherweise noch gar nicht existiert; sie wird dieses Objekt also auf dem Heap erzeugen. So weit, so gut, aber wer löscht das Objekt wieder? Damit nicht der Anwendungsprogrammierer mit der (fehlerträchtigen) Aufgabe belastet wird, alle nicht mehr benötigten Objekte zu löschen, tut dies die Library selbst. Dazu verfügt sie über einen einfachen Garbage Collector, der bei jedem Aufruf von *OnIdle* (siehe Kapitel 3.2.9) alle temporären Objekte löscht – beispielsweise auch das *CMenu*-Objekt, das von *GetMenu* geliefert wurde.

Nun wird *OnIdle* aber immer nur dann aufgerufen, wenn keine weiteren Nachrichten anliegen, insbesondere also nicht asynchron und damit nicht mitten in einer Methode. Die Zeiger auf temporäre Objekte bleiben also mindestens so lange gültig, bis die aktuelle Nachricht beendet ist. Erst

wenn in der globalen Nachrichtenschleife nach der nächsten Nachricht gesucht und damit möglicherweise *OnIdle* aufgerufen wird, kann es sein, daß temporäre Objekte gelöscht werden. Man sollte es daher vermeiden, einen Zeiger auf ein temporäres Objekt als statische oder globale Variable anzulegen und zwischen zwei Nachrichten am Leben zu erhalten.

4.2.3 Menüpunkte aktivieren und deaktivieren

Bei der Windows-Programmierung entspricht es gutem Stil, dem Anwender momentan nicht verfügbare Menüpunkte optisch zu entziehen; Es ist dabei üblich, diese Menüpunkte als sichtbaren Hinweis darauf in grauer Schrift darzustellen. So sind zum Beispiel die Menüpunkte DATEI|SCHLIESSEN und DATEI|SUCHEN nur dann sinnvoll, wenn bereits eine Datei geöffnet ist, andernfalls sollte der Benutzer gar nicht erst die Möglichkeit haben, diese Option auszuwählen. In *SuchText* werden beispielsweise erst in der Methode *OnDateiOeffnen* nach dem Auswählen einer neuen Datei die Menüpunkte DATEI|SCHLIESSEN und DATEI|SUCHEN (die ursprünglich grau waren) aktiviert:

```
void CChildView::OnDateiOeffnen()
{
    ...
    //wenn Datei geöffnet
    CMenu *menu;
    menu = AfxGetMainWnd()->GetMenu()->GetSubMenu(0);
    menu->EnableMenuItem(ID_DATEI_SCHLIESSEN,
MF_BYCOMMAND|MF_ENABLED);
    menu->EnableMenuItem(ID_DATEI_SUCHEN, MF_BYCOMMAND|MF_ENABLED);
    ...
}
```

Listing: Aktivieren eines Menüpunktes über *EnableMenuItem*

Nachdem klar ist, wie Zeiger auf Objekte beschafft werden, die zu vorhandenen Menüs gehören, ist es einfach, einen einzelnen Menüpunkt zu aktivieren oder zu deaktivieren. Die Hauptarbeit übernimmt dabei die Methode *EnableMenuItem* der Klasse *CMenu*:

```
class Cmenu:
    UINT EnableMenuItem(  UINT nIDEnableItem,
                          UINT nEnable );
```

Mit *EnableMenuItem* ist es möglich, die Verfügbarkeit eines Menüpunkts auf drei verschiedene Arten zu beeinflussen:

- ▼ Ein Menüpunkt kann *aktiv* sein. In diesem Fall wird er normal angezeigt und kann vom Benutzer ausgewählt werden.
- ▼ Ein Menüpunkt kann *inaktiv* sein. In diesem Fall wird er unverändert angezeigt, kann aber nicht mehr ausgewählt werden.
- ▼ Ein Menüpunkt kann *grau* sein. In diesem Fall wird der Menütex in grauer Schrift angezeigt und kann nicht mehr ausgewählt werden.

All diese Veränderungen können mit *EnableMenuItem* vorgenommen werden. Der erste Parameter *nIDEnableItem* gibt an, welcher Menüpunkt verändert werden soll; er ist abhängig vom zweiten Parameter, in dem zwei ODER-verknüpfte Bitflags übergeben werden müssen. Eines der Bitflags muß dabei MF_ENABLED, MF_DISABLED oder MF_GRAYED sein und zeigt an, welcher Art die Veränderung ist.

Das zweite Flag muß entweder MF_BYCOMMAND oder MF_BYPOSITION sein. Ist es MF_BYPOSITION, so gibt der erste Parameter den gewünschten Menüpunkt durch seinen Positionsindex im Menü (beginnend bei 0) an; ist es MF_BYCOMMAND, so gibt der erste Parameter den Bezeichner des Menüpunkts an (der in der RC-Datei festgelegt wurde, und der auch beim Auswählen des Menüpunkts in der WM_COMMAND-Meldung mitgeschickt wird).

In diesem Buch wird vorwiegend mit MF_BYCOMMAND gearbeitet, denn der andere Ansatz ist etwas gefährlich: Falls die Reihenfolge der Punkte innerhalb eines Menüs geändert oder ein Menüpunkt eingefügt oder entfernt wurde, müssen bei MF_BYPOSITION auch alle Aufrufe von *EnableMenuItem* angepaßt werden.

Beim Schließen einer Datei werden dieselben Menüpunkte grau, die beim Öffnen aktiviert wurden:

```
void CChildView::OnDateiSchliessen()
{
    ...
    CMenu *menu;
    menu = AfxGetMainWnd()->GetMenu()->GetSubMenu(0);
    menu->EnableMenuItem(ID_DATEI_SCHLIESSEN,
MF_BYCOMMAND|MF_GRAYED);
    menu->EnableMenuItem(ID_DATEI_SUCHEN, MF_BYCOMMAND|MF_GRAYED);
    ...
}
```

Listing: Deaktivieren eines Menüpunktes über EnableMenuItem

Auch hier wird *EnableMenuItem* aufgerufen und mit Hilfe des Bitflags *MF_GRAYED* der zuvor aktivierte Menüpunkt grau dargestellt.

Nicht benötigte Menüpunkte sauber zu deaktivieren, hat zusätzlich den Vorteil, daß während der inaktiven Phase die zugehörigen Nachrichten nicht auftauchen können. In den so geschützten Methoden ist es dann nicht erforderlich, am Anfang einen Test zu machen, ob der Aufruf überhaupt erlaubt ist, bzw. ob erst noch andere Tätigkeiten ausgeführt werden müssen, um ihn zu erlauben. Zudem ist bei einem deaktivierten Menüpunkt automatisch auch der korrespondierende Beschleuniger deaktiviert, ohne daß man sich als Programmierer darum kümmern müßte. Man darf allerdings nicht vergessen, auch den Anfangszustand der Menüs (unmittelbar nach dem Programmstart) zu definieren; dies kann entweder durch Aufrufe von *EnableMenuItem* während der Initialisierungsphase geschehen oder gleich in der Ressource-Datei erledigt werden:

```
IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&Datei"
    BEGIN
        ...
        MENUITEM "&Schließen\tStrg+C",          ID_DATEI_SCHLIESSEN, GRAYED
        ...
    END
```

Listing: Ein deaktivierter Menüpunkt in der Ressource-Datei

Es existiert noch eine weitere Möglichkeit, Menüpunkte zu aktivieren oder zu deaktivieren, die auch sehr einfach verwendbar ist. Voraussetzung dafür sind eine boolesche Variable und ein Eintrag in der Message-Map. Die Variable wird in der Header-Datei definiert, so daß sie für alle Methoden der Klasse sichtbar ist. Sie spiegelt den Zustand des Menüpunkts wieder: *TRUE* – Menüpunkt aktiviert, *FALSE* – Menüpunkt deaktiviert. Mit Hilfe des Klassen-Assistenten wird dann für die entsprechenden Menüpunkte eine Verknüpfung zur *UPDATE_COMMAND_UI*-Nachricht hergestellt. Die in dieser Nachrichtenbehandlungsroutine verfügbare Variable vom Typ *CCmdUI* ermöglicht auf einfache Weise Zugriffe auf den zugehörigen Menüpunkt. Diese Klasse kann auch bei Symbolleisten oder Statuszeilen verwendet werden (siehe Online Dokumentation).

```
class CCmdUI:
    virtual void Enable( BOOL bOn = TRUE );
    virtual void SetCheck( int nCheck = 1 );
```

In der Routine *OnUpdateSchriftInvertiert* wird dann die Methode *SetCheck* mit der booleschen Variablen *bInvers* als Parameter verwendet.

```
void CChildView:: OnUpdateSchriftInvertiert (CCmdUI* pCmdUI)
{
    pCmdUI -> SetCheck(bInvers);
}
```

Die Methode *OnUpdateXx* muß nicht explizit aufgerufen werden. Dies wird automatisch vom Framework erledigt, wenn ein Menü aufgeklappt wird. Dann wird für jeden Menüpunkt der UpdateUI-Handler aufgerufen. Ein Vorteil dieser Methode ist das Verwenden der boolschen Variablen. Diese können je nach Sichtbarkeit in der Anwendung von beliebigen Funktionen gesetzt werden. Alles andere erledigt sich von selbst.

Bei der Erstellung des Beispielprogramms *SuchText* zu diesem Buch gab es Probleme mit der Methode *EnableMenuItem*, die es nicht erlaubten, den Menüpunkt grau darzustellen. Im Beispiel wurde deshalb generell die Methode *Enable* von *CCmdUI* verwendet.



4.2.4 Menüpunkte markieren

Die zweite Art, Menüpunkte zu verändern, besteht darin, ein kleines Häkchen an einen Menüpunkt anzuhängen oder es davon zu entfernen. Hiermit wird dem Benutzer in der Regel ein interner Zustand angezeigt, der mit Hilfe des Menüpunkts aktiviert oder deaktiviert werden kann. In *SuchText* werden beispielsweise die Menüpunkte zur Auswahl der Schriftfarbe bzw. -art auf diese Weise markiert, um dem Benutzer die gerade aktiven Einstellungen kenntlich zu machen.

Um einen Menüpunkt zu (de-)markieren, kann man ganz ähnlich vorgehen wie bei der Aktivierung, nur daß nun anstelle von *EnableMenuItem* die Methode *CheckMenuItem* verwendet wird:

```
class Cmenu:
    UINT CheckMenuItem(  UINT nIDCheckItem,
                        UINT nCheck );
```

Die Bedeutung der Parameter stimmt haargenau mit der von *EnableMenuItem* überein, mit der Ausnahme, daß die Bitflags zum Aktivieren und Deaktivieren der Markierung *MF_CHECKED* und *MF_UNCHECKED* heißen; *MF_ENABLED*, *MF_DISABLED* und *MF_GRAYED* dürfen dagegen nicht verwendet werden. Die Bedeutung von *MF_BYCOMMAND* und *MF_BYPOSITION* ist unverändert.

In der Methode *OnSchriftFarbe* werden beispielsweise nach der Auswahl eines Menüpunkts zum Ändern der Schreibfarbe die Häkchen verändert: Nur der Menüpunkt mit der gerade aktivierten Farbe bekommt ein Häkchen, allen anderen wird es entzogen.

```

void CChildView::OnSchriftFarbe()
{
    CMenu *menu;
    nFarbe = GetCurrentMessage()->wParam - ID_SCHRIFT_FARBE_SCHWARZ;
    menu = (AfxGetMainWnd()->GetMenu()->GetSubMenu(3)
->GetSubMenu(7);
    for (int i=0; i<5; ++i)
        menu->CheckMenuItem(ID_SCHRIFT_FARBE_SCHWARZ + i,
            MF_BYCOMMAND | nFarbe == i?MF_CHECKED:MF_UNCHECKED);
    InvalidateRect(NULL,TRUE);
    //Neuzeichnen
    UpdateWindow();
}

```

Listing: Einstellen der Schriftfarbe und Markieren im Menü

Hier macht sich das Programm zusätzlich die Tatsache zunutze, daß die Menü-Identifikatoren zur Einstellung der Farben eine aufsteigende Zahlenfolge bilden und sich jeweils um 1 erhöhen:

```

#define ID_SCHRIFT_FARBE_SCHWARZ      32785
#define ID_SCHRIFT_FARBE_WEISS       32786
#define ID_SCHRIFT_FARBE_BLAU        32787
#define ID_SCHRIFT_FARBE_GRUEN       32788

```

So kann die Arbeit des Markierens und Demarkierens in einer *for*-Schleife für alle betroffenen Menüpunkte auf einmal erledigt werden.

Obwohl es so scheint, als ob das Markieren von Menüpunkten lediglich kosmetischer Natur ist, auf den Programmablauf aber keinen weiteren Einfluß hat, stimmt dies nicht ganz. Es ist den Programmen nämlich möglich, mit Hilfe der Methode *GetMenuState* aus *CMenu* den Zustand eines Menüpunkts abzufragen – insbesondere also festzustellen, ob er markiert ist oder nicht. So könnte beispielsweise die globale boolesche Variable eingespargt werden, die typischerweise zu einem markierbaren Menüpunkt seinen aktuellen Zustand speichert, in dem einfach der aktuelle Zustand des Menüpunktes mit *GetMenuState* abgefragt wird.

```

class Cmenu:
    UINT GetMenuState(  UINT nID,  UINT nFlags);

```

Die Parameter ähneln denen der anderen beiden Funktionen: *nID* bezeichnet den Menüpunkt, und *nFlags* zeigt (durch einen der Werte *MF_BYCOMMAND* oder *MF_BYPOSITION*) an, was unter *nID* zu verstehen ist. Der Rückgabewert der Methode ist ein bitweise ODER-verknüpftes

Gemisch der Flags `MF_CHECKED`, `MF_UNCHECKED`, `MF_GRAYED`, `MF_DISABLED`, `MF_ENABLED` und noch einigen anderen, aus dem man den jeweiligen Zustand des Menüpunkts ablesen kann.

Auch hier bietet sich, vor allem beim Arbeiten mit einer Ansicht, die Verwendung der `UPDATE_COMMAND_UI`-Nachricht an. In einer numerischen Variablen wird der Wert der gewählten Schrift gespeichert. Dazu wird in der `ON_COMMAND`-Nachricht *OnSchrift*, auf die alle Schriftwahl-Menüpunkte verweisen, die gewählte ID des Menüpunkts in der Variablen `nSchrift` vermerkt.

```
void CChildView::OnSchrift()
{
    nSchrift = GetCurrentMessage()->wParam;
}
```

In der Methode `OnUpdateSchriftXx` wird entsprechend der ID der Menüpunkt markiert oder dessen Markierung entfernt.

```
void CChildView::OnUpdateSchriftSansserif10(CCmdUI* pCmdUI)
{
    if(nSchrift == ID_SCHRIFFT_SANSSERIF10)
        pCmdUI -> SetCheck(TRUE);
    else
        pCmdUI -> SetCheck(FALSE);
}
```

4.3 Dynamisch erzeugte Menüs

4.3.1 Menüpunkte zur Laufzeit löschen oder einfügen

Bisher wurde das Menüsystem eines Programms immer in der Ressource-Datei definiert und damit bereits zum Compile-Zeitpunkt festgelegt; in der Praxis kann es jedoch nützlich sein, das Menü zur Laufzeit zu ändern.

Auch mit den Microsoft Foundation Classes können Menüs zur Laufzeit erzeugt, verändert oder gelöscht werden. Der Schlüssel zu diesen Funktionen liegt in der Klasse *CMenu*, die bereits im vorigen Abschnitt vorgestellt wurde; sie stellt Methoden zur Verfügung, mit denen all diese Aufgaben erledigt werden können.

Angenommen, es soll eine variable Menüstruktur für ein Programm realisiert werden, so daß der Anwender zwischen langen und kurzen Menüs umschalten kann. Genauer gesagt: Es soll möglich sein, den Aufbau des Untermenüs »Schrift« wahlweise auf eine der beiden Arten in Abbildung 4.7 darstellen zu lassen.

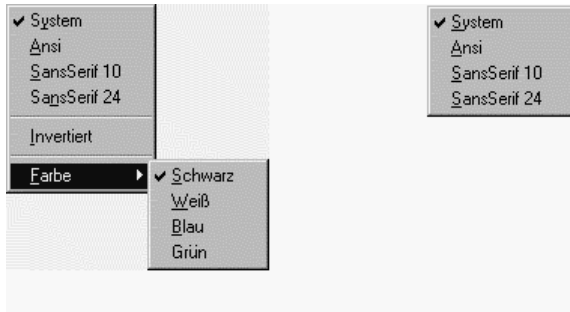


Abbildung 4.7: Lange und kurze Menüs

Angenommen, das SCHRIFT-Menü befindet sich an vierter Stelle im Hauptmenü, und zum Umschalten zwischen lang und kurz werden die Methode `OnAnsichtKurzemenues` und die Variable `bKurz` der Klasse `CChildView` verwendet.

```
void CChildView::OnAnsichtKurzemenues()
{
    CMenu *menu = AfxGetMainWnd()->GetMenu()->GetSubMenu(3);
    if (lbKurz)        //z. Zt. lange Menüs aktiv
    {
        int nItems = menu->GetMenuItemCount();    //Anzahl Menüpunkte
        for (int i=0; i<nItems; ++i)
            menu->DeleteMenu(0,MF_BYPOSITION);    //alle Löschen
        menu->AppendMenu(MF_STRING, ID_SCHRIFT_SYSTEM,"&System");
        menu->AppendMenu(MF_STRING, ID_SCHRIFT_ANSI,"&Ansi");
        menu->AppendMenu(MF_STRING, ID_SCHRIFT_SANSSEIF10,
                        "&SansSerif 10");
        menu->AppendMenu(MF_STRING, ID_SCHRIFT_SANSSEIF24,
                        "&SansSerif 24");

        DrawMenuBar();
        bKurz = TRUE;
    }
    else
    {
        CMenu *pPopupMenu = new CMenu;
        pPopupMenu->CreatePopupMenu();            //Popup-Menü für Farbe
        pPopupMenu->AppendMenu(MF_STRING,
                                ID_SCHRIFT_FARBE_SCHWARZ,"&Schwarz");
        pPopupMenu->AppendMenu(MF_STRING,
                                ID_SCHRIFT_FARBE_WEISS,"&Weiß");
        pPopupMenu->AppendMenu(MF_STRING,
                                ID_SCHRIFT_FARBE_BLAU,"&Blau");
    }
}
```

```

pPopupMenu->AppendMenu(MF_STRING,
    ID_SCHRIFT_FARBE_GRUEN, "&Grün");
menu->AppendMenu(MF_STRING|MF_SEPARATOR,
    0, "");
menu->AppendMenu(MF_STRING,
    ID_SCHRIFT_INVERTIERT, "&Invertiert");
menu->AppendMenu(MF_STRING|MF_SEPARATOR,
    0, "");
menu->AppendMenu(MF_POPUP,
    (UINT)pPopupMenu->m_hMenu, "&Farbe");
DrawMenuBar();
// aktuelle Farbe markieren
menu = (AfxGetMainWnd()->GetMenu()->GetSubMenu(3)-
>GetSubMenu(7);
    for (int i=0; i<5; ++i)
        menu->CheckMenuItem(ID_SCHRIFT_FARBE_SCHWARZ + i,
            MF_BYCOMMAND | (nFarbe == i?MF_CHECKED:MF_UNCHECKED));
    bKurz = FALSE;
}
}

```

Listing: Methode zum Umschalten zwischen langen und kurzen Menüs

Zuerst wird ein *CMenu*-Objekt *menu* für das Untermenü an der vierten Position beschafft (und zwar mit *GetSubMenu(3)*, weil das erste Untermenü den Index 0 hat). Anhand der Variablen *bKurz* wird entschieden, ob ein kurzes oder langes Menü erzeugt werden soll. Bei einem kurzen Menü werden zuerst alle Punkte entfernt, indem die Methode *DeleteMenu* (Anzahl wird mit *GetMenuItemCount* ermittelt) den jeweils ersten Menüpunkt löscht.

```

class Cmenu:
    BOOL DeleteMenu(UINT nPosition, UINT nFlags);
    UINT GetMenuItemCount( ) const;

```

Der erste Parameter *nPosition* gibt den zu entfernenden Menüpunkt an; der zweite *nFlags* bestimmt, ob der erste Parameter die Position (MF_BYPOSITION) oder den Bezeichner (MF_BYCOMMAND) des Menüpunkts angibt. Der Einfachheit halber wird immer der erste Menüpunkt entfernt: Damit ist das Menü leer. Nun werden die Punkte für das kurze Menü durch Aufruf der Methode *AppendMenu* angehängt.

```
class Cmenu:
    BOOL AppendMenu(UINT nFlags,
                    UINT nIDNewItem = 0,
                    LPCTSTR lpszNewItem = NULL);
    BOOL AppendMenu(UINT nFlags,
                    UINT nIDNewItem,
                    const CBitmap* pBmp );
```

AppendMenu erwartet als ersten Parameter eine ODER-Verknüpfung von Bitflags, um die Eigenschaften des Menüpunkts anzugeben. Hierbei können die bereits bekannten Flags `MF_CHECKED`, `MF_GRAYED` etc. zur Steuerung der visuellen Darstellung mit einem der folgenden Flags zum Typ des Menüpunkts kombiniert werden:

Flag	Art des Menüpunktes
<code>MF_STRING</code>	Gibt an, daß der Menüpunkt eine Zeichenkette ist. In diesem Fall muß <code>nIDNewItem</code> den Menübezeichner und <code>lpszNewItem</code> den Namen des Menüpunkts enthalten.
<code>MF_SEPARATOR</code>	Gibt an, daß der Menüpunkt eine horizontale Linie sein soll; die beiden anderen Parameter werden ignoriert.
<code>MF_POPUP</code>	Gibt an, daß der Menüpunkt ein Untermenü darstellen soll. In diesem Fall muß <code>nIDNewItem</code> den Handle des einzufügenden Untermenüs übergeben.

Tabelle 4.1: Bitflags für *AppendMenu*

Durch mehrmaliges Aufrufen von *AppendMenu* werden alle gewünschten Punkte in das Menü eingefügt. Danach verhält sich das dynamisch erzeugte Menü bezüglich des Nachrichtenaustauschs genauso wie ein statisches Menü: Wenn der Benutzer einen Menüpunkt auswählt, wird die zugehörige (in `nIDNewItem` übergebene) `WM_COMMAND`-Nachricht an das Fenster geschickt. Um das neu erzeugte Menü visuell korrekt darzustellen, ist ganz am Ende die Methode *DrawMenuBar* von *CWnd* aufzurufen. Dies gilt auch, wenn das Fenster derzeit nicht aktiv ist oder von anderen Fenstern verdeckt wird.

```
class CWnd:
    void DrawMenuBar();
```

Bei der Darstellung des langen Menüs müssen keine Menüpunkte gelöscht, sondern nur angehängt werden. Dies erfolgt wie bereits beschrieben mit der Methode *AppendMenu*. Zum Schluß muß dann wieder *DrawMenuBar* aufgerufen werden, um das Menü neu zu zeichnen. Die einzige Schwierigkeit besteht darin, die Menüpunkte im Popup-Menü *Farbe* wiederherzustellen. Um dieses zu erzeugen, benötigt *AppendMenu* eine

HMENU-Struktur. Diese wird wiederum aus einem *CMenu*-Objekt erstellt (genauer gesagt wird die Member-Variable *m_hMenu* benutzt). Das Objekt wird durch Aufruf von *CreatePopupMenu* zum Popup-Menü. Danach werden alle Menüpunkte des Popup-Menüs eingefügt. *AppendMenu* fügt dann das gesamte Popup-Menü ein.

```
class CMenu:
    BOOL CreatePopupMenu( );
```

Schließlich wird nach Erzeugung des langen Menüs noch die zuvor aktive Farbe mit einem Haken versehen. Die Methodik dazu wurde bereits beschrieben. Auch an diesem Beispiel sieht man, daß es günstiger ist, die *UPDATE_COMMAND_UI*-Nachricht zu verwenden. Man muß sich dann nicht mehr um das Markieren kümmern.

4.3.2 Erzeugen eines Kontext-Menüs

Angenommen, das Programm soll so erweitert werden, daß auf Drücken der rechten Maustaste an der aktuellen Mausposition ein Fenster mit einem Menü erscheint, das aus den Punkten *ÖFFNEN*, *SCHLIESSEN* und *SUCHEN* besteht. Auch hierfür bietet die Klasse *CMenu* die geeigneten Methoden: *CreatePopupMenu* und *TrackPopupMenu*. Die Vorgehensweise beim Erzeugen eines Popup-Menüs besteht aus den folgenden Schritten:

1. Anlegen eines leeren *CMenu*-Objekts
2. Aufruf der Methode *CreatePopupMenu*
3. Einfügen der gewünschten Menüpunkte
4. Aufruf der Methode *TrackPopupMenu*

Um auf das Drücken der rechten Maustaste hin zu reagieren, wird die zu entwerfende Routine in der Methode *OnRButtonDown* der Klasse *CMainFrame* implementiert und über die Message-Map *ON_WM_RBUTTONDOWN* angesteuert:

```
void CChildView::OnRButtonDown(UINT nFlags,
                                CPoint point)
{
    CMenu menu;
    CRect r;
    GetWindowRect(r);
    menu.CreatePopupMenu();
    menu->AppendMenu(MF_STRING, ID_DATEI_OEFFNEN,
                    "Ö&ffnen");
    menu->AppendMenu(MF_STRING, ID_DATEI_SCHLIESSEN,
                    "&Schließen");
```

```

menu->AppendMenu(MF_STRING, ID_DATEI_SUCHEN,
    "S&uchen");
menu.TrackPopupMenu( TPM_LEFTBUTTON |
    TPM_CENTERALIGN,
    point.x+r.left,
    point.y+r.top,
    this, NULL );
}

```

Listing: Variante 1 des Kontextmenüs

Zunächst erfolgt ein Aufruf von *GetWindowRect*, um die Position und Ausdehnung des aktuellen Hauptfensters zu ermitteln und in dem Rechteck-Objekt *r* abzulegen. Diese Information (genauer gesagt: die linke obere Ecke) wird später gebraucht, um das Popup-Menü an der aktuellen Mausposition aufzubauen.

Der anschließende Aufruf der parameterlosen Methode *CreatePopupMenu* erzeugt ein leeres Popup-Menü und ordnet es dem *CMenu*-Objekt *menu* zu. Durch *AppendMenu* werden die einzelnen Menüpunkte so an das leere Menü angehängt, wie es im vorigen Abschnitt beschrieben wurde. Um das auf diese Weise fertiggestellte Menü auf dem Bildschirm anzuzeigen, wird die Methode *TrackPopupMenu* aufgerufen.

```

class Cmenu:
    BOOL TrackPopupMenu(UINT nFlags, int x,
        int y, const CWnd *pWnd,
        LPCRECT lpRect = NULL );

```

In *nFlags* wird eine ODER-Kombination zweier Bitflags erwartet, mit der einerseits die Ausrichtung des Menüfensters und andererseits die gewünschte Maustaste zur Auswahl der Optionen angegeben wird.

Bitflag	Bedeutung
TPM_CENTERALIGN	Horizontale Ausrichtung des Fensters, zentriert um die in <i>x</i> angegebene Koordinate.
TPM_LEFTALIGN	Ausrichtung des Fensters mit der linken Kante auf der in <i>x</i> angegebenen Koordinate.
TPM_RIGHTALIGN	Ausrichtung des Fensters mit der rechten Kante auf der in <i>x</i> angegebenen Koordinate.
TPM_LEFTBUTTON	Die Auswahl der Menüpunkte erfolgt mit der linken Maustaste.
TPM_RIGHTBUTTON	Die Auswahl der Menüpunkte erfolgt mit der rechten Maustaste.

Tabelle 4.2: Bitflags für *TrackPopupMenu*

x und y geben die Position des Menüfensters in Bildschirmkoordinaten an, die Ausrichtung erfolgt gemäß der Bitflags in $nFlags$. Um das Menü wie geplant an der aktuellen Mausposition anzuzeigen, wird die aktuelle Mausposition (die in dem Parameter *point* in Fenster-Koordinaten an `OnRButtonDown` übergeben wurde) zur linken oberen Ecke des Hauptfensters addiert; das Ergebnis ist die Mauszeigerposition in Bildschirmkoordinaten. Der Parameter `pWnd` ist ein Zeiger auf ein Fensterobjekt, das die vom Menü generierten `WM_COMMAND`-Nachrichten erhält; er ist wichtig, um die Verbindung zwischen dem Menü und der Anwendung herzustellen. Jeder Menüpunkt löst bei seinem Aufruf eine `WM_COMMAND`-Nachricht mit dem in *AppendMenu* übergebenen Bezeichner (`ID_DATEI_OEFFNEN`, `ID_DATEI_SCHLIESSEN`, `ID_DATEI_SUCHEN`) aus, die an das Fenster `pWnd` gesendet wird. Soll diese Nachricht im Programm ausgewertet werden, muß dafür ein entsprechender Message-Map-Eintrag erzeugt werden, der für den Aufruf der gewünschten Methode sorgt; beispielsweise wird der Menüpunkt `ÖFFNEN` mit der Methode *OnFileOpen* wie folgt verbunden:

```
void CChildView::OnFileOpen()
{
    MessageBox("Datei Öffnen gedrueckt", "Treffer", MB_OK);
}

...
BEGIN_MESSAGE_MAP(CChildView, CWnd)
    ...
    ON_COMMAND(ID_DATEI_OEFFNEN, OnFileOpen)
END_MESSAGE_MAP()
```

Mit Hilfe des Parameters `lpRect` kann optional ein Fenster angegeben werden, innerhalb dessen das Drücken einer Maustaste *nicht* das Beenden des Menüs bewirkt. Ist der Parameter `NULL`, so wird das Menü beendet, wenn außerhalb dieses Fensters eine Maustaste gedrückt wurde.

In Visual C++ existiert noch eine vereinfachte Methode, Kontext-Menüs zu erzeugen und anzuzeigen. Dazu sind folgende Arbeitsschritte notwendig:

1. Das Anlegen einer Menü-Ressource für das Kontext-Menü mit den zugehörigen Menüpunkten.
2. Das Einfügen eines Message-Handlers für die Nachricht `WM_CONTEXTMENU`.
3. Das Laden des entsprechenden Kontext-Menüs, wobei auch die Cursorposition ausgewertet werden kann, oder das Menü in Abhängigkeit eines bestimmten Objekts geöffnet wird.
4. Der Aufruf der Methode *TrackPopupMenu*.

Die Verknüpfung der rechten Maustaste mit WM_CONTEXTMENU erfolgt automatisch. In der Methode *OnContextMenu* ist die entsprechende Menü-Ressource zu laden. Das Menü besteht aus nur einem Punkt mit dem zugehörigen Popup-Menü, das angezeigt wird.

```
void CChildView::OnContextMenu(CWnd* pWnd, CPoint point)
{
    CMenu ContextMenu;
    ContextMenu.LoadMenu(IDR_KONTEXT);
    if (!bSuchen) //nur die möglichen Menüpunkte anzeigen
    {
        ContextMenu.GetSubMenu(0)->DeleteMenu(1,MF_BYPOSITION);
        ContextMenu.GetSubMenu(0)->DeleteMenu(1,MF_BYPOSITION);
    }
    ContextMenu.GetSubMenu(0)->TrackPopupMenu(TPM_LEFTALIGN|
        TPM_RIGHTBUTTON, point.x, point.y, this);
}
```

Listing: Das Kontextmenü über die Nachricht WM_CONTEXTMENU

Die Koordinaten des Mausklicks werden in *point* übergeben und bestimmen die Position des Popup-Menüs.

Über die IDs der Menüpunkte können Verbindungen zu Message-Handle­rn aufgebaut werden, indem die WM_COMMAND-Nachrichten ausgewertet werden. Wenn Sie den Klassen-Assistenten aufrufen, um die Nachrichten mit den Funktionen zu verbinden, werden Sie gefragt, ob Sie für das Menü eine neue Klasse erstellen wollen (Abbildung 4.8).

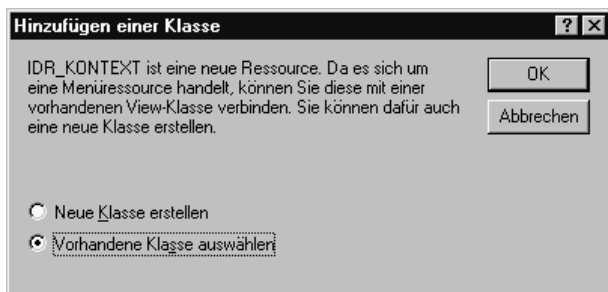


Abbildung 4.8: Das Kontextmenü mit einer vorhandenen Klasse verbinden

Da in den Kontext-Menüs oft die gleichen Funktionen ausgeführt werden sollen wie in »normalen« Menüpunkten, können die gleichen Message-Handler benutzt werden. Die zugehörige Message-Map könnte folgendermaßen aussehen:


```

BEGIN_MESSAGE_MAP(CChildView, CWnd)
    //{{AFX_MSG_MAP(CChildView)
    ...
    ON_WM_CONTEXTMENU()
    ON_COMMAND(ID_K_OEFFNEN, OnDateiOeffnen)
    ON_COMMAND(ID_K_SCHLIESSEN, OnDateiSchliessen)
    ON_COMMAND(ID_K_SUCHEN, OnDateiSuchen)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

```

Listing: Message-Map-Einträge für das Kontextmenü

Alle IDs, die mit ID_K_ beginnen, sind Bestandteile des Popup-Menüs. Diese rufen die gleichen Methoden auf wie das Hauptmenü.

4.3.3 Verändern des Systemmenüs

Das Systemmenü läßt sich genauso bearbeiten, wie ein vom Programm erzeugtes Menü – sofern man ein CMenu-Objekt dafür besitzt. Um es (bzw. einen Zeiger darauf) zu *beschaffen*, gibt es die Methode *GetSystemMenu*.

```

class Cmenu:
    CMenu *GetSystemMenu( BOOL bRevert ) const;

```

Der boolesche Parameter *bRevert* steuert das Verhalten von *GetSystemMenu* auf dramatische Weise: Ist er FALSE, so liefert die Methode wie erwartet den Zeiger auf das CMenu-Objekt, mit dem das Systemmenü bearbeitet werden kann; ist *bRevert* aber TRUE, so ist der Rückgabewert undefiniert (!), und das Systemmenü wurde in seinen originalen Zustand versetzt. Hier hätte Microsoft besser eine zweite Methode mit Namen *ResetSystemMenu* einführen sollen, um die unterschiedliche Funktionalität besser zu trennen. Wie dem auch sei –, Hauptsache man weiß, wie es funktioniert.

Nachdem der Zeiger auf das Systemmenü-Objekt zur Verfügung steht, können dessen Menüpunkte ebenso bearbeitet werden, wie es bei anderen Menüs möglich ist. Als Beispiel sei gezeigt, wie man alle Punkte bis auf MINIMIEREN und MAXIMIEREN löschen und zusätzlich (nach einem Separator) ÜBER SUCHTEXT anhängen kann:

```

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    ...
    CMenu *menu = AfxGetMainWnd()->GetSystemMenu(FALSE);
    for (int i=3; i<5; ++i)
        menu->DeleteMenu(0,MF_BYPOSITION);
    menu->AppendMenu(MF_STRING|MF_SEPARATOR,0,"");
}

```

```

menu->AppendMenu(MF_STRING, ID_APP_ABOUT, "&Über SuchText");
return 0;
}

```

Listing: Veränderungen am Systemmenü in CMainFrame



Die Eigenschaften der Menüpunkte, die aus dem Systemmenü entfernt werden, sind nicht mehr anwählbar! Das betrifft in diesem Fall das Minimieren und Maximieren des Fensters.

Etwas gefährlich ist die Verwendung von *DeleteMenu* mit dem Schalter *MF_BYPOSITION*, weil dabei die Anordnung der Menüpunkte berücksichtigt wird. Man kann aber auch die Punkte des Systemmenüs mit *MF_BYCOMMAND* ansprechen, wenn man die vordefinierten Bezeichner für Systemnachrichten verwendet (*SC_SIZE*, *SC_MOVE*, *SC_MINIMIZE*, *SC_MAXIMIZE*, *SC_CLOSE*, *SC_RESTORE* ... siehe *OnSysCommand* in der Online-Hilfe).

Leider behandelt Windows das Systemmenü bezüglich des Nachrichtenverkehrs etwas anders als die übrigen Menüs. Das Aufrufen des Menüpunkts *Über SuchText* mit dem Bezeichner *ID_APP_ABOUT* führt nicht zum Versenden einer *WM_COMMAND*-, sondern einer *WM_SYSCOMMAND*-Nachricht. Diese wird auch nicht mit einem *ON_COMMAND*-Handler eingefangen, sondern mit Hilfe des Message-Map-Eintrags *ON_WM_SYSCOMMAND*, der zum Aufruf der Methode *OnSysCommand* führt. *OnSysCommand* besitzt zwei Parameter: *nId* ist der Bezeichner des aufgerufenen Punktes, und *lParam* enthält die Cursorposition, falls der Menüpunkt mit der Maus ausgewählt wurde (*x* im Low-Word, *y* im High-Word). Wichtig ist, daß alle übrigen Systemnachrichten weiterhin an den bisherigen *OnSysCommand*-Handler gesendet werden – andernfalls würde das Programm alle Dinge, die in Zusammenhang mit dem Systemmenü stehen, nicht mehr bearbeiten.

Damit kann die Reaktion auf den neuen Über-Punkt folgendermaßen realisiert werden:

```

void CMainFrame::OnSysCommand(UINT nId, LONG lParam)
{
    if (nId == ID_APP_ABOUT)
        MessageBox("SuchText\r\n(C) 1998\r\nN.
Turianskyj",
        "Info über SuchText",
        MB_OK | MB_ICONINFORMATION );
    else
        CFrameWnd::OnSysCommand(nId, lParam);
}

```

Listing: Nachrichtenbehandlung des Systemmenüs

Eine Nachrichtenbehandlungsroutine für die Nachricht `WM_SYSCOMMAND` einzufügen, ist auf den ersten Blick gar nicht so einfach, da die Nachricht nicht im Klassen-Assistenten für `CMainFrame` aufgelistet wird. Die Ursache dafür liegt im Nachrichtenfilter für `CMainFrame`, der auf der Seite *Klassen-Info* eingestellt wird. Wenn Sie diesen von *Oberster Rahmen* auf *Fenster* umstellen, wird auch diese Nachricht angezeigt und kann mit der Methode verbunden werden.

Soll das System-Menü verändert werden, so erledigen Sie das Ändern der Menüpunkte am besten in der Methode `OnInitInstance` oder am Ende von `CMainFrame::OnCreate`. Die Reaktion auf eigene Menüpunkte des System-Menüs muß in `CMainFrame` erfolgen.



Das Beispielprogramm *SuchText* sollte zu diesem Zeitpunkt eine wie oben beschriebene Menüstruktur haben, in der man zwischen langen und kurzen Menüs umschalten kann. Des weiteren wurde das System-Menü geändert. Auf der CD ist dieser Stand der Programmentwicklung unter `SUCHTEXT\STEP1` zu finden.

4.4 Beschleunigertasten

Normalerweise werden Windows-Programme über ein System von Menüs und Untermenüs bedient. Das hat den Vorteil, daß auch unerfahrene Benutzer leicht mit dem Programm umgehen können, da alle Programmfunktionen explizit sind. Der Nachteil einer ausgefeilten und verzweigten Menüstruktur ist, daß oft mehrere Tasten gedrückt und Menüs durchlaufen werden müssen, ehe der gesuchte Programmteil erreicht ist. Dies ist vor allem für erfahrene Benutzer eine Bremse – sie möchten die wichtigsten Programmteile mit *einem einzigen* Tastendruck aufrufen können.

Um beiden Benutzergruppen gerecht zu werden, gibt es das Konzept der Beschleuniger(-tasten) bzw. Accelerator-Table. Hierbei behält ein Programm grundsätzlich seine Menüstruktur, zusätzlich können besonders wichtige Programmteile jedoch direkt über vorgegebene Tastenkürzel aufgerufen werden.

Beschleuniger gibt es auf zwei Ebenen: Zum einen können einzelne Menüpunkte direkt über eine Buchstabentaste angewählt werden, wenn im Menütext ein Zeichen unterstrichen ist; darüber hinaus kann man eine Beschleuniger-Ressource definieren, die einen Menüpunkt direkt (ohne visuelles Aufklappen des Menüs) anwählt. Die erste Alternative wurde bereits erklärt, nun soll gezeigt werden, wie man direkte Beschleuniger definiert. Es können damit auch verborgene Tastenkombinationen definiert werden, um beispielsweise Dialogfenster zur Fehlersuche zu integrieren, die dem »normalen« Anwender verborgen bleiben, oder zur Ausgabe einer versteckten Copyright-Message benutzt werden.

4.4.1 Anlegen einer Beschleuniger-Ressource

Um festzulegen, welches Kommando durch welchen Tastendruck beschleunigt werden soll, muß eine Beschleuniger-Ressource (ACCELERATOR) angelegt werden. Diese Ressource ist eine Tabelle, in der einer ID eine Tastenkombination zugeordnet wird. Hierzu gibt es im Developer Studio einen Editor, der sehr leicht zu bedienen ist.

Falls in der Ressource-Datei noch keine Accelerator-Tabelle definiert ist, muß eine solche neu angelegt werden. Dazu wird über EINFÜGEN|RESOURCE|ACCELERATOR eine neue Tabelle angelegt. Sobald das erfolgt ist, wird im *Editor*-Fenster eine Tabelle zur Definition der Beschleunigertasten gezeigt (siehe Abbildung 4.9).

Durch Markieren einer leeren Zeile in der Tabelle kann über die Eigenschaften eine neue Taste definiert werden. Ein Doppelklick mit der Maus auf diese Zeile hat den gleichen Effekt. Auch hier kann das *Eigenschaften*-Fenster über den Pin festgesetzt werden. Wird der Cursor in eine bereits definierte Zeile bewegt, läßt sich die darin enthaltene Definition ändern.



Abbildung 4.9: Aufbau einer Accelerator-Tabelle

Zuerst wird in der Combobox *ID* der gewünschte Menüpunkt ausgewählt. Nun kann entweder über das Kombinationsfeld *Taste* und die Kontrollkästchen *Zusatzstasten* manuell die gewünschte Zugriffstaste ausgewählt oder – noch einfacher – die Schaltfläche *NÄCHSTE TASTE* gedrückt werden. In diesem Fall legt der nächste Tastendruck die zugeordnete Beschleunigertaste fest. Um den nächsten Beschleuniger zu editieren, klickt man einfach mit der Maus das leere Feld am Ende der Beschleunigertabelle an.

Der Name der Accelerator-Ressource wurde ebenfalls vom Klassen-Assistenten festgelegt. Auch sie heißt IDR_MAINFRAME. Das hat den Vorteil, daß neben dem Menü automatisch die Tabelle mit den Beschleunigertasten geladen wird. Um den Bezeichner zu ändern, muß lediglich der Tabelle über die Eigenschaften () oder rechte Maustaste und *Eigenschaften*) der neue Name zugewiesen werden.

Nach dem Speichern der Daten ist in der RC-Datei ein Abschnitt angelegt worden, der folgendermaßen aussieht:

```
IDR_MAINFRAME ACCELERATORS PRELOAD MOVEABLE PURE
BEGIN
    "C",      ID_EDIT_COPY,      VIRTKEY, CONTROL, NOINVERT
    "C",      ID_DATEI_SCHLIESSEN, VIRTKEY, CONTROL, NOINVERT
    "F",      ID_DATEI_SUCHEN,   VIRTKEY, CONTROL, NOINVERT
    "O",      ID_DATEI_OEFFNEN,  VIRTKEY, CONTROL, NOINVERT
    "V",      ID_EDIT_PASTE,     VIRTKEY, CONTROL, NOINVERT
    VK_BACK,  ID_EDIT_UNDO,      VIRTKEY, ALT, NOINVERT
    VK_DELETE, ID_EDIT_CUT,      VIRTKEY, SHIFT, NOINVERT
    VK_END,   ID_ANSICHT_ENDE,   VIRTKEY, CONTROL, NOINVERT
    VK_F6,    ID_NEXT_PANE,      VIRTKEY, NOINVERT
    VK_F6,    ID_PREV_PANE,      VIRTKEY, SHIFT, NOINVERT
    VK_HOME,  ID_ANSICHT_ANFANG, VIRTKEY, CONTROL, NOINVERT
    VK_INSERT, ID_EDIT_COPY,     VIRTKEY, CONTROL, NOINVERT
    VK_INSERT, ID_EDIT_PASTE,    VIRTKEY, SHIFT, NOINVERT
    "X",      ID_EDIT_CUT,       VIRTKEY, CONTROL, NOINVERT
    "Z",      ID_EDIT_UNDO,      VIRTKEY, CONTROL, NOINVERT
END
```

Listing: Beschleunigertabelle in der Ressource-Datei

Mit Hilfe dieser Tabelle wird beim Übersetzen des Programms eine Beschleuniger-Ressource angelegt und in die Programmdatei eingebunden. Diese Tabelle ist nur automatisch wirksam, wenn sie vom Anwendungs-Assistenten angelegt und in die Dokument-Ansicht-Architektur eingebunden wurde.

4.4.2 Integration einer Beschleuniger-Ressource

Die Definition einer ACCELERATOR-Ressource in der RC-Datei alleine reicht meist noch nicht aus, um über die Beschleunigertasten im Programm zu verfügen. In einem SDK-Programm muß dazu die Beschleunigertabelle geladen und die Nachrichtenschleife zwecks Vorübersetzung der virtuellen Tastencodes unterbrochen werden.

Glücklicherweise geht dies unter den Foundation Classes viel leichter. Man muß lediglich in der Hauptfensterklasse mit der Methode *LoadAccelerTable* die Beschleunigertabelle laden, und alles weitere wird automatisch veranlaßt.

```
CMainFrame::CMainFrame()
{
    LoadAccelerTable(
        MAKEINTRESOURCE(IDR_MAINACCELTABLE));
```

```
Create( NULL, "Hello world",
        WS_OVERLAPPEDWINDOW | WS_VSCROLL |
        WS_HSCROLL, rectDefault, NULL,
        MAKEINTRESOURCE(IDR_MAINMENU)
    );
}
```

Listing: Laden einer Beschleunigertabelle in einer Anwendung

Durch den Aufruf von *LoadAccelTable* wird der Member *m_hAccelTable* der Klasse *CFrameWnd* mit dem Handle der angegebenen Beschleunigertabelle initialisiert, und durch die in *CFrameWnd* realisierte virtuelle Methode *PreTranslateMessage* erfolgt eine automatische Konvertierung der virtuellen Beschleuniger-Tastencodes in WM_COMMAND-Nachrichten. Um die Freigabe der Ressource braucht man sich nicht zu kümmern – das wird automatisch durch den Destruktor von *CFrameWnd* erledigt. Es kann immer nur eine Accelerator-Tabelle geladen sein.

Wurde wie bei *SuchText* die Anwendung mit dem Anwendungs-Assistenten erstellt, wird die Beschleunigertabelle automatisch beim Erstellen des Hauptfensters geladen. Dies erfolgt zum Beispiel in der Funktion *LoadFrame* in der Methode *InitInstance*. Die Ressource-ID des ersten Parameters gilt nicht nur für das Menü, sondern auch für die Beschleunigertabelle.

4.4.3 Anlegen versteckter Beschleuniger

Unter Umständen kann es erforderlich werden, für eine Applikation Tastenkombinationen zu definieren, die nicht an ein Menü gebunden sind. Eine solche Variante bietet sich zum Beispiel zur Suche von Fehlern an. So kann, wenn an einer bestimmten Stelle im Programm eine Taste gedrückt wird, eine Dialogbox geöffnet werden, die Parameter oder Daten anzeigt, die sonst nicht sichtbar sind. Die zweite Anwendungsmöglichkeit könnte sein, den Echtheitsbeweis für ein Programm zu erbringen, wofür aber ist nur eine geheime, unübliche Tastenkombination benötigt wird, die dann eine Meldung in einem Fenster ausgibt. Dazu sind folgende Arbeiten erforderlich:

- ▼ Erstellen eines Eintrags mit dem gewünschten Hotkey in der Accelerator-Tabelle. Dabei muß für *ID* eine neue, noch nicht benutzte ID eingetragen werden.
- ▼ Eintragen einer neuen ON_COMMAND-Message in die Message-Map, in der der Aufruf möglich sein soll.
- ▼ Anlegen der Funktion *OnXxx*, in der die Behandlung der Message erfolgt.

Ein Beispiel dafür könnte wie folgt aussehen:

```
IDR_MAINFRAME ACCELERATORS PRELOAD MOVEABLE PURE
BEGIN
    ...
    "T", ID_COPYRIGHT, VIRTKEY, CONTROL,ALT,SHIFT, NOINVERT
    ...
END
```

In der Accelerator-Tabelle wird die Tastenkombination     definiert, die mit der ID ID_COPYRIGHT versehen wird. Es ist ziemlich ausgeschlossen, diese Kombination unbeabsichtigt im Programm zu drücken. In der Message-Map wird für diese ID eine ON_COMMAND-Message eingetragen. In der zugehörigen Funktion erfolgt die Abarbeitung der Befehle.

```
BEGIN_MESSAGE_MAP(CChildView, CWnd)
    ...
    ON_COMMAND(ID_COPYRIGHT, OnCopyright)
    ...
END_MESSAGE_MAP()

void CChildView::OnCopyright()
{
    AfxMessageBox („Dieses Programm ist
        urheberrechtlich geschützt“,MB_OK);
}
```

Es existieren noch einige weitere Ressourcen wie Icons und Zeichenkettenfolgen, auf die hier nicht näher eingegangen werden soll. Die Benutzung der Ressourcen und die Bedienung des Editors kann intuitiv schnell erlernt werden. Deshalb sollte die Verwendung in der Anwendung kein Problem darstellen.



Einfache Dialoge

5

5.1	Der Dialog Info über ...	178
5.2	Beenden des Programms	180
5.2.1	Beenden von Windows	182
5.3	Standard-Dialoge	183
5.3.1	Die verschiedenen standardisierten Dialoge	184
5.3.2	Vorgehensweise bei der Programmierung	185
5.3.3	Erweiterungsmöglichkeiten	186
5.4	Der Dialog Datei öffnen in SuchText	187
5.5	Der Suchen-Dialog	190
5.5.1	Implementierung in das Programm	191
5.5.2	Nachrichtenbehandlung des Dialogs	193
5.5.3	Implementierung von OnSuchen	194
5.6	Verändern der Programm-Titelleiste	197

5.1 Der Dialog *Info über ...*

Unter Windows hat es sich eingebürgert, Programme mit Informationen über sich selbst zu versehen. Der Benutzer hat zumeist die Möglichkeit, eine Dialogbox zu öffnen, in der Informationen wie Programmname, Version, Copyright und Autor angezeigt werden. Außer einem Button besitzt ein solcher *Info über*-Dialog keine weiteren Steuerungen. Bei so einfachen Dialogen ist es nicht unbedingt erforderlich, eine eigene Dialogboxklasse anzulegen, wie in einem folgenden Abschnitt beschrieben; statt dessen kann auch die Methode *MessageBox* der Klasse *CWnd* verwendet werden.

```
class Cwnd:
    int MessageBox( LPCTSTR lpszText,
                   LPCTSTR lpszCaption = NULL,
                   UINT nType = MB_OK );
```

MessageBox erzeugt nach seinem Aufruf ein Fenster mit einem oder mehreren Buttons und einem erläuternden Text. Die Titelzeile des Fensters bekommt den in *lpszCaption* übergebenen Text, während innerhalb des Arbeitsbereichs die in *lpszText* übergebene Information ausgegeben wird. Bei beiden Parametern handelt es sich um nullterminierte Zeichenketten, wie sie in C üblich sind. Mit Hilfe des dritten Parameters *nType* (siehe *MessageBox* in der Online-Hilfe) werden die optischen Eigenschaften des Fensters festgelegt. Eine Auswahl der Möglichkeiten zeigt Tabelle 5.1; die einzelnen Werte können unter Verwendung der bitweisen ODER-Verknüpfung miteinander kombiniert werden. Dabei kann aus jeder Kategorie ein Parameter gewählt werden.

Wert	Bedeutung	Rückgabewert
Message-Box-Typen		
MB_OK	Die Box erhält einen OK-Button.	IDOK
MB_OKCANCEL	Die Box erhält einen OK- und einen CANCEL-Button.	IDOK oder IDCANCEL
MB_YESNO	Die Box erhält einen YES- und einen NO-Button.	IDNO oder IDYES
MB_YESNOCANCEL	Die Box enthält die Buttons YES, NO und CANCEL.	IDYES, IDNO oder IDCANCEL
MB_RETRYCANCEL		IDRETRY oder IDCANCEL
MB_ABORTRETRYIGNORE		IDABORT, IDRETRY oder IDIGNORE

Wert	Bedeutung	Rückgabewert
Message-Box-Icons		
MB_ICONSTOP		
MB_ICONQUESTION		
MB_ICONINFORMATION		
MB_ICONEXCLAMATION		
Message-Box-Standard-button		
MB_DEFBUTTON1	Erster Button wird bei Enter ausgelöst (Standard).	
MB_DEFBUTTON2	Zweiter Button wird bei Enter ausgelöst.	
MB_DEFBUTTON3	Dritter Button wird bei Enter ausgelöst.	
Message-Box-Modalität		
MB_APPMODAL	Die Applikation wird erst nach Ende der Message-Box fortgesetzt (Standard).	
MB_SYSTEMMODAL	Alle Applikationen des Betriebssystems werden angehalten.	
MB_TASKMODAL	Ähnlich APPMODAL, jedoch für Applikationen gedacht, die kein Window-Handle besitzen.	

Tabelle 5.1: Der Parameter `nType` von `MessageBox`

Das Erzeugen eines *Info über*-Dialogs erfordert es normalerweise, mehrere Zeilen Text darzustellen. Dazu kann `lpszText` Zeilenschaltungen enthalten, die in der Form »`\r\n`« (also als Binärcodes `0x0D`, gefolgt von `0x0A`) in den Text eingeflochten werden. Diese werden von `MessageBox` korrekt als Zeilenumbrüche interpretiert und dargestellt. Leider bietet `MessageBox` keine weitergehenden Formatierungsmöglichkeiten, beispielsweise um den Text zentriert auszugeben oder die Schriftart zu verändern. Die Ausgabe erfolgt immer in Systemschrift und immer linksbündig.

In *SuchText* wurde der *Über Suchtext*-Dialog in die Methode `OnSysCommand` eingebaut. Alternativ kann auch die Methode `AfxMessageBox` verwendet werden.

```
MessageBox("SuchText\r\n(C) 1998\r\nN. Turianskyj",
           "Info über SuchText",
```

```

        MB_OK | MB_ICONINFORMATION );
    AfxMessageBox("SuchText\r\n(C) 1998\r\nN. Turianskyj",
        MB_OK | MB_ICONINFORMATION );

```

```

int AfxMessageBox( LPCTSTR lpszText,
    UINT nType = MB_OK,
    UINT nIDHelp = 0 );
int AFXAPI AfxMessageBox( UINT nIDPrompt,
    UINT nType = MB_OK,
    UINT nIDHelp = (UINT) -1 );

```

Diese Methode besitzt keinen Text für den Titel, dafür aber eine Hilfe-ID. Der Titel wird automatisch aus dem Namen der Applikation gebildet. Stimmt auch der Typ der Message-Box überein (MB_OK), so muß nur noch der anzuzeigende Text als Parameter übergeben werden.

Alle mit dem Anwendungs-Assistenten erstellten Anwendungen (MFC-EXE) erhalten automatisch einen *Info über*-Dialog. Die Anwendung von *MessageBox* bzw. *AfxMessageBox* ist deshalb nur beispielhaft beschrieben. Die Methoden finden oft bei Fehlerausschriften oder Hinweisen an den Benutzer Anwendung.

5.2 Beenden des Programms

Das Schließen eines Fensters wurde bereits im Kapitel »Zerstören eines Fensters« behandelt: Das Beenden der Anwendung erfolgt genau dann, wenn das Hauptfenster geschlossen wird. Vor dem Schließen des Hauptfensters erfolgt ein Aufruf der Methode *OnClose*, in der dann standardmäßig (*OnClose* ist eine Methode von *CWnd*) durch einen Aufruf von *DestroyWindow* alles weitere erledigt wird.

In vielen Fällen darf ein Programm jedoch nicht ohne weiteres beendet werden; möglicherweise sind noch Daten zu sichern oder andere wichtige Tätigkeiten zu erledigen. Außerdem kann es erforderlich werden, mit dem Benutzer noch einen kurzen Dialog zu führen, nach dem Muster »Wollen Sie wirklich beenden?« oder »Sollen die Änderungen gesichert werden?«. Um dies zu demonstrieren, wird in der Hauptfensterklasse von *SuchText* die Methode *OnClose* überlagert:

```

void CMainFrame::OnClose()
{
    if(AfxMessageBox("Programm Beenden ?", MB_YESNO)==IDYES)
        CFrameWnd::OnClose();
}

```

Listing: Behandeln der Nachricht WM_CLOSE

Es hat wenig Sinn, den Menüpunkt **BEENDEN** im **DATEI**-Menü zur Abfrage zu benutzen. Zwar funktioniert dies auch, aber nur dann, wenn der Anwender das Programm auch über das Menü beendet.



Wie fast alle überlagerten Fenster-Methoden wird auch *OnClose* erst dann aktiv, wenn zusätzlich ein passender Eintrag in die Message-Map aufgenommen wird. Dazu wird mit dem Klassen-Assistenten ein entsprechender Eintrag für die Nachricht **WM_CLOSE** der Klasse *CMainFrame* erstellt.

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    //{{AFX_MSG_MAP(CMainFrame)
    ...
    ON_WM_CLOSE()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Im Gegensatz zu den **ON_COMMAND**-Einträgen ist das **ON_WM_CLOSE**-Makro parameterlos; insbesondere kann also nicht der Name einer Methode angegeben werden, die bei Auftreten der **WM_CLOSE**-Nachricht aufgerufen wird. Vielmehr erwartet **ON_WM_CLOSE** eine parameterlose *void*-Methode, die exakt den Namen *OnClose* besitzt.

OnClose wird genau dann aufgerufen, wenn der Benutzer eine der folgenden Aktionen durchgeführt hat:

- ▼ Er hat die Tastenkombination **[Alt] [F4]** gedrückt.
- ▼ Er hat den *Schliessen*-Button der Titelseite gedrückt.
- ▼ Er hat einen Doppelklick auf dem *System*-Menü ausgeführt.
- ▼ Er hat die Option *Schließen* des *System*-Menüs ausgewählt.
- ▼ Er hat im Menü *Datei* den Punkt *Beenden* gewählt.

Durch die Überlagerung von *OnClose* wird nun nicht mehr automatisch *DestroyWindow* aufgerufen, sondern der Benutzer wird in einer Message-Box (siehe Kapitel »Beenden von Windows«) gefragt, ob er das Programm wirklich beenden will. Bestätigt er dies durch Drücken der JA-Schaltfläche, so wird *OnClose* der Basisklasse *CFrameWnd* aufgerufen und das Programm beendet, andernfalls geschieht dies nicht, und das Programm wird normal fortgesetzt.

Durch diese Maßnahmen ist sichergestellt, daß alle Möglichkeiten des Benutzers, das Programm zu beenden, korrekt abgefangen werden. Und was noch besser ist: Die Endebehandlung erfolgt an einer einzigen Stelle im Programm. Diese Vorgehensweise ist natürlich nicht mehr ohne weiteres möglich, wenn das Programm unterscheiden muß, welche der vier Möglichkeiten zum Aufruf von *OnClose* geführt hat. In diesem Fall muß man

die Nachrichten entweder durch unterschiedliche Methoden bearbeiten oder mit Hilfe der Funktion *GetCurrentMessage* versuchen, weitere Informationen zu beschaffen.

5.2.1 Beenden von Windows

Das Beenden von Windows ist in zweierlei Hinsicht interessant: Einerseits muß festgestellt werden, ob eine laufende Applikation noch Daten sichern muß, bevor sie beendet werden kann, andererseits kann es vorkommen, daß vom eigenen Programm aus Windows beendet werden soll. In jedem Fall sieht der Ablauf so aus:

1. Windows verschickt die Nachricht *WM_QUERYENDSESSION* an alle aktiven Anwendungen und erwartet einen booleschen Rückgabewert, der das Beenden von Windows erlaubt.
2. In der Applikation wird zuerst die Methode *OnQueryEndSession* aufgerufen. Danach wird *OnClose* gerufen.
3. Wurden alle Applikationen nach Punkt 1 und 2 abgearbeitet und haben das Beenden von Windows erlaubt, wird die Nachricht *WM_ENDSESSION* verschickt.
4. Windows übergibt der Nachricht *WM_ENDSESSION* den Parameter *wParam* vom Typ *BOOL*, der angibt, ob die aktuelle Sitzung beendet oder Windows heruntergefahren werden soll.

In der eigenen Applikation kann auch noch die Nachricht *WM_QUERYENDSESSION* überlagert werden. Damit kann man durch den entsprechenden Rückgabewert (*FALSE*) verhindern, daß Windows beendet wird.

```
afx_msg void OnQueryEndSession( BOOL bEnding );
```

Für das Beenden von Windows aus eigenen Programmen heraus existieren zwei Funktionen:

```
BOOL ExitWindows(   DWORD dwReserved,  
                   UINT uReserved );  
BOOL ExitWindowsEx( UINT uFlags,  
                   DWORD dwReserved )
```

Die Methode *ExitWindows* beendet die aktuelle Sitzung, ohne Windows herunterzufahren. Die beiden Parameter müssen mit 0 angegeben werden. Falls ein anderes Programm das Beenden verhindert, wird *FALSE* zurückgegeben.

Die zweite Variante erlaubt durch Angabe eines Parameters (uFlags) die Unterscheidung, ob nur die Sitzung beendet oder auch Windows heruntergefahren werden soll. Eine Aufstellung der Flags ist in Tabelle 5.2 zu sehen. Der zweite Parameter wird zur Zeit nicht benutzt und ignoriert. Der Rückgabewert entspricht der Bedeutung von *ExitWindows*. Weitere Informationen dazu sind in der Online-Dokumentation unter dem Stichwort *ExitWindows* zu finden.

Parameter für uFlags	Bedeutung
EWX_FORCE	Fordert die Beendigung von Prozessen nach einer gewissen Zeit, wenn vom Benutzer keine Entscheidung getroffen wurde (Task beenden).
EWX_FORCEIFHUNG	Diese Option ist erst ab Windows NT 5.0 verfügbar. Sie fordert die Beendigung von Prozessen, die nicht auf WM_QUERYENDSESSION oder WM_ENDSESSION reagieren. Bei Benutzung von EWX_FORCE wird diese Flag ignoriert.
EWX_LOGOFF	Der Benutzer wird abgemeldet und die Sitzung beendet.
EWX_POWEROFF	Windows wird heruntergefahren, und der Computer wird ausgeschaltet. Voraussetzung sind entsprechende Hardware und die Rechte dafür.
EWX_REBOOT	Windows wird heruntergefahren und neu gestartet. Auch hier sind entsprechende Rechte erforderlich.
EWX_SHUTDOWN	Windows wird heruntergefahren und beendet. Der PC kann ausgeschaltet werden.

Tabelle 5.2: Parameter für ExitWindowsEx

Bei Windows NT ist es erforderlich, die entsprechenden Rechte zu haben, um ein Shutdown oder Restart auszulösen. Auf der Begleit-CD finden Sie ein Beispielprogramm dazu.

5.3 Standard-Dialoge

Neben den Einfachst-Dialogen, die durch Aufruf von *MessageBox* erzeugt werden, gibt es noch eine weitere Klasse, die ohne das Erzeugen von Dialogboxklassen und -schablonen auskommt: die Standard-Dialoge (in der SDK-Dokumentation *common dialogs* genannt). Sie realisieren einige der am häufigsten wiederkehrenden Interaktionen mit dem Anwender, wie sie beispielsweise vor dem Öffnen oder Speichern einer Datei, beim Einstellen der Farben oder dem Ausdrucken von Daten erforderlich sind. Zwar gehören diese Dialoge nicht zu den Microsoft Foundation Classes, sondern sind Bestandteil des normalen SDK, nützlich sind sie aber auch für den MFC-Programmierer, und sie lassen sich unter C++ ebenso gut verwenden wie unter C.

Während unter Windows 3.0 jeder Entwickler seine eigenen Kreationen entwarf, gibt es seit Windows 3.1 die dynamische Link-Library *COMMDDL* bzw. *COMDLG32*, die alle Funktionen enthält, um diese (teil-

weise sehr komplexen) Dialoge zu realisieren. Die Verwendung dieser Bibliothek hat für den Programmierer den Vorteil, daß er nicht ständig das Rad neu erfinden muß, während der Anwender es schätzen wird, nicht von Applikation zu Applikation die einfachsten Tätigkeiten neu erlernen zu müssen. Die COMMDLG.DLL gehört zu den Bestandteilen des SDK und seit Windows 3.1 auch zum Betriebssystem. Die Unterstützung von Standard-OLE-Dialogen erfolgt über die Datei OLEDLG.DLL. Auf die Beschreibung der Klassen wird in diesem Buch verzichtet.

Nachdem zunächst die allgemeine Vorgehensweise, die bei der Programmierung aller Standard-Dialoge anzuwenden ist, erklärt wurde, zeigt der letzte Teil dieses Abschnitts die konkrete Vorgehensweise für den *Datei Öffnen*- und den *Suchen*-Dialog des *SuchText*-Programms.

5.3.1 Die verschiedenen standardisierten Dialoge

Insgesamt gibt es acht standardisierte Dialoge:

1. *Farben*: Auswählen von Farbeinstellungen aus der Menge der auf dem System verfügbaren Farben. Zuweisen von Farben an Dialogbox-Kontrollen.
2. *Schriften*: Erlaubt das Auswählen aus den vorhandenen Schriften in den verfügbaren Größen. Nach jeder Auswahl wird ein Beispieltext in der aktuellen Schriftart und -größe angezeigt.
3. *Datei öffnen*: Zeigt eine Liste von Dateien, die bestimmte Datei-Erweiterungen haben. Der Benutzer kann dann interaktiv aus diesen Dateien auswählen, Verzeichnisse oder Laufwerke wechseln oder einen Dateinamen direkt eingeben.
4. *Datei unter einem bestimmten Namen speichern*: Funktioniert ähnlich wie das Öffnen einer Datei und dient dazu, einen Namen zum Speichern einer Datei zu ermitteln. *Datei öffnen* und *schließen* benutzen die gleiche Klasse.
5. *Drucken*: Zeigt Informationen über den aktuellen Drucker an und erlaubt es dem Benutzer, bestimmte Einstellungen für die Druckausgabe zu verändern.
6. *Druckerkonfiguration*: Zeigt die Liste der verfügbaren Drucker an und erlaubt die Auswahl eines Druckers daraus. Zusätzlich können die Eigenschaften des ausgewählten Druckertreibers verändert werden.
7. *Suchen*: Erlaubt die Eingabe einer Such-Zeichenkette und einiger Optionen zur Steuerung des Suchvorgangs. Dient dazu, eine Suche (beispielsweise in einer Textdatei) zu starten.

8. *Ersetzen*: Ähneln dem *Suchen*-Dialog, erlaubt aber zusätzlich die Eingabe einer zweiten Zeichenkette, welche als Ersatz für die erste verwendet werden soll. Hat die Aufgabe, die zum Suchen und Ersetzen erforderlichen Informationen zu beschaffen. Auch hier wird für beide Arten die gleiche Klasse benutzt.

Es ist wichtig zu wissen, daß die DLL nur die halbe Arbeit erledigt, die andere Hälfte obliegt nach wie vor dem Programmierer. Am Beispiel des *Datei öffnen*-Dialogs bedeutet dies: Der Standard-Dialog unterstützt zwar die Auswahl eines Dateinamens, indem er interaktives Suchen in Unterverzeichnissen ermöglicht, die Eingabe von Wildcards unterstützt oder den Wechsel des Laufwerks vorsieht – als Ergebnis gibt es aber immer nur einen Dateinamen. Das Öffnen der Datei und gegebenenfalls das Einlesen ihres Inhalts muß man dann immer noch selbst erledigen.

Alle Dialoge für OLE-Funktionen sind von der Klasse *COleDialog* abgeleitet. Weitere Informationen sind in der Klassenhierarchie im Anhang dieses Buches zu finden und in der Online-Hilfe unter *COleDialog*.

5.3.2 Vorgehensweise bei der Programmierung

Die Programmierung der Standard-Dialoge ist zwar nicht vollkommen trivial, geht aber systematisch und einheitlich vonstatten und gestaltet sich, verglichen mit dem Aufwand, den man betreiben müßte, um einen gleichwertigen Dialog selber herzustellen, noch recht einfach.

Innerhalb der MFC erfolgt eine Kapselung der Standard-Dialoge. Es ist aber auch weiterhin möglich, ohne die MFC-Klassen direkt mit den Funktionen der DLL zu arbeiten. Für den Programmierer bedeuten die MFC-Klassen eine Vereinfachung der Programmierung, aber zum Teil auch eine Verringerung der Flexibilität gegenüber Veränderungen.

Wenn die folgenden Regeln beachtet werden, kann eigentlich nichts mehr schiefgehen:

1. Das Anlegen einer neuen Instanz der Klasse des Standard-Dialogs (am besten auf dem Heap mit dem Operator *new*).
2. Im Konstruktor der Klasse sind alle nötigen Parameter anzugeben. Für die meisten können die Standardwerte übernommen werden, so daß nur wenige Parameter zwingend notwendig sind.

Jede der Klassen enthält eine öffentliche Member-Variable (z.B. *m_ofn* bei *CFileDialog*), die den SDK-Datenstrukturen *OPENFILENAME*, *CHOOSECOLOR*, *PRINTDLG*, *CHOOSEFONT* und *FINDREPLACE* entsprechen, welche in *COMMDLG.H* definiert sind. In diesen Datenstrukturen werden einerseits Input-Parameter für den Aufruf der Dialogfunktion übergeben, andererseits liefert die Dialogfunktion auch

Werte über sie zurück. In der objektorientierten Programmierung ist es jedoch üblich, Member-Funktionen für den Zugriff auf die Daten zu benutzen.

3. Es erfolgt der Aufruf der Member-Funktion *DoModal*, um die Dialogbox anzuzeigen und auf die Eingaben des Benutzers zu warten. Da der *Suchen/Ersetzen*-Dialog kein modaler Dialog ist, muß dieser anders behandelt werden. Dort wird *Create* aufgerufen, um ihn anzuzeigen.
4. Der Rückgabewert von *DoModal* wird ausgewertet. Es wird entweder IDOK oder IDCANCEL zurückgegeben, je nachdem, welchen Button der Benutzer zum Verlassen des Dialogs betätigt hat. Wenn der Rückgabewert 0 ist, kann die spezielle Funktion *CommDlgExtendedError* aufgerufen werden. Sie liefert ein DWORD zurück, in dem der zuletzt aufgetretene Fehler näher bezeichnet wird.
5. Es werden die entsprechenden Werte geholt, die der Benutzer gewählt hat.
6. Falls der Dialog mit *new* auf dem Heap angelegt wurde, muß er mit *delete* wieder zerstört werden!

5.3.3 Erweiterungsmöglichkeiten

Mit den angesprochenen Eigenschaften sind die Fähigkeiten der Standard-Dialoge noch nicht erschöpft, vielmehr lassen sich die Dialoge auf verschiedene Art erweitern oder verändern; die Möglichkeiten reichen von Modifikationen bzw. Austausch der Dialogboxschablone bis zu Veränderungen am Dialogbox-Code, der mit Hilfe von Hook-Funktionen ausgeführt wird. In allen Fällen hat dies zwei Konsequenzen, die man als Programmierer beachten sollte:

- ▼ Einerseits verschlechtern Modifikationen der Standard-Dialogboxen unter Umständen das Aussehen und die Bedienung; gerade diese sollen aber mit Hilfe der Standard-Dialoge verbessert werden. Unter diesem Aspekt sollten Veränderungen nur dann eingeführt werden, wenn sie absolut unumgänglich sind.
- ▼ Andererseits gestaltet sich das Verändern der Standard-Dialoge für den Programmierer sehr aufwendig und bei weitem umfangreicher, als das Ausfüllen einer Struktur mit anschließendem Aufruf einer einzigen Funktion. Auch aus diesem Grund sollte man sich überlegen, ob der Aufwand durch das angestrebte Ergebnis gerechtfertigt wird.

Hier wird nicht weiter auf die Vorgehensweise bei der Veränderung von Standard-Dialogen eingegangen.

5.4 Der Dialog *Datei öffnen* in *SuchText*

In *SuchText* wird der Standard-Dialog zum Öffnen einer Datei verwendet, um den Namen der zu durchsuchenden Datei zu ermitteln – wohlge-merkt, den Namen; das Öffnen selbst muß manuell ausgeführt werden. Das endgültige Aussehen des Dialogs kann der Abbildung 5.1 entnommen werden, dessen Funktionalität wird als bekannt vorausgesetzt.



Abbildung 5.1: Der Dialog »Datei öffnen«

Im folgenden Beispiel wird das Anlegen und Aktivieren des *Datei öffnen*-Dialogs dargestellt. Zusätzlich werden in Tabelle 5.3 einige der Variablen der OPENFILENAME-Struktur erläutert.

Der zugehörige Auszug aus den Listings befindet sich in der Methode *OnDateiOeffnen* von *CChildView*, die (über einen entsprechenden Message-Map-Eintrag) durch den Menüpunkt *DATEI|ÖFFNEN* aktiviert wird:

```
void CChildView::OnDateiOeffnen()
{
    CString szFilter = "C++-Dateien (*.CPP)|*.cpp|
                      DXF-Datei (*.DXF) |*.dxf|
                      Alle Dateien (*.*) |*.*||";
    CString defName = "*.CPP";
    CStdioFile fi;
    CString sFileName;
    CFileDialog *mDateiWahl;
    mDateiWahl = new CFileDialog(TRUE, NULL,
                                defName, OFN_HIDEREADONLY, szFilter);
    mDateiWahl->m_ofn.lpstrTitle =
        "SuchText Datei wählen ...";
    if (mDateiWahl->DoModal() == IDOK)
```

```

    {
        sFileName = mDateiWahl->GetPathName();
        if ( !fi.Open(sFileName,CFile::modeRead))
        {
            AfxMessageBox("Fehler beim Öffnen der
                           Datei!",MB_OK|MB_ICONSTOP);
        }
        else
        {
            //Datei verarbeiten
        }
    }
    delete mDateiWahl;
}

```

Listing: Öffnen einer Datei über den Standard-Dialog »Datei öffnen«

Zunächst werden zwei Zeichenketten-Variablen angelegt, die den Dateifilter und den Standard-Dateinamen enthalten. Die Variable `sFileName` dient zur Aufnahme des Dateinamens, den der Benutzer gewählt hat. Zum Lesen der Datei wird die Klasse *CStdioFile* benutzt, die den einfachen Zugriff auf Textdateien mittels der Member-Funktion *ReadString* ermöglicht (siehe Kapitel 11.5.5). Schließlich wird eine Instanz von *CFileDialog* angelegt, und im Konstruktor werden die notwendigen Parameter übergeben.

```

CFileDialog( BOOL bOpenFileDialog,
             LPCTSTR lpszDefExt = NULL,
             LPCTSTR lpszFileName = NULL,
             DWORD dwFlags = OFN_HIDEREADONLY |
                             OFN_OVERWRITEPROMPT,
             LPCTSTR lpszFilter = NULL,
             CWnd* pParentWnd = NULL );

```

In *bOpenFileDialog* wird festgelegt, ob es ein Dialog zum Öffnen oder zum Speichern einer Datei sein soll. Einige der Parameter sind direkt in der Struktur *OPENFILENAME* zu finden. Die Erklärung der wichtigsten Flags, die das Verhalten und Aussehen der Dialogbox beeinflussen, erfolgt in Tabelle 5.3.

Nach dem Erzeugen der Instanz von *CFileDialog* kann auch auf die Member-Variable `m_ofn` zugegriffen werden. So wird zum Beispiel die Titelzeile des Dialogs verändert. Mit dem Aufruf von *DoModal* wird die Box angezeigt und der Benutzer kann die Datei auswählen. Durch Drücken von *ÖFFNEN* oder *ABBRECHEN* wird die Dialogbox wieder vom Bildschirm entfernt. Jetzt erfolgt die Auswertung des Rückgabewertes von *DoModal*.

Wurde IDOK zurückgegeben, kann mit dem Öffnen und Einlesen der Datei begonnen werden. Dabei werden zuerst über die Member-Funktion *GetPathName* der gewählte Pfad und Dateiname ermittelt und auf ihr Vorhandensein hin überprüft. Zum Schluß wird die Instanz von *CFileDialog* wieder entfernt.

Flag	Bedeutung
OFN_CREATEPROMPT	Der Benutzer wird in einer Dialogbox nach dem Überschreiben einer vorhandenen Datei gefragt.
OFN_FILEMUSTEXIST OFN_PATHMUSTEXIST	Es dürfen nur Dateinamen und Pfade eingegeben werden, die wirklich existieren. Damit wird dem Programmierer der Test auf Vorhandensein einer Datei abgenommen.
OFN_HIDEREADONLY	Verbirgt die Checkbox »Mit Schreibschutz öffnen«.
OFN_OVERWRITEPROMPT	Beim Speichern-Dialog wird vor dem Überschreiben einer vorhandenen Datei nachgefragt.
OFN_SHOWHELP	Zeigt zusätzlich einen Hilfe-Button an, der eine Verknüpfung zur Online-Hilfe beinhaltet.
OFN_NONETWORKBUTTON	Zeigt in Netzwerkumgebungen den Button NETZWERK nicht an.

Tabelle 5.3: Einige Flags in CFileDialog

Member	Bedeutung	Ein-/Ausgabe
lStructSize	Bestimmt die Größe der Struktur in Byte.	E
hwndOwner	Bezeichnet das Fenster, welches Besitzer der Dialogbox wird. Hier darf auch NULL angegeben werden, wenn die Dialogbox keinen Besitzer haben soll. Am einfachsten ist es, den Handle <i>m_hWnd</i> des aktuellen Fensters anzugeben.	E
lpstrFilter	Zeigt auf einen Puffer, der Paare nullterminierter Strings enthält, die zur Spezifikation der erlaubten Datei-Erweiterungen dienen. Jedes Paar besteht aus einer verbalen Beschreibung, die dem Anwender sichtbar gemacht wird, und einer zugehörigen Dateierweiterung *.xyz, die von der Dialogboxprozedur verwendet wird. Das letzte Paar muß mit einem Doppelnullbyte enden, damit die Funktion das Ende erkennen kann – aus diesem Grund die \0 am Ende der Stringkonstanten. Es ist möglich, mehrere Erweiterungen bei einem Element anzugeben, in diesem Fall sind die einzelnen Erweiterungen durch »;« zu trennen (z.B. *.txt;*.doc;*.ini).	E
nFilterIndex	Gibt an, welche der in <i>lpstrFilter</i> angegebenen Erweiterungen unmittelbar nach dem Aufruf der Box aktiviert werden soll. Das erste Element hat den Index 1.	E
lpstrDefExt	Gibt an, welche Datei-Erweiterung verwendet werden soll, wenn der Benutzer keine eigene angegeben hat. Falls dieser Parameter NULL ist, wird der Dateiname nicht automatisch mit einer Erweiterung versehen.	E

Tabelle 5.4: Die wichtigsten Member von OPENFILENAME

Member	Bedeutung	Ein-/Ausgabe
lpstrFile	Zeigt auf einen Puffer, der den Inhalt des Edit-Feldes (die Vorgabe für den Dateinamen) unmittelbar nach dem Aufruf der Dialogbox angibt. Nach dem erfolgreichen Ende der Funktion steht in diesem Puffer der komplette Name der zu öffnenden Datei. Falls der Puffer zu klein für den vom Benutzer ausgewählten Dateinamen ist, liefert die Funktion FALSE und <i>CommDlgExtendedError</i> den Wert FNERR_BUFFERTOOSMALL.	E / A
nMaxFile	Gibt die Größe des Puffers <i>lpstrFile</i> an. Der Puffer sollte 256 Zeichen lang sein.	E
lpstrTitle	Gibt den Titeltext für die Dialogbox an.	E
Flags	Gibt die Initialisierungsflags für die Dialogbox an. OFN_HIDEREADONLY versteckt beispielsweise den Read-only-Check-Button.	E

Tabelle 5.4: Die wichtigsten Member von OPENFILENAME

5.5 Der Suchen-Dialog

In *SuchText* wird der Standard-Dialog zum Suchen von Text verwendet. Dabei wird der zu suchende Text vom Benutzer eingegeben und über eine Schaltfläche der Suchvorgang gestartet. Die Dialogbox stellt nur den Standard-Dialog zur Verfügung – die eigentliche Suchfunktion muß der Programmierer selbst schreiben. Das Aussehen des *Such*-Dialogs kann der Abbildung 5.2 entnommen werden, die Funktionalität wird als bekannt vorausgesetzt.

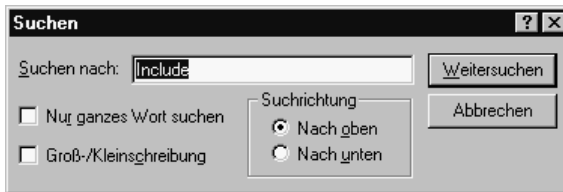


Abbildung 5.2: Der Standard-Dialog zum Suchen

In Abbildung 5.3 sehen Sie den Dialog aus *SuchText*, in dem zusätzliche Styles zum Ausblenden nicht benötigter Funktionen gesetzt wurden.



Abbildung 5.3: Der Such-Dialog in SuchText

5.5.1 Implementierung in das Programm

Der *Suchen*-Dialog unterscheidet sich grundlegend vom *Datei-öffnen*-Dialog. Wenn der Anwender eine Datei öffnen will, kann er keine anderen Aktionen im Programm anwählen, solange der *Öffnen*-Dialog aktiv ist. Dieser Dialog ist modal. Beim Suchen von beispielsweise Text in Winword können Sie weiterhin Menüpunkte auswählen oder im Text editieren, wenn der *Suchen*-Dialog geöffnet ist. Das wird dadurch möglich, daß dieser Dialog nicht modal ist. Das Erzeugen eines nicht-modalen Dialogs erfolgt auch auf andere Weise (siehe auch Kapitel 12.7). Zuerst wird ein Zeiger auf eine Variable von Typ *CFindReplaceDialog* angelegt. Anschließend wird der Dialog mit der Member-Funktion *Create* erzeugt und auf dem Bildschirm dargestellt.

```
void CChildView::OnDateiSuchen()
{
    pFR = new CFindReplaceDialog();
    pFR->Create(TRUE,"include",
        NULL,FR_HIDEWHOLEWORD | FR_HIDEUPDOWN,this);
}
```

Listing: Aufruf eines Suchen-Dialogs

Damit ist die Implementierung der Funktion *OnDateiSuchen*, die durch Aufruf des Menüpunktes DATEI|SUCHEN oder über das Kontext-Menü erfolgt, bereits abgeschlossen. Jedoch wird zu diesem Zeitpunkt durch Drücken auf die Schaltfläche WEITERSUCHEN noch keine Aktion ausgelöst.

Die Klasse *CFindReplaceDialog* besitzt einige Methoden und einen Daten-Member *m_fr*, der auf eine FINDREPLACE-Struktur verweist. Diese Struktur dient ähnlich wie bei *Datei öffnen* zur Aufnahme der Parameter des Dialogs und wird standardmäßig in der SDK-Programmierung benutzt. Die Klasse *CFindReplaceDialog* kapselt diese Struktur und ermöglicht den Zugriff darauf über Methoden der Klasse.

```
class CFindReplaceDialog
{
    CFindReplaceDialog( );
    BOOL Create(BOOL bFindDialogOnly,
        LPCTSTR lpszFindWhat,
        LPCTSTR lpszReplaceWith = NULL,
        DWORD dwFlags = FR_DOWN,
        CWnd* pParentWnd = NULL );
}
```

Der Konstruktor wird parameterlos aufgerufen. Der erste Parameter in der Methode *Create* dient zur Unterscheidung, ob nur der *Suchen*-Dialog angezeigt wird (TRUE), oder ob *Suchen und Ersetzen* erfolgen soll. Die nächsten

beiden Zeichenketten dienen zur Vorgabe der *Suchen*- und der *Ersetzen*-Zeichenfolge. Die in `dwFlags` abgelegten Bitflags bestimmen das Aussehen des Dialogs. Eine Aufstellung der Flags ist in Tabelle 5.5 zu finden. Schließlich wird noch ein Zeiger auf ein Fenster übergeben, das die Nachrichten des Dialogs empfangen soll.

```
class CFindReplaceDialog
{
public:
    static CFindReplaceDialog* PASCAL
        GetNotifier( LPARAM lParam );

    CString GetFindString( ) const;
    BOOL FindNext( ) const;
    BOOL IsTerminating( ) const;
    BOOL MatchCase( ) const;
    BOOL SearchDown( ) const;
};
```

In der vorangehenden Aufstellung sind weitere Member-Funktionen der Klasse *CFindReplaceDialog* aufgeführt. Mit Hilfe von *GetNotifier* kann man sich zum Beispiel einen Zeiger auf die `FINDREPLACE`-Struktur holen und diese direkt abfragen oder füllen. Einfacher funktionieren jedoch die Methoden. *GetFindString* liefert die Zeichenkette, die gesucht werden soll. Mit *FindNext* wird ermittelt, ob der Benutzer die Schaltfläche `WEITERSUCHEN` gedrückt hat. Mit Hilfe von *IsTerminating* wird geprüft, ob der Dialog beendet werden soll. *MatchCase* fragt ab, ob zwischen Groß- und Kleinschreibung unterschieden werden soll, und *SearchDown* ermittelt die Suchrichtung.

Flag	Bedeutung
FR_FINDNEXT FR_DIALOGTERM	In Zusammenhang mit einer <code>FINDMSGSTRING</code> -Nachricht, in der Nachrichten an das übergeordnete Fenster geleitet werden, wird entweder die Funktion für »Weitersuchen« oder »Abbrechen« aufgerufen.
FR_MATCHCASE	Schalter zum Setzen oder Lesen des Kontrollkästchens zur Unterscheidung von »Groß-/Kleinschreibung«.
FR_DOWN	Schalter zum Setzen oder Lesen des Optionsfeldes »Suchrichtung«.
FR_WHOLEWORD	Schalter zum Setzen oder Lesen des Kontrollkästchens »Nur ganzes Wort suchen«.
FR_SHOWHELP	Zeigt zusätzlich einen Hilfe-Button an, der eine Verknüpfung zur Online-Hilfe herstellt.
FR_NOUPDOWN FR_NOMATCHCASE FR_NOWHOLEWORD	Wenn diese Flags vor dem Initialisieren der Dialogbox gesetzt sind, werden die jeweiligen Steuerelemente grau dargestellt und sind nicht anwählbar.
FR_HIDEUPDOWN FR_HIDEMATCHCASE FR_HIDEWHOLEWORD	Wenn diese Flags vor dem Initialisieren der Dialogbox gesetzt sind, werden die jeweiligen Steuerelemente nicht dargestellt.

Tabelle 5.5: Einige Flags in *CFindReplaceDialog*

5.5.2 Nachrichtenbehandlung des Dialogs

Nachdem nun die Dialogbox mittels *Create* dem Anwender präsentiert wird, muß auf die Aktionen des Benutzers reagiert werden. Wie Nachrichtenbehandlungsfunktionen angelegt werden, wurde schon besprochen. Doch in diesem Fall läßt sich das Problem nicht so einfach mittels des Klassen-Assistenten lösen. Die Nachrichten müssen sozusagen aus der Klasse heraus an das übergeordnete Fenster geleitet werden. Eine Voraussetzung dafür ist das Übergeben eines *CWnd*-Zeigers in der Methode *Create*. Des weiteren ist es erforderlich, die Nachricht der *Suchen/Ersetzen*-Dialogbox (FINDMSGSTRING) zu registrieren. Das erfolgt am Anfang der Datei *ChildView.cpp*.

```
static UINT WM_FINDREPLACE = ::RegisterWindowMessage(FINDMSGSTRING);
```

Die Nachricht erhält im Parameter *lParam* einen Zeiger auf die *FINDREPLACE*-Struktur. Anschließend wird ein Eintrag in der Message-Map erstellt, um auf die Nachricht reagieren zu können. Damit ist es dann möglich, die Flags auszuwerten und entsprechende Aktionen auszuführen.

```
BEGIN_MESSAGE_MAP(CChildView, CWnd)
    //{{AFX_MSG_MAP(CSuchTextView)
    ...
    //}}AFX_MSG_MAP
    ON_REGISTERED_MESSAGE( WM_FINDREPLACE, OnSuchen)
END_MESSAGE_MAP()
```

Bitte beachten Sie, daß die Nachricht von Hand in die Message-Map eingetragen wird und deshalb auch außerhalb der Klammern des Klassen-Assistenten stehen muß.



Nachdem all diese Arbeiten ausgeführt sind, kann nun endlich mit der Implementierung der Methode *OnSuchen* begonnen werden. Die grobe Struktur sieht so aus, daß in der Methode die *FINDREPLACE*-Struktur ausgewertet wird oder die einzelnen Parameter direkt aus der Klasse abgerufen werden. Möglich wird das durch Definition der Member-Variablen der Klasse *CFindReplaceDialog* in der Header-Datei der Ansichts-Klasse. Alle Methoden innerhalb der Klasse können darauf zugreifen.

```
LRESULT CChildView::OnSuchen(WPARAM, LPARAM lParam)
{
    if (pFR->IsTerminating())
    {
        ...
    }
    else if (pFR->FindNext())
    {
        ...
    }
}
```

```

    }
    InvalidateRect(NULL, FALSE);
    UpdateWindow();
    return 0;
}

```

Listing: Nachrichtenbehandlung für WM_FINDREPLACE

Über die *if*-Abfragen oder eine *switch*-Anweisung läßt sich ermitteln, welche Aktion zum Aufruf von *OnSuchen* geführt hat. Das ist erforderlich, da bei jeder Aktion im Dialog die gleiche Nachricht verschickt wird. Wenn die *Ersetzen*-Funktionen aktiv sind, müssen auch diese dahingehend geprüft werden. Zuerst muß allerdings immer ermittelt werden, ob die Dialogbox mit der Schaltfläche ABBRECHEN verlassen werden soll. Dann darf keine weitere Aktion folgen.

Wird nicht mit einer öffentlichen Member-Variablen für den Dialog gearbeitet, muß in der Nachrichtenbehandlungsfunktion *lParam* ausgewertet werden.

```

LPFINDREPLACE lpfr;
...
lpfr = (LPFINDREPLACE)lParam;
if (lpfr->Flags & FR_DIALOGTERM)
{
    ...
}
if (lpfr->Flags & FR_FINDNEXT)
    Suchen(lpfr->lpstrFindWhat,
           (BOOL) (lpfr->Flags & FR_DOWN),
           (BOOL) (lpfr->Flags & FR_MATCHCASE));
...

```

5.5.3 Implementierung von *OnSuchen*

Nachdem der Benutzer eine der Schaltflächen im *Suchen*-Dialog gedrückt hat, muß eine Reaktion darauf erfolgen. Das Gerüst der Routine wurde bereits im vorhergehenden Abschnitt erläutert.

```

LRESULT CChildView::OnSuchen(WPARAM, LPARAM lParam)
{
    CString sSuche;    //Suchstring
    BOOL bGrossKlein; //Groß-/Kleinschreibung
    CStdioFile fi;     //Datei-Klasse
    DWORD fLength;     //Dateilänge
    long lZeile;       //Zeilen-Nummer
    CString sZeile, sTmpZeile; //eine Text-Zeile

```

```

if (pFR->IsTerminating())
{
    pFR = NULL;
    //Beenden Such-Dialog
}
else if (pFR->FindNext())    //Weitersuchen
{
    bGrossKlein = pFR->MatchCase();
    sSuche = pFR->GetFindString();
    if (!bGrossKlein)
        sSuche.MakeUpper();
    if ( fi.Open(sFileName,CFile::modeRead))
    {
        FLength=fi.GetLength();
        lZeile=0;
        while (fi.GetPosition()<FLength)
        {
            lZeile++;
            fi.ReadString(sZeile);
            sTmpZeile = sZeile;
            if (!bGrossKlein)
                sTmpZeile.MakeUpper();
            if (sTmpZeile.Find(sSuche) > -1)
            {
                sTmpZeile.Format("%5ld:
                                %s",lZeile,sZeile);
                aZeilen->Add(sTmpZeile);
            }
        }
        fi.Close();
    }
}
InvalidateRect(NULL,FALSE);
UpdateWindow();
return 0;
}

```

Listing: Die gesamte Such-Funktion

Nach der Definition einiger Variablen wird zuerst mit der Methode *IsTerminating* kontrolliert, ob der Benutzer die ABBRECHEN-Schaltfläche betätigt hat. Ist das der Fall, wird der Dialog auf NULL gesetzt und damit als ungültig gekennzeichnet. Als nächstes wird auf das Betätigen der WEITERSUCHEN-

Schaltfläche hin geprüft. Hier wird die eigentliche Textsuche durchgeführt. Dazu muß die zu suchende Zeichenkette mit der Methode *GetFindString* ermittelt werden. Über *MatchCase* wird ermittelt, ob der Benutzer eine Unterscheidung zwischen Groß- und Kleinschreibung wünscht. Sind noch weitere Optionen, wie z.B. *ganze Wörter suchen*, anwählbar, so müssen auch diese abgefragt werden.

Nach dem erfolgreichen Öffnen der Datei wird die Dateilänge ermittelt (*GetLength*) und in einer Schleife so lange gelesen, bis die aktuelle Position des Dateizeigers (*GetPosition*) mit der Dateilänge übereinstimmt – d.h., das Dateiende ist erreicht. Das Lesen einer Zeile erfolgt mit der Methode *ReadString*. Die Klasse *CStdioFile* eignet sich sehr gut für Textdateien, da mittels *ReadString* immer nur eine Zeile gelesen wird, gleichgültig, wie lang sie ist. Der restliche Teil gestaltet sich ebenfalls recht einfach. Wird keine Unterscheidung von Groß- und Kleinschreibung gefordert, werden die gelesene Zeile und die Suchzeichenkette in Großbuchstaben umgewandelt. Der Vergleich selbst erfolgt über eine Methode der Klasse *CString* (siehe Kapitel 11.1), die selbst eine Teilzeichenkette in einer anderen sucht. War die Suche erfolgreich, wird die Zeile in ein Zeichenketten-Array eingefügt. Die Erläuterungen dazu erfolgen in Kapitel 11.2. Hier genügt es, zu wissen, daß mit der Methode *Add* die Zeile an das Array angehängt wird.

Damit ist die Implementierung von Step2 des *SuchText*-Programms abgeschlossen. Zur Zeit können wir schon eine Datei auswählen und in ihr nach einer Zeichenkette suchen. Eigentlich fehlt nur noch die Ausgabe der gefundenen Zeilen.



Daß diese Variante des Suchens nicht optimal ist, können Sie sich vorstellen. Der Benutzer drückt einmal auf *Weitersuchen*, und die gesamte Datei wird durchsucht. Das mag dann von Vorteil sein, wenn mehrere Dateien oder ein Verzeichnis durchsucht werden sollen. Mögliche Erweiterungen sehen so aus:

- ▼ Anhalten des Suchvorgangs nach jedem gefundenen Vorkommen.
- ▼ Ausgeben von zusätzlich einer Zeile vor und einer nach der Fundstelle, um einen besseren Überblick zu erhalten.
- ▼ Durchsuchen mehrerer Dateien über eine Suchmaske (*.cpp o.ä.) oder eines Verzeichnisses.

Diese Erweiterungsmöglichkeiten sollten Sie sich für einen späteren Zeitpunkt vormerken, da sie nicht ohne weiteres realisierbar sind. Die eigentliche Aufgabe ist leichter zu lösen. Ein Problem, das es zu lösen gibt, besteht darin, daß beim Weitersuchen in einer bereits durchsuchten oder in einer neuen Datei die gefundenen Zeilen an das Array angehängt werden.

Dadurch geht schnell der Überblick verloren. Sie können dies umgehen, indem Sie das Array vor einer neuen Suche leeren. Plazieren Sie dazu den Befehl

```
aZeilen.RemoveAll();
```

an geeigneter Stelle im Programm.

5.6 Verändern der Programm-Titelleiste

In der Regel besitzt jedes Hauptfenster eine Titelleiste, also einen Text, der dem Benutzer einen Hinweis auf das Programm und eventuell zusätzlich auf den Kontext seiner Arbeit gibt. In *SuchText* enthält die Titelleiste nach dem Starten des Programms die Überschrift »SuchText. Nach dem Öffnen soll zusätzlich der Name der geladenen Datei angezeigt werden. Der ursprüngliche Inhalt der Titelleiste wird durch den Aufruf von *CFrameWnd::Create* bestimmt, dessen zweiter Parameter ein Zeiger auf den nullterminierten String mit dem Fenstertitel ist:

```
Create( NULL,
        "SuchText", WS_OVERLAPPEDWINDOW |
                    WS_VSCROLL | WS_HSCROLL,
        rectDefault, NULL,
        MAKEINTRESOURCE(IDR_MAINMENU) );
```

Diese Methode wird nicht bei Verwendung eines vom Anwendungs-Assistenten generierten Projekts gewählt. Bei Verwendung der Dokument-Ansicht-Architektur wird beim Anlegen eines neuen Dokuments automatisch die Methode *SetDefaultTitle* von *CDocTemplate* aufgerufen, die den Text der Titelzeile setzt. Letzterer kommt in jedem Fall aus der Zeichenkettenressource IDR_MAINFRAME.

Um den vorgegebenen Fenstertitel zu verändern, kann die Methode *SetWindowText* von *CWnd* verwendet werden.

```
class Cwnd:
    void SetWindowText( LPCTSTR lpszString );
```

Sie erwartet einen Zeiger auf einen nullterminierten String, der den Text für die neue Titelleiste des Fensters enthält. Es ist auch möglich, diese Funktion auf Steuerungen anzuwenden, die keinen Titel haben. So verändert sich bei einer Schaltfläche beispielsweise deren Beschriftung. Intern bekommt die Fensterklasse in jedem Fall die Nachricht WM_SETTEXT zugesandt. Der für das Anzeigen des Dateinamens verantwortliche Teil von *SuchText* befindet sich am Ende der Methode *OnDateiOeffnen* und sieht so aus:

```
void ChildView::OnDateiOeffnen()
{
    ...
    char buf[256];
    sprintf(buf,"SuchText - %s",sFileName);
    AfxGetMainWnd()->SetWindowText( buf );
    ...
}
```

Listing: Verändern der Titelzeile beim Öffnen einer Datei

Die Methode *SetWindowText* ist in der Klasse *CWnd* definiert. Sie wirkt immer auf das Fenster der Klasse, in der sie aufgerufen wird. Über die Methode *AfxGetMainWnd()* kann das Handle des Hauptfensters geholt und somit an jeder Stelle im Programm dessen Titelzeile verändert werden.

Denken Sie daran, den Dateinamen aus der Titelzeile zu entfernen, wenn die Datei geschlossen wird!



Weitere Gestaltungsmöglichkeiten der Titelzeile finden Sie im Microsoft System Journal 2/97 und 3/97. Dort wird einerseits gezeigt, wie Statusinformationen über den verfügbaren Speicher angezeigt werden und andererseits, wie eine Titelzeile mit Farbverlauf ähnlich der von Winword erzeugt wird.



Ausgaben in Fenstern

6

Kapitelübersicht

6.1	Die OnPaint/OnDraw-Routine	200
6.1.1	Die Klasse CPaintDC	202
6.1.2	Der Arbeitsbereich des Fensters	203
6.1.3	Regionen im Device-Kontext	205
6.2	Schrift- und Hintergrundfarbe	208
6.2.1	Einstellen der Schriftfarbe	208
6.2.2	Einstellen der Hintergrundfarbe	210
6.2.3	Verwenden von Systemfarben	211
6.3	Schriftattribute	212
6.3.1	Merkmale von Schriftarten	212
6.3.2	Vordefinierte Schriftarten	214
6.3.3	Erzeugen eigener Schriftarten	217
6.4	Ausgeben des Textes	219
6.4.1	Die Datenstruktur zur Darstellung des Textes	219
6.4.2	Größenmerkmale der Schrift	220
6.4.3	Textausrichtung	222
6.4.4	Die Textausgabe mit ExtTextOut	223
6.4.5	Einfachere Formen der Textausgabe	225
6.5	Bildschirm Ausgaben außerhalb von OnPaint	227

6.1 Die *OnPaint/OnDraw-Routine*

Bei der Programmierung mit herkömmlichen Entwicklungsumgebungen, die weder grafik- noch fensterorientiert sind, ist das Ausgeben von Text recht einfach: Dem Programm steht der gesamte Bildschirm zur Verfügung und es muß nicht befürchtet werden, daß andere Programme seine Ausgaben überschreiben; Schriftattribute gibt es nicht, bestenfalls unterschiedliche Farben. Betrachtet man Schritt für Schritt die Evolution von einem solchen Textbildschirm zu einem voll grafikfähigen System, wird schnell deutlich, daß diese Einfachheit in der Programmierung nicht lange beibehalten werden kann.

Aufwendiger wird es nämlich schon in einem fensterorientierten, nicht-grafischen System. Ist die Fenstertechnik stackorientiert, d.h., sind Ausgaben nur jeweils im obersten Fenster erlaubt, hat der Fenstermanager noch nicht viel zu tun: Beim Anlegen eines neuen Fensters speichert er den Teil des davon verdeckten Bildschirms, überwacht beim Schreiben die Fenstergrenzen und restauriert beim Schließen den ehemals verdeckten Bereich.

Soll zusätzlich erlaubt werden, daß auch in Fenstern geschrieben werden darf, die nicht ganz oben liegen, daß Fenster vertauscht werden können und daß die Größe und Position der Fenster vom Anwender bestimmt werden können, braucht es schon einen wesentlich komplexeren Fenstermanager. Er muß beispielsweise Ausgaben in verdeckten Bereichen speichern, aber nicht (sofort) anzeigen, oder er muß dafür sorgen, daß beim Vergrößern eines Fensters keine weißen Flecken entstehen. Die Veränderlichkeit der Fensteranordnung durch den Anwender erfordert zusätzlich einen Mechanismus, der das Programm über einen solchen Vorgang informiert – der Weg bis zur Einführung eines nachrichtenorientierten Systems ist nun nicht mehr weit.

Soll jetzt auch noch die Grafikfähigkeit eingeführt werden, ist es vorbei mit einem simplen Ansatz: nicht nur die Komplexität, sondern auch der Speicherbedarf für die verdeckten Fensterteile steigt steil an – genau entgegengesetzt verhält es sich mit der Performance. Alle bekannten grafischen, fensterorientierten Oberflächen gehen daher einen anderen Weg: sie sorgen zwar für die Verwaltung der allgemeinen Fenstermerkmale, überlassen aber das Neuzeichnen (z.B. bei Veränderungen der Anordnung der Fenster) den zugeordneten Applikationen; eine Speicherung verdeckter Bildschirm Inhalte erfolgt nicht (oder nur in Ausnahmefällen). Die Koppelung zwischen Programm und Bildschirmausgabe wird mit Hilfe nachrichtenorientierter Verarbeitung umgekehrt: die Dialoge steuern das Programm.

Unter Windows gibt es für die Darstellung eines Fensters die Nachricht `WM_PAINT`. Sie wird von Windows immer dann an eine Fensterprozedur geschickt, wenn das Fenster teilweise oder ganz neu gezeichnet werden

muß. Um beispielsweise nach dem Schließen eines Fensters Teile von bisher verdeckten Fenstern neu zu zeichnen, sendet Windows einfach in der richtigen Reihenfolge WM_PAINT-Nachrichten an die zugehörigen Fensterprozeduren. So wird automatisch der Bildschirm korrekt aufgebaut, und jedes Programm kann annehmen, es hätte den ganzen Arbeitsbereich seines Fensters zur Verfügung.

In einem MFC-Programm wird bei einer WM_PAINT-Nachricht die Methode *OnPaint* aufgerufen; sie ist in *CWnd* realisiert und ruft die Standard-Fensterprozedur auf.

```
class CWnd:
    afx_msg void OnPaint();
```

Durch Einfügen von ON_WM_PAINT in die Message-Map einer Klasse sorgt man dafür, daß die eigene *OnPaint*-Methode angesprungen wird.

```
void CMainFrame::OnPaint()
{
    ...
}

...
BEGIN_MESSAGE_MAP(CChildView, CWnd)
...
    ON_WM_PAINT()
...
END_MESSAGE_MAP()
```

OnPaint ist also eine Methode, die immer dann aufgerufen wird, wenn ein Teil des Fensters neu gezeichnet werden muß. Dabei ist es durchaus möglich, daß derselbe Inhalt mehrfach ausgegeben werden muß; *OnPaint* muß also immer den aktuellen Zustand der Ausgabedaten kennen.

Bei Verwendung der Dokument-Ansicht-Architektur wird *OnPaint* nicht direkt aufgerufen. Statt dessen stellt die Ansicht die Methode *OnDraw* bereit, die genauso zu behandeln ist. *OnDraw* wird in der *OnPaint*-Methode der Klasse *CView* aufgerufen. Wenn in den folgenden Abschnitten von *OnPaint* die Rede ist, kann das Gesagte auch auf *OnDraw* angewendet werden.

```
void CView::OnPaint()
{
    // standard paint routine
    CPaintDC dc(this);
    OnPrepareDC(&dc);
    OnDraw(&dc);
}
```

Listing: Ausschnitt aus der Klasse CView

6.1.1 Die Klasse *CPaintDC*

Um Ausgaben im Client-Bereich eines Fensters vorzunehmen, benötigt die *OnPaint*-Routine einen Handle auf einen *Device-Kontext*. Dies ist eine Datenstruktur, die einem bestimmten Ausgabegerät zugeordnet ist und die Aufgabe hat, dessen Eigenschaften zu verwalten. In der herkömmlichen Windows-Programmierung (SDK) wird ein Handle auf einen Device-Kontext mit Hilfe der Funktion *BeginPaint* beschafft und am Ende der Zeichenroutine durch Aufruf von *EndPaint* explizit wieder freigegeben. Allen Zeichenfunktionen wird der Handle als Parameter übergeben.

Unter den Foundation Classes wird die Verwaltung des Device-Kontexts durch die Klasse *CDC* und ihre Abkömmlinge, insbesondere *CPaintDC*, vereinfacht. Statt des Aufrufs von *BeginPaint* und *EndPaint* braucht nur noch am Anfang der *OnPaint*-Methode ein lokales *CPaintDC*-Objekt angelegt und zum Zeichnen im Arbeitsbereich brauchen nur noch dessen Methoden verwendet zu werden. Dies hat zwei Vorteile: Erstens sorgt der Destruktor von *CPaintDC* automatisch für die Freigabe des Device-Kontexts beim Verlassen von *OnPaint*, und zweitens sind alle GDI-Funktionen Methoden von *CDC*, die einfach auf dem erzeugten *CPaintDC*-Objekt aufgerufen werden; die umständliche Handhabung des Handles entfällt also völlig.

Das Gerüst einer *OnPaint*-Methode sieht damit typischerweise so aus:

```
void CMainFrame::OnPaint()
{
    CPaintDC dc(this);
    ...
    //Hier wird das Fenster gezeichnet und die GDI-
    Funktionen werden aufgerufen
    ...
}
```

Wenn Sie sich die Definition von *OnDraw* in der Klasse *CView* noch einmal ansehen, finden Sie genau das dort wieder.

CPaintDC besitzt keinen Default-Konstruktor, sondern verlangt einen Zeiger auf das Fenster, zu dem der Device-Kontext gehört – also meist *this*. Weiterhin besitzt *CPaintDC* einen öffentlichen Datenmember *m_ps*, der die *PAINTSTRUCT*-Struktur enthält, die manchmal beim Zeichnen des Fensters gebraucht wird.

```
typedef struct tagPAINTSTRUCT {  
    HDC   hdc;  
    BOOL  fErase;  
    RECT  rcPaint;  
    BOOL  fRestore;  
    BOOL  fIncUpdate;  
    BYTE  rgbReserved[16];  
} PAINTSTRUCT;
```

`m_ps` enthält den Handle `hdc` auf den Device-Kontext; er kann verwendet werden, um herkömmliche SDK-Funktionen aufzurufen, die diesen als Parameter benötigen. `fErase` ist genau dann `TRUE`, wenn der Hintergrund neu gezeichnet werden soll, und `rcPaint` ist der rechteckige Teil des Client-Bereichs, der wenigstens neu gezeichnet werden muß. Die anderen Bestandteile von `PAINTSTRUCT` sind nicht dokumentiert und nur für Windows selbst von Bedeutung.

Ein Objekt aus `CPaintDC` darf nur in Reaktion auf eine `WM_PAINT`-Nachricht erzeugt werden, also in der Methode `OnPaint`. Wie man einen Device-Kontext beschafft, um außerhalb von `OnPaint` zu zeichnen, wird weiter unten erläutert.

6.1.2 Der Arbeitsbereich des Fensters

Oft muß die Anwendung wissen, wie groß der Arbeitsbereich ist, in dem sie zeichnen darf. Zur Compilezeit ist dieser Wert nicht bekannt, und selbst zur Laufzeit kann er sich zwischen zwei `OnPaint`-Aufrufen verändern (nämlich immer dann, wenn der Benutzer das Fenster vergrößert oder verkleinert hat). Auf unterschiedlicher Bildschirmhardware kann außerdem zur Darstellung derselben Information ein unterschiedlich großer Arbeitsbereich nötig sein. Mitunter muß die Anwendung außerdem wissen, welche Ausdehnung das gesamte Fenster hat, beispielsweise um Bildschirm- in Fensterkoordinaten umzurechnen.

```
class Cwnd:  
    void GetClientRect( LPRECT lpRect ) const;  
    void GetWindowRect( LPRECT lpRect ) const;
```

Aus diesem Grund gibt es die Methoden `GetClientRect` und `GetWindowRect` in der Klasse `CWnd`, mit deren Hilfe diese Größen ermittelt werden können. Beide Funktionen erwarten als Parameter einen Zeiger auf eine `RECT`-Struktur, in der sie die gewünschten Informationen zurückgeben können. Die `RECT`-Struktur hat folgenden Aufbau:

```
typedef struct tagRECT {  
    int left;  
    int top;  
    int right;  
    int bottom;  
} RECT;
```

Sie beschreibt ein Rechteck durch zwei Eckpunkte; *left* und *top* sind die Koordinaten der linken oberen Ecke, *bottom* und *right* diejenigen der rechten unteren Ecke. Beim Aufruf von *GetClientRect* liefert Windows die Koordinaten des Client-Bereichs des aktuellen Fensters in Fensterkoordinaten, daher hat die linke obere Ecke die Koordinaten (0,0); an der rechten unteren Ecke kann die Ausdehnung des Client-Bereichs in x- und y-Richtung abgelesen werden. Gemeint ist der Bereich des aktuellen Fensters, der für die Bildschirmausgabe durch das Programm zur Verfügung steht: also das gesamte Fenster abzüglich Rahmen, Titelleiste, Menüzeile usw.

Anders sieht es aus, wenn *GetWindowRect* aufgerufen wird: Hier liefert Windows die Position und Ausdehnung des kompletten Fensters (also inklusive Rahmen, Titel, Bildlaufleisten etc.) in Bildschirmkoordinaten. Die linke obere Ecke des gesamten Bildschirms hat hier die Koordinaten (0,0); (left, top) liefert die linke obere Ecke des Fensters und (right, bottom) die rechte untere Ecke.

Beide Funktionen werden typischerweise beim Neuzeichnen innerhalb von *OnPaint* aufgerufen. Da sich die Größe und Positionierung des Fensters zwischen zwei *OnPaint*-Aufrufen verändern kann, ist es meist nicht sinnvoll, die Resultate der Funktionsaufrufe längere Zeit zu speichern.

In *SuchText* wird *GetClientRect* ebenfalls innerhalb von *OnPaint* aufgerufen:

```
void CChildView::OnPaint()  
{  
    CPaintDC dc(this); // Gerätekontext zum Zeichnen  
    CRect winrect;  
    //Fenstergröße  
    GetClientRect(winrect);  
    ...  
}
```

Interessanterweise wird bei diesem Aufruf aber kein FAR-Zeiger auf eine RECT-Struktur übergeben, so wie es der Deklaration von *GetClientRect* entspräche, sondern ein *CRect*-Objekt. (Die Klasse *CRect* dient zur Kapselung der RECT-Struktur und verfügt neben den Datenmitgliedern *left*, *top*, *bottom* und *right* über Methoden zur Manipulation von Rechtecken; (siehe

CRect in der Online-Dokumentation). *CRect* besitzt einen Operator zur Typumwandlung, der es erlaubt, anstelle eines *LPRECT* ein einfaches *CRect*-Objekt an eine Funktion zu übergeben – und zwar ohne, daß der *&*-Operator verwendet wird!

Dieses Beispiel findet sich übrigens genau so in den MFC-Dokumentationen: Nimmt man es genau, wird hier Call-By-Reference durch die Hintertür eingeführt, und zwar auf eine nicht ganz ungefährliche Art. Die von (Standard-)C-Programmierern lange entbehrten Fähigkeiten des Compilers, Typprüfungen in Funktionsaufrufen durchzuführen, werden durch solche Methoden ziemlich ausgehebelt!

6.1.3 Regionen im Device-Kontext

Unter Windows gibt es ein Konzept zum abstrakten Umgang mit Flächen auf einem imaginären Ausgabegerät, die *Regionen*. Eine Region besteht aus einer oder mehreren rechteckigen, vieleckigen oder ellipsenförmigen Flächen, die nebeneinander liegen oder sich überlappen. Es gibt eine Reihe von Funktionen, die mit Regionen arbeiten: So können beispielsweise zwei bestehende Regionen zu einer neuen zusammengefaßt werden, es kann festgestellt werden, ob ein vorgegebener Punkt innerhalb einer Region liegt, oder man kann eine Region mit einer Farbe füllen und auf dem Bildschirm anzeigen lassen.

Regionen können vom Programmierer nach Belieben erzeugt und manipuliert werden, sie werden oft eingesetzt, um komplexe grafische Effekte zu erzielen. Darüber hinaus spielen zwei von Windows verwaltete Regionen bei der Ausgabe im Client-Bereich eine wichtige Rolle:

- ▼ die *Update-Region* gibt den Bereich der Client-Area an, der ungültig ist, und der nach Senden einer *WM_PAINT*-Nachricht einer Neuerstellung bedarf
- ▼ die *Clipping-Region* gibt den Bereich der Client-Area an, in dem Ausgaben der *OnPaint*-Routine auch tatsächlich auf dem Bildschirm erscheinen

Die Update-Region ist zunächst leer – sie wird von Windows vergrößert, wenn ein Teil des Fensters sichtbar wird, der bisher nicht sichtbar war. Dies kann dadurch zustande kommen, daß ein Fenster das erste Mal angezeigt wird oder ein bisher (auch teilweise) verdecktes Fenster durch Eingriffe des Benutzers sichtbar wird. Die Update-Region hat unmittelbaren Einfluß auf das Neuzeichnen des Fensters: Ist sie nicht leer und liegen keine weiteren Nachrichten an, so sendet Windows eine *WM_PAINT*-Nachricht an die Fensterprozedur, ruft also die *OnPaint*-Methode auf.

Im Zusammenhang mit der Update-Region gibt es einige interessante Techniken. Diese sollen anhand der drei Methoden *UpdateWindow*, *InvalidateRect* und *GetUpdateRect* von *CWnd* kurz vorgestellt werden.

```
class Cwnd:
    void UpdateWindow();
    void InvalidateRect( LPCRECT lpRect, BOOL bErase = FALSE);
    BOOL GetUpdateRect( LPRECT lpRect, BOOL bErase = FALSE);
```

Will eine Anwendung eine WM_PAINT-Nachricht an eines ihrer Fenster senden, so braucht sie nicht explizit eine Methode zum Nachrichtentransfer aufzurufen, sondern kann dazu *UpdateWindow* verwenden. WM_PAINT wird allerdings nur dann geschickt, wenn die Update-Region nicht leer ist. Mit Hilfe von *InvalidateRect* wird die Update-Region vergrößert, und zwar um das Rechteck *lpRect*. Falls *lpRect* gleich NULL ist, wird der gesamte Client-Bereich zur Update-Region. Will eine Anwendung ein Neuzeichnen ihres Fensters erreichen, reicht es also nicht aus, *UpdateWindow* aufzurufen, sondern es muß zuvor mit *InvalidateRect* wenigstens ein kleiner Bereich als ungültig markiert worden sein.

Da es aus Performance-Gründen nicht immer sinnvoll ist, den kompletten Client-Bereich neu zu zeichnen, kann innerhalb von *OnPaint* durch Aufruf von *GetUpdateRect* festgestellt werden, wie groß der Update-Bereich ist. Hierbei ist *lpRect* ein Ergebnisparameter, der nach dem Aufruf das kleinste Rechteck angibt, das die Update-Region vollständig umschließt. Mit diesen Angaben wäre *OnPaint* in der Lage, nur den wirklich reparaturbedürftigen Teil seines Fensters wiederherzustellen – ob der dazu erforderliche Mehraufwand an Programmierung lohnt, ist von der Komplexität der Ausgabe und dem Zeitbedarf zum Neuaufbau des gesamten Bildschirms abhängig.

Der boolesche Parameter *fErase* hat in beiden Methoden die gleiche Bedeutung: Ist er TRUE, so wird vor dem Neuzeichnen des Fensters dessen Hintergrund restauriert. Da dies implizit (durch Senden einer WM_ERASEBKGD-Nachricht) in der *BeginPaint*-Routine geschieht, funktioniert es allerdings nur dann, wenn die Update-Region nicht leer ist.

Beim Aufruf von *GetUpdateRect* ist etwas Vorsicht geboten, denn dieser muß vor *BeginPaint* (also vor dem Anlegen des *CPaintDC*-Objektes) erfolgen:

```
void CChildView::OnPaint()
{
    CRect updirect;
    GetUpdateRect(updirect);
    CPaintDC dc(this);
    ...
}
```

Der Grund dafür ist, daß der in *CPaintDC* *dc(this)* versteckte Aufruf von *BeginPaint* die Update-Region auf Null setzt, und damit in nachfolgenden Aufrufen von *GetUpdateRect* immer das Rechteck (0,0,0,0) angegeben wird. Alternativ dazu steht dieselbe Information auch im Member *rcPaint* der *PAINTSTRUCT*-Struktur *dc.m_ps* des Device-Kontext-Objekts *dc*, so daß es innerhalb von *OnPaint* einfacher ist, sich die Informationen auf diese Weise zu beschaffen.

Neben der Update-Region beeinflusst auch die Clipping-Region die Bildschirmausgabe erheblich; bei ihr handelt es sich um die Teile des Client-Bereichs, die beim Neuzeichnen auf dem Bildschirm tatsächlich verändert werden können. Ausgaben von Grafik-, Text- oder sonstigen Bildschirm-routinen, die *außerhalb* der Clipping-Region liegen, werden von Windows unterdrückt und verändern den Bildschirminhalt nicht.

Durch das Erzeugen des *CPaintDC*-Objekts wird die Clipping-Region am Anfang von *OnPaint* mit der Update-Region gleichgesetzt, d.h., alle Versuche der Anwendung, außerhalb der Update-Region zu zeichnen, werden von Windows unterdrückt – dies ist auch sinnvoll, denn der Fensterinhalt außerhalb des Update-Bereichs ist ja noch intakt.

Auch für den Clipping-Bereich gibt es einige interessante Funktionen. Mit Hilfe von *ExcludeClipRect* kann der Clipping-Bereich verkleinert werden; jeder Aufruf entfernt das Rechteck *lpRect* (bzw. *x1*, *y1*, *x2*, *y2*) aus dem bisherigen Clipping-Bereich. So kann man rechteckige Bereiche von der Ausgabe ausparen, ohne dies speziell programmieren zu müssen.

```
class CDC:
    int ExcludeClipRect( int x1, int y1,
                        int x2, int y2);
    int ExcludeClipRect( LPCRECT lpRect );
    virtual int SelectClipRgn( CRgn* pRgn );
    virtual int GetClipBox( LPRECT lpRect );
```

Genau den gegenteiligen Effekt erzielt man mit *SelectClipRgn*, wodurch der Clipping-Bereich des Fensters mit der Region *pRgn* gleichgesetzt wird. So würden beispielsweise die drei Statements

```
CRgn rgn;
rgn.CreateRectRgn (100,100,200,200);
dc.SelectClipRgn(&rgn);
```

am Anfang von *OnPaint* bewirken, daß alle Ausgaben auf den rechteckigen Bereich (100, 100, 200, 200) beschränkt bleiben, der Rest der Client-Area bleibt unverändert.

Um die aktuelle Ausdehnung der Clipping-Region zu ermitteln, kann die Methode *GetClipBox* aufgerufen werden. Nach dem Aufruf enthält der Rückgabeparameter *lpRect* das kleinste Rechteck, das die Clipping-Region vollständig umschließt.

6.2 Schrift- und Hintergrundfarbe

6.2.1 Einstellen der Schriftfarbe

Die Schriftfarbe gehört – ebenso wie Stifte, Hintergrundfarben, Schriftigenschaften und andere Attribute – zu den *GDI-Objekten*, die sich ein Device-Kontext merkt. Dieses Objekt ist für die Farbe verantwortlich, in der Text ausgegeben wird, und kann mit Hilfe der Methode *SetTextColor* von *CDC* eingestellt werden:

```
class CDC:  
    virtual COLORREF SetTextColor( COLORREF crColor);
```

Der Parameter *crColor* ist ein *RGB-Farbwert*, der mit dem Makro *RGB* zusammengesetzt werden kann:

```
RGB(cRed, cGreen, cBlue)
```

Die Parameter *cRed*, *cGreen* und *cBlue* sind jeweils vom Typ *BYTE* (also 8 Bit ohne Vorzeichen) und können Werte zwischen 0 und 255 annehmen. Jeder der Parameter bestimmt die Intensität einer der Teilfarben Rot, Grün oder Blau. Sind alle drei Parameter 0, so ist das Ergebnis Schwarz; sind alle 255, so ist es Weiß. Einige dazwischenliegende Werte und der dadurch entstehende Farbton sind in Tabelle 6.1 zu finden.

Farbe	ROT-Anteil	GRÜN-Anteil	BLAU-Anteil
Weiß	255	255	255
Rot	255	0	0
Grün	0	255	0
Blau	0	0	255
Gelb	255	255	0
Lila	255	0	255
Braun	128	64	0
Dunkelgrau	128	128	128
Hellgrau	192	192	192
Schwarz	0	0	0

Tabelle 6.1: Einige RGB-Werte

Der Rückgabewert von *SetTextColor* ist die bisher eingestellte Schriftfarbe als RGB-Farbwert. Wie bei allen Farbwünschen von Windows-Anwendungen kann es passieren, daß die genaue Farbe in dem aktuellen Device-Kontext nicht zur Verfügung steht; selbst hochauflösende Grafikkarten können nur eine begrenzte Anzahl an Farben gleichzeitig anzeigen, typischerweise 16, 256 oder 32768. Mit Hilfe eines RGB-Farbwerts können aber 16 Millionen Farben (genau 16777216) dargestellt werden. Ist der gewünschte Farbwert nicht durch das Ausgabegerät darstellbar, so verwendet der Farb-Mapper von Windows die am besten passende Farbe: Linien und Text werden in der günstigsten verfügbaren Vollfarbe angezeigt, Flächen und Hintergründe per *Dithering* (Bildung von Mischfarben durch Aufrasterung mit mehrfarbigen Pixelgruppen) angenähert.

In *SuchText* geschieht die Farbeinstellung in zwei Schritten; zuerst muß der Benutzer mit Hilfe des Menüs SCHRIFT/FARBE die gewünschte Farbe auswählen, danach wird ein Aufruf von *OnPaint* erzeugt und der komplette Text in der neuen Farbe *nFarbe* ausgegeben.

Die Auswahl eines Punktes aus dem Menü SCHRIFT/FARBE löst zunächst eine WM_COMMAND-Nachricht mit einem der Bezeichner ID_SCHRIFT_FARBE_SCHWARZ, ID_SCHRIFT_FARBE_BLAU, ID_SCHRIFT_FARBE_GRUEN oder ID_SCHRIFT_FARBE_WEISS aus. All diese Nachrichten werden durch entsprechende Einträge in der Message-Map mit dem Klassen-Assistenten gemeinsam auf die Methode *OnSchriftFarbe* umgeleitet:

```
BEGIN_MESSAGE_MAP(CChildView, CWnd)
    //{{AFX_MSG_MAP(CChildView)
    ...
    ON_COMMAND(ID_SCHRIFT_FARBE_SCHWARZ, OnSchriftFarbe)
    ON_COMMAND(ID_SCHRIFT_FARBE_WEISS, OnSchriftFarbe)
    ON_COMMAND(ID_SCHRIFT_FARBE_GRUEN, OnSchriftFarbe)
    ON_COMMAND(ID_SCHRIFT_FARBE_BLAU, OnSchriftFarbe)
    ...
}
```

Zunächst soll der programminterne Farbwert *nFarbe* ermittelt werden. Dazu wird einfach die ID des gewählten Menüpunktes ermittelt. *GetCurrentMessage* liefert die aktuelle Nachricht, die in *wParam* die ID enthält. Durch Subtraktion von ID_SCHRIFT_FARBE_SCHWARZ erhält man Werte zwischen Null und drei. Anschließend wird das Neuzeichnen veranlaßt.

```
void CChildView::OnSchriftFarbe()
{
    nFarbe = GetCurrentMessage()->wParam - ID_SCHRIFT_FARBE_SCHWARZ;
    ...
    InvalidateRect(NULL,TRUE);
    UpdateWindow();
}
```

Listing: Ermittlung des Farb-Index

Innerhalb von *OnPaint* wird die Variable *nFarbe* dann dazu verwendet, um die Textfarbe vor der Ausgabe des Textes auf den gewünschten Wert einzustellen:

```
void CChildView::OnPaint()
{
    ...
    switch (nFarbe)
    {
        case 0 : dc.SetTextColor(RGB( 0, 0, 0)); break;
        case 1 : dc.SetTextColor(RGB(255,255,255)); break;
        case 2 : dc.SetTextColor(RGB( 0, 0,255)); break;
        case 3 : dc.SetTextColor(RGB( 0,255, 0)); break;
    }
    ...
}
```

Listing: Einstellen der Textfarbe in OnPaint

Es ist übrigens auch möglich, die Textfarbe zu ermitteln. Dazu gibt es die Funktion *GetTextColor* von *CDC*, deren Rückgabewert die aktuell eingestellte Textfarbe, definiert als RGB-Farbwert, ist:

```
class CDC:
    COLORREF GetTextColor() const;
```

6.2.2 Einstellen der Hintergrundfarbe

Ebenso wie das Einstellen der Vordergrundfarbe möglich ist, kann auch die Farbe des Hintergrunds verändert werden. Hierbei muß man sich neben der eigentlichen Farbgebung auch um den Darstellungsmodus für den Hintergrund kümmern.

Die Auswahl der Hintergrundfarbe erfolgt – analog zur Vordergrundfarbe – durch die Methode *SetBkColor* von *CDC*:

```
class CDC:
    virtual COLORREF SetBkColor(COLORREF crColor);
```

In *crColor* wird die gewünschte Hintergrundfarbe als RGB-Farbwert übergeben; zurückgegeben wird die bisher eingestellte Hintergrundfarbe. Die so eingestellte Hintergrundfarbe wird danach bei drei verschiedenen Ausgabeoperationen verwendet:

- ▼ in den Lücken zwischen den Buchstaben bei der Textausgabe
- ▼ in den Lücken nicht-durchgehender Linien
- ▼ bei Pinseln mit Muster

Neben der Farbe spielt der Darstellungsmodus eine wichtige Rolle, er kann entweder *undurchsichtig* oder *transparent* sein und wird mit der Methode *SetBkMode* von *CDC* eingestellt:

```
class CDC:
    int SetBkMode( int nBkMode );
```

Soll der Hintergrund undurchsichtig sein, muß der Wert *OPAQUE* an den Parameter *nBkMode* übergeben werden. In diesem Fall wird die neue Hintergrundfarbe als Lückenfüller bei der Ausgabe von Text, Linien oder Pinseln benutzt. Ist der Parameter *nBkMode* hingegen *TRANSPARENT*, so wird die eingestellte Hintergrundfarbe nicht verwendet. Statt dessen bleiben die Lücken im Text, in Linien oder in gemusterten Pinseln unverändert. Auf diese Weise bleibt der bisherige Hintergrund erhalten, und man hat den Eindruck, als würde durchsichtige Farbe verwendet.

In *SuchText* wird der Hintergrund durch den Menüpunkt *SCHRIFT/INVERTIERT* verändert. Jeder Aufruf invertiert den Zustand der logischen Variablen *bInvers*, die dann in *OnPaint* benutzt wird, um den Hintergrund entweder schwarz oder weiß zu machen. In *SuchText* ist der Hintergrundmodus immer *OPAQUE*, um die Textlücken in derselben Farbe darzustellen, wie den Rest des Client-Bereichs. Da *OPAQUE* die Standardeinstellung ist, muß die Funktion nicht explizit aufgerufen werden.

```
void CChildView::OnPaint()
{
    ...
    dc.SetBkColor(bInvers ? RGB(0,0,0):RGB(255,255,255));
    ...
}
```

Listing: Einstellen der Hintergrundfarbe

6.2.3 Verwenden von Systemfarben

Jeder Windows-Anwender kennt die Möglichkeit, mit der Systemsteuerung den eigenen Rechner frei zu konfigurieren – unter anderem gibt es hier die Möglichkeit, die Farben einzustellen. Dabei werden bestimmten Bildelementen feste Farben zugewiesen – in der Hoffnung, daß die Anwendungen sich daran orientieren. Bei den Nicht-Client-Elementen eines Fensters sorgt die Standard-Fensterprozedur automatisch dafür, daß dies so ist, im Client-Bereich jedoch muß die Anwendung dafür sorgen.

Um herauszufinden, welche Farbe die einzelnen Bildelemente haben sollen, gibt es die globale Methode *GetSysColor*:

```
DWORD GetSysColor( int nIndex );
```

Sie liefert die Farbe des durch *nIndex* bezeichneten Fensterelementes als RGB-Farbwert zurück, der dann von der Anwendung dazu verwendet werden kann, die eigenen Fensterelemente entsprechend darzustellen. Die wichtigsten Werte für *nIndex* können Tabelle 6.2 entnommen werden, jeder einzelne entspricht dabei einer Einstellmöglichkeit in der Systemsteuerung:

nIndex	Bildschirmelement
COLOR_WINDOW	Fensterhintergrund der Client-Area
COLOR_WINDOWTEXT	Text im Fenster
COLOR_HIGHLIGHTTEXT	hervorgehobener Text in einer Dialogbox-Steuerung
COLOR_HIGHLIGHT	Hintergrund von hervorgehobenem Text in einer Dialogbox-Steuerung
COLOR_BACKGROUND	Desktops

Tabelle 6.2: Systemfarbenwerte



Eine vollständige Auflistung aller möglichen Parameter für *nIndex* finden Sie in der Online-Dokumentation unter *GetSysColor*. Die übrigen Werte sind vor allem für Darstellungen im Nicht-Client-Bereich von Bedeutung und können der Dokumentation entnommen werden.

Um auf die einzelnen Teilfarben eines RGB-Farbvals zuzugreifen, können die vordefinierten Makros *GetRValue*, *GetGValue* und *GetBValue* verwendet werden. Sie liefern jeweils den Anteil der Teilfarbe Rot, Grün oder Blau in Form eines Bytes zurück:

```
BYTE GetRValue( DWORD rgb );
BYTE GetGValue( DWORD rgb );
BYTE GetBValue( DWORD rgb );
```

6.3 Schriftattribute

6.3.1 Merkmale von Schriftarten

Unter Windows kann man Text in vielen verschiedenen Schriftarten darstellen. Jede Schriftart zeichnet sich durch eine Reihe von Merkmalen aus, von denen die wichtigsten *Name*, *Schriftfamilie*, *Größe* und *Gewicht* sind; darüber hinaus gibt es noch eine Reihe weniger wichtiger Eigenschaften, die das Aussehen einer Schriftart beeinflussen.

Der *Name* einer Schriftart ist nach der Installation von Windows oder zusätzlichen Schriften fest vorgegeben. So stehen unter Windows 95 unter anderem »Arial« oder »Courier New« standardmäßig zur Verfügung. Der Name einer Schriftart muß nicht unbedingt bekannt sein, um diese zu selektieren.

Die *Schriftfamilie* unterteilt die Schriftarten in die Gruppen SWISS, ROMAN, MODERN, SCRIPT und DECORATIVE. SWISS bezeichnet serifenlose Proportionalschriften, also Schriften mit einer variablen Buchstabenbreite, die keine Verzierungen an den Enden haben. Beispiele dafür sind »Arial« und alle Schriften, die »Helv« im Namen haben.

ROMAN bezeichnet Proportionalschriften mit Serifen; zu dieser Gruppe gehören alle Schriften mit »Roman« im Namen, also beispielsweise »Times New Roman«, aber auch andere wie z.B. »Garamond« oder »Palatino«.

SCRIPT bezeichnet Schreibschriften (z.B. »Script«), MODERN Nicht-Proportionalschriften mit oder ohne Serifen. Bei nicht-proportionalen Schriften besitzen alle Zeichen dieselbe Breite (wie auf normalen Textbildschirmen), so daß es einfach ist, spaltengetreue Darstellungen zu erzeugen. Beispiele für nicht-proportionale Schriften sind die nach Schreibmaschine aussehende »Courier New« und alle anderen Schriftarten mit »Courier« im Namen, aber auch »Letter Gothic«. DECORATIVE schließlich bezeichnet Effektschriften für besondere Anwendungen, beispielsweise »Old English«, »Mandarin« oder »Accent«.

Die *Größe* einer Schriftart gibt deren Höhe an, die Anpassung der Breite einzelner Zeichen erfolgt automatisch. Im Buchsatz wird die Größe einer Schrift oft in *Punkt [Pt]* angegeben. Verwendet man keine besonderen *Mapping-Modi* in einem Programm, so erwartet Windows die Schriftgröße in Geräteeinheiten, also Bildschirmpixeln.

Das *Gewicht* einer Schrift gibt an, wie dick die Linien und Kurven sind, aus denen sich die einzelnen Zeichen zusammensetzen. Dieses Merkmal kann bei der Windows-Programmierung zwischen 0 und 1000 liegen, dabei steht 0 für unsichtbar und 1000 für völlig schwarz. Normale Schriften besitzen den Wert 400, Fettdruck erzielt man mit 700.

Weitere Merkmale betreffen die Orientierung der Schrift auf dem Bildschirm, den Neigungsgrad der einzelnen Zeichen, die durchschnittliche Breite der Zeichen oder den verwendeten Zeichensatz.

6.3.2 Vordefinierte Schriftarten

Das Einstellen einer neuen Schriftart zur Ausgabe von Text kann auf zwei verschiedene Arten erfolgen:

- ▼ Durch Anlegen einer *logischen Schriftart* mit anschließender Selektion dieser Schriftart in den aktuellen Device-Kontext oder
- ▼ durch Selektieren einer *vordefinierten Schriftart* in den Device-Kontext.

Die erste Möglichkeit wird im nächsten Abschnitt erklärt, denn sie ist etwas komplizierter zu programmieren. Das Selektieren einer vordefinierten Schriftart ist dagegen sehr einfach und wird in *SuchText* genutzt, um die Schriftarten `SYSTEM_FONT` und `ANSI_FIXED_FONT` einzustellen. Es gibt sechs vordefinierte Schriftarten, die auf diese Weise aktiviert werden können (siehe Tabelle 6.3).

Um eine vordefinierte Schriftart in den Device-Kontext zu selektieren, ist die Methode *SelectStockObject* von *CDC* zu verwenden:

```
class CDC:  
    virtual CGdiObject *SelectStockObject( int nIndex );
```

Sie erwartet als Parameter eine Konstante, die das zu selektierende GDI-Objekt angibt. Hier können zur Selektion von Schriftarten direkt die Namen aus Tabelle 6.3 verwendet werden; darüber hinaus kann *SelectStockObject* andere Typen von vordefinierten GDI-Objekten selektieren, die hier nicht behandelt werden sollen (siehe Online-Dokumentation zu *SelectStockObject*).

Bezeichner	Bedeutung
<code>ANSI_FIXED_FONT</code>	nicht-proportionale Schrift in Standardgröße im Windows-Zeichensatz.
<code>ANSI_VAR_FONT</code>	proportionale Schrift in Standardgröße im Windows-Zeichensatz.
<code>DEVICE_DEFAULT_FONT</code>	geräteabhängige Standardschriftart.
<code>OEM_FIXED_FONT</code>	nicht-proportionale Schrift im OEM-Zeichensatz.
<code>SYSTEM_FONT</code>	die proportionale Systemschrift (wird z. B. in Dialogboxen oder Menüs verwendet).
<code>SYSTEM_FIXED_FONT</code>	die nicht-proportionale Systemschrift von Pre-3.0-Windows-Versionen (nur noch aus Kompatibilitätsgründen vorhanden).

Tabelle 6.3: Standardschriften

Der Rückgabewert von *SelectStockObject* ist das *CGdiObject*, das vor dem Aufruf selektiert war. Um diesen Rückgabewert zu verstehen, muß man sich das Konzept der GDI-Objekte im Device-Kontext klarmachen. Jeder Device-Kontext speichert eine ganze Reihe von Merkmalen, welche die Ausgabe auf dem zugeordneten Gerät beeinflussen – schon bekannt ist beispielsweise das Speichern der Schriftfarbe und des Schrifthintergrundes.

Darüber hinaus verwaltet der Device-Kontext sechs unterschiedliche GDI-Objekte, die beim Aufruf bestimmter Ausgaberoutinen benötigt werden:

GDI-Objekt	MFC-Klasse	Bedeutung
Stift	CPen	wird z.B. benötigt, um Linien oder Ellipsen zu zeichnen und bestimmt deren Dicke, Farbe und Muster.
Pinsel	CBrush	wird z.B. benötigt, um eine Region zu füllen oder als Füllmuster beim Zeichnen von Kreisen.
Schriftart	CFont	bestimmt die Schriftart bei der Ausgabe von Text.
Farbpalette	CPalette	erlaubt das Definieren einer Farbpalette für Geräte mit Farbfähigkeiten, um z.B. die Farbverfälschungen durch andere Fenster zu minimieren.
Bitmap	CBitmap	dient als Speichermedium für einen Memory-Device-Kontext.
Region	CRegion	bestimmt die Clipping-Region des Device-Kontexts.

Tabelle 6.4: Die unterschiedlichen GDI-Objekte

Für jedes dieser GDI-Objekte existiert eine eigene MFC-Klasse, deren Name in der mittleren Spalte angegeben ist; sie ist jeweils aus der Klasse *CGdiObject* abgeleitet. Wird nun mit einer der Methoden *SelectObject* (s.u.) oder *SelectStockObject* ein GDI-Objekt eines bestimmten Typs in den Device-Kontext selektiert, so ist der Rückgabewert ein Zeiger auf das bis dahin selektierte Objekt dieses Typs.

Nach dem Aufruf von *SelectStockObject* mit einem der oben erwähnten Bezeichner für Schriftarten ist der Rückgabewert also ein Zeiger auf ein temporär erzeugtes *CFont*-Object, das der bisherigen Schriftart dieses Device-Kontexts entspricht. In *SuchText* wird der Rückgabewert zwischengespeichert, so daß nach der Textausgabe die originale Schriftart wiederhergestellt werden kann:

```
void CChildView::OnPaint()
{
    ...
    //Schriftart einstellen
    CFont newfont,*oldfont;
    switch (nSchrift)
    {
```

```

        case ID_SCHRIFT_SYSTEM :
            oldfont = (CFont*) dc.SelectStockObject(SYSTEM_FONT);
            break;
        case ID_SCHRIFT_ANSI :
            oldfont = (CFont*) dc.SelectStockObject(ANSI_FIXED_FONT);
            break;
        case ID_SCHRIFT_SANSERIF10 :
        case ID_SCHRIFT_SANSERIF24 :
            BOOL ret;
            ret = newfont.CreateFont((nSchrift==
                ID_SCHRIFT_SANSERIF10?-10:-24),
                0,0,0,
                (nSchrift==ID_SCHRIFT_SANSERIF24?700:0),
                (nSchrift==ID_SCHRIFT_SANSERIF24?1:0),
                0,0, ANSI_CHARSET,
                OUT_DEFAULT_PRECIS,
                CLIP_DEFAULT_PRECIS,
                DEFAULT_QUALITY, DEFAULT_PITCH |
                (nSchrift==ID_SCHRIFT_SANSERIF10 ?
                FF_SWISS:FF_ROMAN), NULL);
            oldfont = dc.SelectObject(&newfont);
            break;
    }
    ...
}

```

Listing: Einstellen der Schriftart in OnPaint

Da *SelectStockObject* formal einen Zeiger auf ein *CGdiObject* zurückgibt, muß er in den richtigen Typ konvertiert werden – in diesem Fall in *CFont*.

Die Variable *nSchrift*, mit der die Auswahl der Schriftart gesteuert wird, erhält in der Methode *OnSchrift* ihren Wert:

```

void CChildView::OnSchrift()
{
    nSchrift = GetCurrentMessage()->wParam;
    InvalidateRect(NULL, TRUE);
    UpdateWindow();
}

```

Listing: Wahl der Schriftart über das Menü

Analog zur Vorgehensweise bei der Farbeinstellung wird auch hier über entsprechende Message-Map-Einträge eine einzige Methode bei allen vier Schriftart-Menüpunkten angesteuert und mit *GetCurrentMessage()->wPa-*

ram die tatsächliche Nachricht ermittelt. Die Variable `nSchrift` enthält also nach jedem Aufruf von `OnSchrift` den Wert der gewählten ID, der die gewünschte Schriftart angibt, die in `OnPaint` ausgewählt wird.

6.3.3 Erzeugen eigener Schriftarten

Leider sind die vordefinierten Schriftarten weder skalierbar, noch lassen sich andere Merkmale der Schrift verändern. Daher kommt der Wunsch nach weiteren Möglichkeiten auf. Die zweite Möglichkeit, die Schrift zu verändern, besteht darin, zuerst eine *logische Schriftart* zu erzeugen und sie dann mit der Methode `SelectObject` in den Device-Kontext zu selektieren.

Um eine logische Schriftart zu erzeugen, benötigt man ein leeres `CFont`-Objekt. Diesem wird dann mit der Methode `CreateFont` ein logischer Font zugeordnet:

```
class Cfont:
    BOOL CreateFont(
        int nHeight,
        int nWidth,
        int nEscapement,
        int nOrientation,
        int nWeight,
        BYTE bItalic,
        BYTE bUnderline,
        BYTE cStrikeOut,
        BYTE nCharSet,
        BYTE nOutPrecision,
        BYTE nClipPrecision,
        BYTE nQuality,
        BYTE nPitchAndFamily,
        LPCTSTR lpszFacename
    );
```

An der etwas unhandlichen Parameterliste sieht man schon, daß *Create-Font* eine Unmenge an Möglichkeiten bietet. Die wichtigsten sind nachfolgend erläutert, für alle übrigen wurden Default-Werte verwendet.

Eine ausführliche Beschreibung der Parameter und vordefinierten Konstanten finden Sie in der Online-Dokumentation unter *CreateFont*.



Der folgende Auszug aus `OnPaint` beschäftigt sich mit dem Einstellen der beiden Schriftarten *Sans Serif 10 Pixel* und *Serif 24 Pixel, kursiv, fett*, die durch die Werte der IDs in `nSchrift` bestimmt werden.

```

switch (nSchrift)
{
    ...
    case ID_SCHRIFT_SANSERIF10 :
    case ID_SCHRIFT_SANSERIF24 :
        BOOL ret;
        ret = newfont.CreateFont(
            (nSchrift==ID_SCHRIFT_SANSERIF10?-10:-24),
            0,0,0,
            (nSchrift==ID_SCHRIFT_SANSERIF24?700:0),
            (nSchrift==ID_SCHRIFT_SANSERIF24?1:0),
            0,0,
            ANSI_CHARSET, OUT_DEFAULT_PRECIS,
            CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
            DEFAULT_PITCH |
            (nSchrift==ID_SCHRIFT_SANSERIF10 ?
             FF_SWISS:FF_ROMAN),NULL);
        oldfont = dc.SelectObject(&newfont);
        break;
}

```

Listing: Die Anwendung von CreateFont

Der erste Parameter, `nHeight`, gibt die Größe in logischen Koordinaten an, interessanterweise muß der Wert negativ angegeben werden, damit das komplette Zeichen (inklusive Unterlängen) als Maßstab verwendet wird. Die Werte von -10 bzw. -24 wählen also eine Schriftart aus, die 10 bzw. 24 Bildschirmpixel hoch ist. Die nächsten drei Parameter werden auf 0 gesetzt, um die Voreinstellungen auszuwählen. `nWeight` bekommt bei der Schriftart *24 Pixel Serif, kursiv, fett* den Wert 700, um Fettschrift einzustellen, ansonsten 0, um die Voreinstellung zu wählen. Ebenfalls bei dieser Schriftart bekommt der nächste Parameter, `bItalic`, den Wert 1, um anzuzeigen, daß Schrägschrift gewählt werden soll, ansonsten ist auch dieser Parameter 0. Auch die nächsten 6 Parameter enthalten wieder Voreinstellungen. Der vorletzte Parameter ist `nPitchAndFamily` und dient zur Auswahl der Schriftfamilie; er muß aus einer ODER-Kombination der Bitflags `DEFAULT_PITCH`, `FIXED_PITCH`, `VARIABLE_PITCH` und `FF_DECORATIVE`, `FF_DONTCARE`, `FF_SWISS`, `FF_ROMAN`, `FF_SCRIPT`, `FF_MODERN` bestehen. `SuchText` wählt entsprechend dem Wert in `nSchrift` eine Serifenschrift oder eine serifenlose Schrift aus und überläßt die Entscheidung, ob die Schrift proportional sein soll, dem Font-Mapper. Der letzte Parameter gibt den Namen der Schriftart an. Er ist NULL, um anzuzeigen, daß das Programm keinen Wert auf eine Schriftart mit einem bestimmten Namen legt.

Mit Hilfe der Methode `CreateFont` wird keine neue Schriftart erzeugt, sondern aus den vorhandenen Schriftarten die am besten passende ausgewählt. Diese wird durch den nachfolgenden Aufruf von `SelectObject` in den aktuellen Device-Kontext selektiert und bestimmt so das Aussehen des nachfolgenden Textes. An der Übergabe von `NULL` für den Namen der Schriftart erkennt man, daß die tatsächliche Schrift vom Font-Mapper aufgrund der angegebenen Eigenschaften ausgewählt wird – man muß den Namen gar nicht kennen. Als Programmierer kann man daher allerdings nie ganz sicher sein, welche Schriftart wirklich für die Ausgabe verwendet wird – falls eine Schrift mit den angegebenen Eigenschaften nicht verfügbar ist, wählt der Font-Mapper aus den vorhandenen Schriften die am besten passende.

Bei der SDK-Programmierung mußte man sich als Programmierer selbst darum kümmern, daß Objekte, die in den Device-Kontext selektiert wurden, nach Gebrauch gelöscht wurden – dazu gab es den Aufruf von `DeleteObject`. Ein häufiger Fehler bestand darin, irgendwo einen `DeleteObject`-Aufruf zu vergessen und damit in jedem `OnPaint` ein wenig Ressourcen zu verbrauchen, die nicht wieder zurückgegeben wurden. Als MFC-Programmierer hat man es da leichter: Verwendet man die aus `CGdiObject` abgeleiteten Klassen zur Erzeugung und Manipulation von GDI-Objekten, so sorgt der Destruktor `~CGdiObject` dafür, daß `DeleteObject` automatisch aufgerufen wird, wenn das Objekt zerstört wird. Aus diesem Grund brauchen die benötigten Objekte nur als lokale Klasseninstanzen in `OnPaint` angelegt zu werden, und das Thema »Löschen« kann vergessen werden. Wichtig ist es allerdings, daß vor Aufruf des Destruktors eines GDI-Objekts dieses aus dem Device-Kontext deselektiert wurde; dies kann z.B. dadurch geschehen, daß am Ende von `OnPaint` das jeweils älteste Objekt wieder selektiert wird:

```
...  
dc.SelectObject(oldfont);
```

6.4 Ausgaben des Textes

6.4.1 Die Datenstruktur zur Darstellung des Textes

Zur Speicherung der gefundenen Textzeilen wird im Programm eine relativ einfache Struktur verwendet. Sie ist zum Einlesen kleinerer Dateien gut geeignet, für größere sollten aber andere Methoden verwendet werden. Jede Zeile der Textdatei wird in ein `CString`-Objekt eingelesen. Dieses Objekt der MFC speichert eine Zeichenkette und stellt Funktionen zur String-Manipulation bereit. Jede eingelesene Zeile, die die gesuchte Zeichenkette enthält, wird an ein Array angehängt. Somit besteht dieses aus `CString`-Objekten. Auch dafür existiert in der MFC eine Klasse – `CStringArray`. Die Variable `nZeilen` speichert die Anzahl der eingelesenen Zeilen.

CString besitzt eine Reihe von Konstruktoren, die für verschiedene Datentypen gedacht sind. Interessant ist der Operator »=«, mit dessen Hilfe dem *CString*-Objekt Zeichenketten zugewiesen werden. Damit läßt sich das Objekt wie eine Variable ansprechen.

```
CString( );
CString( const CString& stringSrc );
CString( TCHAR ch, int nRepeat = 1 );
CString( LPCTSTR lpch, int nLength );
CString( const unsigned char* psz );
CString( LPCWSTR lpwsz );
CString( LPCSTR lpasz );
```

CStringArray ist eine von *CObArray* abgeleitete Klasse für ein Array von *CString*-Objekten. Nach dem Anlegen des Arrays werden mit der Methode *Add* die *CString*-Objekte an das Array angehängt. Mittels der Methode *GetAt* oder dem Operator »[]« kann man auf einzelne Elemente zugreifen.



Eine genaue Beschreibung der Klasse *CString* finden Sie in Kapitel 11.1; dort wird auch der Umgang mit den Array-Klassen erläutert.

Der auf dem Bildschirm dargestellte Ausschnitt der Textdatei wird durch die Variable *nErsteZeile* bestimmt: sie ist die erste angezeigte Zeile. Der Wert von *nErsteZeile* liegt immer zwischen 0 und *nZeilen*-1. Die Variable wird durch Benutzereingaben verändert. Wie dies geht, wird im nächsten Abschnitt ausführlich erklärt; für den Moment reicht es aus anzunehmen, daß die Variable irgendwelche festen Werte innerhalb ihres Gültigkeitsbereichs angenommen hat.

6.4.2 Größenmerkmale der Schrift

Die Merkmale der ausgewählten Schrift werden durch den Aufruf von *CreateFont* und *SelectObject* nicht eindeutig bestimmt. Je nach Display-Adapter, installierten Schriften und anderen Eigenschaften der aktuellen Hardwareumgebung wählt der Font-Mapper jeweils die am besten passende Schriftart aus. Um Textausgaben korrekt positionieren zu können, muß das Programm daher zur Laufzeit in der Lage sein, Informationen über die Abmessungen der einzelnen Zeichen der ausgewählten Schriftart einzuholen; wichtig ist vor allem die Höhe der Zeichen. Zu diesem Zweck gibt es in der Klasse *CDC* die Methode *GetTextMetrics*:

```
class CDC:
    BOOL GetTextMetrics( LPTEXTMETRIC lpMetrics) const;
```

GetTextMetrics liefert eine Reihe von Größenangaben über die aktuelle Schriftart, die sie in der per Zeiger übergebenen Struktur *lpMetrics* ablegt.

lpMetrics ist ein FAR-Zeiger auf eine Struktur vom Typ *TEXTMETRIC*, aus der alle wichtigen Informationen der aktuellen Schriftart abgelesen werden können. Tabelle 6.5 beschreibt einige der Member, die für Größenbestimmungen wichtig sind.

Alle Größenangaben werden in logischen Einheiten angegeben – also in Bildschirmpixeln, wenn nicht explizit ein anderes Einheitensystem gewählt wurde. Um Text zeilenweise auszugeben, ist es wichtig, die Höhe einer Zeile zu kennen: sie ergibt sich aus der Summe *tmHeight* + *tmExternalLeading* (Abbildung 6.1). Befindet sich eine bestimmte Textzeile etwa an der vertikalen Position *y1*, so sollte die darunterliegende Zeile mindestens an *y1* + *tmHeight* + *tmExternalLeading* beginnen, damit es zu keinen Überschneidungen kommt.

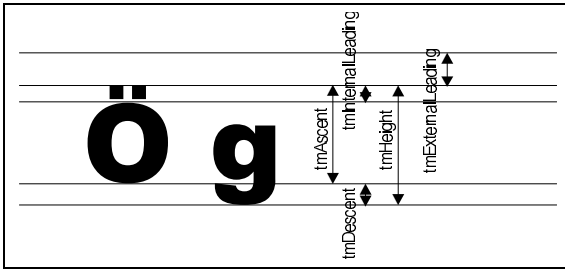


Abbildung 6.1: Kennwerte von Schriften

Weitere Informationen zu der Struktur *TEXTMETRIC* finden sich in der elektronischen Dokumentation unter *TEXTMETRIC-Structure* und in der Win32 SDK-Dokumentation.



Member	Bedeutung
int <i>tmHeight</i>	Höhe der Zeichen; ist die Summe aus <i>tmAscent</i> und <i>tmDescent</i> .
int <i>tmAscent</i>	Oberlänge der Zeichen, also die Höhe von der Grundlinie bis zum oberen Ende.
int <i>tmDescent</i>	Unterlänge der Zeichen, also die Höhe vom unteren Ende bis zur Grundlinie.
int <i>tmExternalLeading</i>	Abstand zwischen übereinanderliegenden Textzeilen, um zu verhindern, daß beispielsweise die Punkte von Umlauten mit den Unterlängen der darüberliegenden Zeile zusammenwachsen.
int <i>tmAveCharWidth</i>	Durchschnittliche Breite eines Zeichens.
int <i>tmMaxCharWidth</i>	Maximale Breite eines Zeichens.

Tabelle 6.5: Parameter von *TEXTMETRIC*

Während die Höhe einer Schriftart für alle Zeichen gleich ist, hat jedes Zeichen seine eigene Breite. Die Member *tmAveCharWidth* und *tmMaxCharWidth* von *TEXTMETRIC* sind also nur zu gebrauchen, um die Länge eines Textstücks ganz grob zu überschlagen. Soll der genaue Wert ermittelt werden, kann die Methode *GetTextExtent* von *CDC* verwendet werden:

```
class CDC:
    CSize GetTextExtent( LPCTSTR lpszString, int nCount ) const;
```

GetTextExtent erwartet eine Zeichenkette in *lpszString* und eine Längenangabe in *nCount*. Der Rückgabewert ist die Länge der ersten *nCount*-Zeichen von *lpszString*, wenn sie in der aktuellen Schriftart des Device-Kontexts ausgegeben werden. Das Ergebnis wird in einem Objekt der Klasse *CSize* abgeliefert, das im wesentlichen aus den Datenmembern *x* und *y* und einigen überlagerten Operatoren besteht; dabei gibt *x* die Ausdehnung der Zeichenkette in positiver x- und *y* diejenige in positiver y-Richtung an. Diese Werte werden ebenfalls in logischen Koordinaten zurückgegeben.

Um die korrekte Längenangabe einer Zeichenkette zu erhalten, sollte man *GetTextExtent* immer mit der kompletten Zeichenkette als Argument aufrufen; das Addieren der Länge von Teilstrings kann zu Fehlern führen, wenn das Ausgabegerät in der Lage ist, Zeichen automatisch zu unterschneiden. Das Unterschneiden ist ein übliches satztechnisches Verfahren, bei dem bestimmte Buchstabenpaare näher zusammenrücken (als ihre Breite dies normalerweise erlauben würde), weil der zweite Buchstabe in eine Aussparung des ersten paßt; Beispiele sind »Te« oder »AV«.

Neben den hier erwähnten Methoden, mit denen die Abmessungen von Schriftarten bestimmt werden können, gibt es noch weitere, die verwandte Aufgaben erledigen.

6.4.3 Textausrichtung

Während bei normalen Textbildschirmen die Ausgabe eines Textes immer links beginnt und sich nach rechts fortsetzt, ist dies bei der Windows-Programmierung konfigurierbar. Denkt man sich um den auszugebenden Text ein gerade passendes Rechteck, so kann der Ursprung des Textes aus einer Kombination von jeweils drei möglichen horizontalen und drei möglichen vertikalen Positionen bestehen. Horizontal kann der linke oder rechte Rand des Textes oder dessen Mitte gewählt werden, vertikal der obere oder untere Rand oder die Grundlinie der Schriftzeichen.

Standardmäßig ist der Ausgangspunkt für die Textausgabe die linke obere Ecke, er kann mit Hilfe der Methode *SetTextAlign* von *CDC* verändert werden:

```
class CDC:
    UINT SetTextAlign( UINT nFlags );
```

nFlags ist eine ODER-Verknüpfung eines der vertikalen Bitflags *TA_CENTER*, *TA_LEFT*, *TA_RIGHT* mit einem der horizontalen Bitflags *TA_TOP*, *TA_BOTTOM* oder *TA_BASELINE*. Mit Hilfe dieser Methode ist es möglich, Text ohne Rechenaufwand zentriert oder rechtsbündig auszugeben.

6.4.4 Die Textausgabe mit *ExtTextOut*

Es gibt bei der Windows-Programmierung eine Reihe von Funktionen, die Text auf dem Bildschirm ausgeben; einige sind sehr einfach zu bedienen, andere dagegen schwieriger. Die in *SuchText* verwendete Funktion *ExtTextOut* ist recht aufwendig zu programmieren, bietet aber die meisten Möglichkeiten. Sie wurde vor allem deshalb verwendet, um den Hintergrund der Client-Area auch bei inverser Darstellung lückenlos ausfüllen zu können.

ExtTextOut erwartet in *x* und *y* zunächst die Bildschirmkoordinaten, an denen mit der Textausgabe begonnen werden soll. In welche Richtung die Textausgabe läuft, ist von der Textausrichtung abhängig, die zuvor mit Hilfe von *SetTextAlign* ausgewählt wurde. Der Parameter *nOptions* bestimmt die Interpretation des nachfolgenden Rechtecks *lpRect*. Er kann eines der Bitflags *ETO_OPAQUE* oder *ETO_CLIPPED* oder eine ODER-Verknüpfung beider annehmen. Die Konstante *ETO_OPAQUE* sorgt dafür, daß das Rechteck an den Stellen ohne Text in der aktuellen Hintergrundfarbe gefüllt wird, *ETO_CLIPPED* beschränkt die Ausgabe auf das angegebene Rechteck *lpRect*.

```
class CDC:
    virtual BOOL ExtTextOut(    int x, int y,
                               UINT nOptions, LPCRECT lpRect,
                               LPCTSTR lpszString, UINT nCount,
                               LPINT lpDxWidths );

    BOOL ExtTextOut( int x, int y,
                    UINT nOptions, LPCRECT lpRect,
                    const CString& str,
                    LPINT lpDxWidths );
```

Sowohl *lpRect* als auch *x* und *y* sind in Client-Koordinaten anzugeben, haben also ihren Ursprung in der linken oberen Ecke des Fensters. Die nächsten beiden Parameter *lpszString* und *nCount* liefern den Ausgabe-string in der bekannten Aufteilung in Daten- und Längenangabe. Der letzte Parameter wird selten gebraucht; er zeigt auf ein Array von *int*-Wer-

ten, welches die horizontale Positionierung der auszugebenden Zeichen festlegt; hier kann NULL angegeben werden, wenn der Parameter nicht benutzt wird.

Zur Vorbereitung der Textausgabe in *SuchText* wird die Textausrichtung auf die linke obere Ecke eingestellt und mit *GetTextMetrics* die Größe der aktuellen Schrift in der TEXTMETRIC-Struktur *tm* ermittelt. Darüber hinaus bestimmt *OnPaint* die Größe des Client-Bereichs durch Aufruf der Methode *GetClientRect* und legt das Ergebnis in der *CRect*-Variablen *winrect* ab:



```
void CSuchTextView::OnPaint()
{
    ...
    CRect drawrect,winrect;
    TEXTMETRIC tm;
    //Anzahl Zeilen ermitteln
    nZeilen = aZeilen->GetSize();
    //Textausrichtung
    dc.SetTextAlign(TA_TOP | TA_LEFT);
    dc.GetTextMetrics(&tm);
    //Fenstergröße
    GetClientRect(winrect);
    drawrect=winrect;
    nZeilenProSeite = winrect.bottom /
                      (tm.tmHeight + tm.tmExternalLeading);
    //Text ausgeben
    for (int i=0;i<nZeilen ; ++i)
    {
        drawrect.bottom = drawrect.top+tm.tmHeight+
                          tm.tmExternalLeading;
        if (nErsteZeile+i<nZeilen)
        {
            dc.ExtTextOut(drawrect.left+1,
                          drawrect.top,
                          ETO_CLIPPED|ETO_OPAQUE,
                          drawrect,
                          aZeilen->GetAt(nErsteZeile+i),
                          strlen(aZeilen->GetAt(
                          nErsteZeile+i)), NULL);
        }
        else
        {

```



```

        dc.ExtTextOut(drawrect.left+1,
            drawrect.top,
            ETO_CLIPPED|ETO_OPAQUE,
            drawrect, "", 0, NULL);
    }
    //neue Schreibposition
    drawrect.top += tm.tmHeight +
        tm.tmExternalLeading;
    if (drawrect.top>=winrect.bottom) break;
}
dc.SelectObject(oldfont);
}

```

Listing: Textausgabeteil von OnPaint

Die Textausgabe erfolgt nun zeilenorientiert von oben nach unten, dabei spielt das Rechteck *drawrect* die Rolle des Ausgabebereichs der aktuellen Zeile; seine linke obere Ecke *drawrect.left* und *drawrect.top* liefert die *x*- und *y*-Koordinaten für den Aufruf von *ExtTextOut*, gleichzeitig ist *drawrect* selbst das Clipping-Rechteck *lpRect*.

Jeweils vor der Ausgabe einer Zeile ist die obere Kante von *drawrect* korrekt eingestellt, so daß die untere Kante durch Addition von *tm.tmHeight* + *tm.tmExternalLeading* bestimmt werden kann, während nach der Ausgabe die untere Kante durch Addition eben dieser Zeilenhöhe auf den richtigen Wert gesetzt wird. Zur Ausgabe der aktuellen Zeile muß nur noch überprüft werden, ob die aktuelle Zeile *nErsteZeile+i* kleiner als die Anzahl der eingelesenen Zeilen *nZeilen* ist. Ist dies der Fall, wird der Inhalt des Textarrays *aZeilen[nErsteZeile+i]* ausgegeben, andernfalls wird ein Leerstring ausgegeben.

Wenn am Ende der Schleife nach Berechnung der neuen Oberkante von *drawrect* festgestellt wird, daß *drawrect* außerhalb des Client-Bereichs liegt, so bricht die Schleife ab, und die Textausgabe ist beendet. Diese Abbruchbedingung stellt sicher, daß die letzte anzuzeigende Zeile auch dann ausgegeben wird, wenn sie nur noch unvollständig auf den Bildschirm paßt. Da der Clipping-Bereich bei Aufruf der *OnPaint*-Methode auf den Client-Bereich beschränkt ist, werden Textausgaben außerhalb des Client-Bereichs unterdrückt.

6.4.5 Einfachere Formen der Textausgabe

ExtTextOut ist eine Methode, die sich einerseits durch große Flexibilität auszeichnet, andererseits erfordert sie einigen Aufwand bei der Programmierung. Soll einfach nur ein Textstring auf dem Bildschirm ausgegeben werden, so genügt oft die Methode *TextOut*, die wesentlich einfacher zu programmieren ist:

TextOut gibt es in zwei Variationen: Die erste erwartet neben den Koordinaten *x* und *y* der linken oberen Ecke einen Zeiger *lpszString* auf den Textpuffer und die Anzahl *nCount* der auszugebenden Zeichen. Die zweite Variante ist eine Überlagerung, bei der die Parameter *lpszString* und *nCount* durch ein *CString*-Objekt *str* ersetzt werden. Die Klasse *CString* dient zur Darstellung von Zeichenketten, deren Länge und Speicherbedarf zur Laufzeit variabel ist (siehe *CString*). An Stelle eines *CString*-Objekts kann aber auch ein normaler *char*-Zeiger übergeben werden; er wird dann mit einem in der Klasse vorhandenen Konstruktor automatisch in ein *CString*-Objekt konvertiert.

Beide Varianten von *TextOut* geben den Text mit der linken oberen Ecke an der Position *x*, *y* aus und benutzen die eingestellte Schriftart des Device-Kontexts.

```
class CDC:
    virtual BOOL TextOut( int x, int y,
                        LPCTSTR lpszString,
                        int nCount );
    BOOL TextOut( int x, int y,
                const CString& str );
```

Der wichtigste Unterschied zu *ExtTextOut* ist das Fehlen des Ausgaberechtecks *lpRect*, mit dem der Hintergrund eingefärbt wird. Würde *SuchText* lediglich *TextOut* zur Ausgabe verwenden, so ergäbe sich beim Scrollen der Effekt, daß kurze Textzeilen vorhergehende längere Textzeilen nicht vollständig überschreiben – am rechten Rand würden Fragmente der längeren Textzeile stehen – bleiben. Durch Übergabe eines Ausgaberechtecks, das die Breite der gesamten Client-Area hat, wird in *ExtTextOut* dafür gesorgt, daß nicht beschriebene Bereiche am rechten Rand in der Farbe des Hintergrunds eingefärbt (und damit scheinbar gelöscht) werden.

Neben *TextOut* und *ExtTextOut* gibt es noch zwei weitere Funktionen zur Textausgabe, *DrawText* und *TabbedTextOut*, die beide zur Klasse *CDC* gehören. *DrawText* ist eine Erweiterung von *TextOut* und bietet einige zusätzliche Möglichkeiten der Textformatierung wie die Interpretation von Tabulatoren, die mehrzeilige Ausgabe und den automatischen Wortumbruch am Zeilenende.

Die Methode *TabbedTextOut* ist ebenfalls eine Erweiterung von *TextOut*, ihr wird als wesentliche Erweiterung ein Array mit Tabulatorpositionen übergeben. Damit ist es leicht, spaltenorientierte Ausgaben zu erzeugen, selbst wenn die Spalten eine unterschiedliche Breite haben. Beide Methoden sind in den Referenzen ausführlich dokumentiert (siehe *DrawText*, *TabbedTextOut* in der Online-Dokumentation).

6.5 Bildschirmausgaben außerhalb von *OnPaint*

Die *OnPaint*-Routine eines Fensters muß in der Lage sein, den aktuellen Bildschirminhalt »aus dem Gedächtnis« zu rekonstruieren, d.h., sie muß alle Bildschirmobjekte kennen, die in der Client-Area ausgegeben werden sollen. Oft ist es aber gar nicht nötig, den ganzen Client-Bereich zu restaurieren, beispielsweise dann, wenn nur ein kleiner Teil der Ausgabe vom Programm geändert werden soll. Mit den bisher bekannten Mitteln müßte die Anwendung zuerst die Datenstruktur für *OnPaint* aktualisieren, dann eine WM_PAINT-Nachricht schicken und in *OnPaint* mit viel Aufwand herausfinden, welcher Teil der Client-Area neu gezeichnet werden muß – oder sie müßte gleich *alles* neu zeichnen.

Glücklicherweise ist es möglich, auch außerhalb von *OnPaint* Ausgaben in der Client-Area vorzunehmen, man braucht dazu nur einen Device-Kontext für die Client-Area. Bei der SDK-Programmierung gibt es dafür die Funktion *GetDC*. Sie besorgt zuerst einen Handle auf den Device-Kontext für den Client-Bereich, dann kann gezeichnet werden und zum Schluß muß der Device-Kontext mit *ReleaseDC* wieder freigegeben werden.

Bei der MFC-Programmierung geht es etwas einfacher, denn man muß nur ein lokales Objekt der Klasse *CClientDC* anlegen und erhält einen Device-Kontext für den Client-Bereich des Fensters. *CClientDC* ist aus *CDC* abgeleitet und ruft in ihrem Konstruktor die Funktion *GetDC* auf. Ebenso wie *CPaintDC* kümmert sie sich auch um die Freigabe des Device-Kontexts: in ihrem Destruktor wird automatisch *ReleaseDC* aufgerufen.

```
class CClientDC:
    CClientDC( CWnd *pWnd );
    ~CClientDC();
```

Soll beispielsweise eine Methode *OnHello* existieren, die in der linken oberen Ecke des Client-Bereichs das Wort »hello!« ausgibt, so kann dies ganz einfach programmiert werden:

```
void CMainFrame::OnHello()
{
    CClientDC dc(this);
    dc.TextOut(1,1,"hello!");
}
```

Listing: Beispiel einer Textausgabe in einer Nachrichtenfunktion

Diese Ausgabe ist allerdings nicht sehr beständig: Soll sie nach der nächsten WM_PAINT-Nachricht nicht verschwunden sein, muß *OnPaint* davon unterrichtet werden, daß in der linken oberen Ecke das Wort »hello!« zu stehen hat und es ebenfalls ausgeben.

Neben *CClientDC* gibt es noch die Klasse *CWindowDC*, die einen Device-Kontext für das komplette Fenster liefert, also Client-Area und Non-Client-Area. Damit ist es möglich, auch in der Titelleiste, der Menüleiste und allen anderen Elementen außerhalb des Client-Bereichs Ausgaben vorzunehmen (siehe *CWindowDC*). *CWindowDC* führt einen Aufruf von *GetWindowDC* im Konstruktor und von *ReleaseDC* im Destruktor aus.

Einen Device-Kontext für den kompletten Bildschirm kann man nicht über eine vordefinierte MFC-Klasse bekommen, dafür muß nach wie vor die SDK-Funktion *GetDC* mit dem Wert NULL im Parameter *hwnd* aufgerufen werden (siehe *GetDC*).



Der bis hierher beschriebene Stand des Beispielsprogramms *SuchText* ist im Verzeichnis `SUCHTEXT\STEP3` zu finden. Einige der verwendeten Variablen wurden in der Header-Datei definiert und teilweise im Konstruktor von *CChildView* initialisiert!



Bildlaufleisten und Scrollen

7

Kapitelübersicht

7.1	Anlegen der Bildlaufleisten	230
7.2	Initialisieren der Bildlaufleisten	233
7.3	WM_HSCROLL und WM_VSCROLL	234
7.4	Reaktion des Programms auf Scroll-Nachrichten	236
7.5	Beispielprogramm	239

7.1 Anlegen der Bildlaufleisten

Eine der wesentlichen Fähigkeiten von *SuchText* sollte es sein, mit Hilfe der Tastatur oder der Maus im Text zu blättern. Gängigen Windows-Konventionen folgend, erledigt man dies am besten, indem man das Hauptfenster mit horizontalen und vertikalen Bildlaufleisten ausstattet. Die Bedienung dieser Bildlaufleisten erzeugt dann Nachrichten, die an das Hauptfenster gesendet werden und durch Aufruf geeigneter Methoden zu den gewünschten Effekten führen.

Bevor es losgeht, sei noch kurz etwas zur Terminologie gesagt. Es macht zwar Sinn, von *Bildlaufleisten* zu sprechen, wenn damit tatsächlich ein Bildausschnitt verschoben werden kann; in der Praxis tauchen aber vor allem in Dialogboxen oft »Bildlaufleisten« auf, die andere Aufgaben haben. In diesem Fall paßt dieser Begriff nicht mehr und sollte statt dessen durch einen sinnvolleren ersetzt werden. Hier sollen daher die Begriffe *Schieberegler* oder die oft verwendete direkte Übersetzung *Rollbalken* des englischen Originals *Scrollbar* verwendet werden (nicht zu verwechseln mit dem Regler oder Slider!). Der kleine Knopf, den der Benutzer mit der Maus direkt bewegen kann, soll *Schieber* heißen.

Um das Hauptfenster mit Bildlaufleisten auszustatten, brauchen beim Aufruf von *CFrameWnd::Create* lediglich die Fensterflags *WS_VSCROLL* und *WS_HSCROLL* an *dwStyle* übergeben zu werden. In diesem Fall bekommt das Fenster sowohl eine vertikale als auch eine horizontale Bildlaufleiste.

```
CMainFrame::CMainFrame()
{
    LoadAccelTable("MainAccelTable");
    Create(NULL,
        "TestScrollbar",
        WS_OVERLAPPEDWINDOW | WS_VSCROLL |
        WS_HSCROLL, rectDefault,
        NULL, "MainMenu");
    ShowScrollBar(SB_BOTH, FALSE);
}
```

Listing: Aktivieren der Bildlaufleisten

Am Ende der Routine befindet sich ein Aufruf von *ShowScrollBar*, mit dessen Hilfe die Bildlaufleisten abgeschaltet, also unsichtbar gemacht werden. Erst nach dem Anzeigen von Text werden sie durch einen erneuten Aufruf von *ShowScrollBar* wieder aktiviert, diesmal mit *TRUE* als zweitem Parameter. Beim Schließen einer Datei wird die Methode *OnFileClose* aufgerufen, und beide Bildlaufleisten werden wieder unsichtbar gemacht. In der Methode *OnFind* wird *ShowScrollBar* mit den entsprechenden Parametern aufgerufen, um sie zur Anzeige zu bringen.

```
void CChildView::OnDateiSuchen()
{
    ...
    ShowScrollBar(SB_VERT, TRUE);
    SetScrollRange(SB_VERT, 0, nZeilen-1, FALSE);
    ...
}
```

Listing: Einblenden der vertikalen Bildlaufleiste

Der erste Parameter von *ShowScrollBar* gibt an, ob die vertikale (SB_VERT), die horizontale (SB_HORZ) oder beide Bildlaufleisten (SB_BOTH) von der Änderung betroffen sein sollen, der zweite regelt, ob an- (TRUE) oder ausgeschaltet (FALSE) werden soll.

```
class Cwnd:
    void ShowScrollBar( UINT nBar, BOOL bShow = TRUE );
```

Die oben beschriebene Methode eignet sich nur dann, wenn wirklich *CMainFrame* als Ausgabefenster genutzt wird. Wird jedoch, wie in unserem Fall, mit einer Ansicht gearbeitet, muß etwas anders vorgegangen werden. Dies ist aber keinesfalls komplizierter. Bei Verwendung der Dokument-Ansicht-Architektur kann von vornherein die Klasse *CScrollView* als Basisklasse für die Ansicht mit Bildlaufleisten verwendet werden.

Weitere Informationen dazu und zur Arbeit mit Ansichten finden Sie in Kapitel 13.



Weiterhin besteht auch hier die Möglichkeit, die entsprechenden Window-Styles zu setzen und damit die Bildlaufleisten anzuzeigen. Da *CMainFrame* aber das Hauptfenster bildet, und die eigentlichen Ausgaben im Kindfenster der Klasse *CChildView* erfolgen, müssen dafür die Bildlaufleisten eingeschaltet werden. Um Änderungen an den Styles von Fenstern vorzunehmen, existiert die Methode *PreCreateWindow*, die Arbeiten übernimmt, die vor dem Anzeigen von Fenstern erledigt werden müssen.

```
class Cwnd:
    virtual BOOL PreCreateWindow( CREATESTRUCT& cs );
```

Diese Methode stammt aus der Klasse *CWnd* und wird automatisch vom Framework vor dem Anzeigen des Fensters aufgerufen. Sie darf niemals direkt aufgerufen werden. Auch die MFC selbst nutzt diese Methode, um Einstellungen vorzunehmen. Diese sind allerdings nicht dokumentiert und können nur im Quellcode der MFC betrachtet werden. Als Parameter wird die Variable *cs* vom Typ *CREATESTRUCT* übergeben. Diese Struktur besitzt unter anderem den Parameter *style*; er enthält Window-Styles, die

das Aussehen und das Verhalten des Fensters beeinflussen. Man kann die Parameter komplett überschreiben oder ergänzen. Beim `style`-Parameter empfiehlt sich die Ergänzung um die Styles `WS_VSCROLL` und `WS_HSCROLL`, um die Bildlaufleisten hinzuzufügen. Dies wird durch eine ODER-Verknüpfung erreicht.

```
BOOL CChildView::PreCreateWindow(CREATESTRUCT& cs)
{
    ...
    //Scrollleiste hinzufügen
    cs.style |= WS_VSCROLL | WS_HSCROLL;
    ...
}
```

Listing: Hinzufügen von Bildlaufleisten

Damit werden die Bildlaufleisten sichtbar. Eine andere Anwendungsmöglichkeit ist das Verändern der Größe des Fensters und der Elemente in der Titelzeile. Dies erfolgt im Hauptrahmenfenster der Klasse *CMainFrame*.

```
BOOL CChildView::PreCreateWindow(CREATESTRUCT& cs)
{
    ...
    //Größe des Fensters festlegen
    cs.cy = ::GetSystemMetrics(SM_CYSCREEN) / 2;
    cs.cx = ::GetSystemMetrics(SM_CXSCREEN) / 2;
    //Position des Fensters
    cs.y = ((cs.cy * 2) - cs.cy) / 2;
    cs.x = ((cs.cx * 2) - cs.cx) / 2;
    ...
}
```

Listing: Größe und Position des Fensters verändern



Weitere Hinweise auf Window-Styles und die hier beschriebenen Techniken finden Sie in der Online-Dokumentation unter `CREATESTRUCT`, *Window-Styles* und *PreCreateWindow*.

Nun ist es mit dem Anlegen der Bildlaufleisten allein aber noch nicht getan. Würde das Programm nichts weiter tun, könnte der Anwender zwar die Schieberegler manipulieren, einen Effekt auf das Programm hätte dies aber nicht. Folgende Dinge sind zusätzlich zu erledigen:

- ▼ Die Schieber müssen initialisiert werden.
- ▼ Das Programm muß Methoden zur Verfügung stellen, die auf Nachrichten der Bildlaufleisten reagieren.

- ▼ Diese Methoden müssen entsprechend den Benutzeranforderungen sowohl den dargestellten Textausschnitt verschieben als auch für die korrekte Darstellung der Schieber sorgen.

7.2 Initialisieren der Bildlaufleisten

Zum Initialisieren der Bildlaufleisten muß zunächst einmal deren *Regelbereich* angegeben werden. Dieser wird durch eine benutzerdefinierbare Unter- und Obergrenze festgelegt, deren Differenz nicht mehr als INT_MAX betragen darf. Diese Aufgabe wird mit Hilfe der Methode *SetScrollRange* von *CWnd* erledigt:

```
class Cwnd:
    void SetScrollRange( int nBar, int nMinPos,
                       int nMaxPos, BOOL bRedraw = TRUE );
```

Der Parameter *nBar* muß einen der Werte SB_HORZ oder SB_VERT annehmen, wodurch festgelegt wird, ob sich der Aufruf dieser Methode auf den horizontalen oder vertikalen Schieberegler auswirken soll. Mit *nMinPos* und *nMaxPos* werden Unter- und Obergrenze des Regelbereichs festgelegt, die Endpunkte gehören dazu. Der letzte Parameter *bRedraw* gibt an, ob der Schieberegler nach der Veränderung des Regelbereichs neu gezeichnet werden soll, um die Veränderungen sichtbar zu machen.

Nachdem mit *SetScrollRange* der Regelbereich eines Schiebereglers eingestellt wurde, wirken sich die neuen Einstellungen sowohl beim Abfragen als auch beim Setzen der Schieberposition aus. Beim Abfragen wird immer der Wert zurückgegeben, der dem Wegverhältnis des Schiebers zwischen Anfangs- und Endpunkt entspricht; beim Setzen des Schiebers muß die Positionsangabe innerhalb der angegebenen Grenzen liegen. Ein Wert von *nMinPos* stellt dabei den Schieber auf den Anfang, *nMaxPos* auf das Ende des Regelbereichs.

Die Voreinstellung des Regelbereichs ist 0 bis 100 (= 101 Schritte!). Dies bleibt solange gültig, bis mit *SetScrollRange* etwas anderes festgelegt wird. Wenn dieser Bereich akzeptabel ist, kann man auch auf den Aufruf von *SetScrollRange* verzichten. Für eine Bildlaufleisten-Steuerung ist der Regelbereich leer (siehe Kapitel 10.4)!

Nach dem Aufruf von *SetScrollRange* ist die tatsächliche Position des Schiebers undefiniert, sie muß erst mit Hilfe der Methode *SetScrollPos* der Klasse *CWnd* festgelegt werden:

```
class Cwnd:
    int SetScrollPos( int nBar, int nPos, BOOL bRedraw = TRUE );
```

`nBar` gibt wieder an, ob der vertikale oder der horizontale Schieberegler verändert werden soll, auch hier ist einer der Werte `SB_HORZ` oder `SB_VERT` anzugeben. Mit `nPos` wird die neue Position des Schiebers festgelegt; dieser Wert muß im vorher definierten Regelbereich des Schiebereglers liegen. Der Parameter `bRedraw` hat dieselbe Aufgabe wie in *SetScrollRange*. Er gibt an, ob der Schieber neu gezeichnet werden soll oder nicht. Falls er `FALSE` ist, bleibt die Änderung der Position des Schiebers für den Benutzer unsichtbar.

Es ist noch wichtig, zu wissen, an welcher Stelle auf dem Bildschirm der Regelbereich beginnt, und wo er endet. Bei einem vertikalen Schieberegler ist der Anfang oben und das Ende unten, d. h., durch Aufruf von *SetScrollPos*(`SB_VERT`, 0) wird der Schieber ganz nach oben gestellt. Bei einem horizontalen Schieberegler ist der Anfang links und das Ende rechts; hier stellt ein Aufruf von *SetScrollPos*(`SB_HORZ`, 0) den Schieber ganz nach links.

In *SuchText* erfolgt das Festlegen des Regelbereichs nach dem Suchen in der Methode *OnDateiSuchen*. Der vertikale Schieber darf Werte zwischen 0 und der Anzahl der eingelesenen Zeilen minus 1 annehmen.

Diese Werte wurden gewählt, um die aktuelle Position des vertikalen Schiebers exakt mit der Variablen `nErsteZeile` zu synchronisieren; diese gibt an, welches die erste auf dem Bildschirm dargestellte Zeile ist. Wie noch zu sehen sein wird, wirken sich Veränderungen an den Bildlaufleisten direkt auf diese Variable aus. Zusätzlich bestimmt der Inhalt von `nErsteZeile` durch Aufruf der privaten Methode *UpdateDisplay* von *CSuchTextView* die visuelle Stellung des Schiebers; aus diesem Grund tauchen in *OnDateiSuchen* keine Aufrufe von *SetScrollPos* auf.

7.3 WM_HSCROLL und WM_VSCROLL

Nimmt der Benutzer Änderungen an den Bildlaufleisten vor, so bekommt das zugehörige Fenster die Nachricht `WM_HSCROLL` oder `WM_VSCROLL` zugeschickt, je nachdem, welcher der beiden Schieberegler davon betroffen ist. Diese Nachrichten können mit den Message-Map-Einträgen `ON_WM_HSCROLL` und `ON_WM_VSCROLL` abgefangen werden. Bei Vorhandensein eines `ON_WM_HSCROLL`-Eintrags muß in der Fensterklasse eine Methode *OnHScroll* realisiert werden, bei `ON_WM_VSCROLL` die Methode *OnVScroll*:

```
class Cwnd:
afx_msg void OnHScroll( UINT nSBCode,
                        UINT nPos,
                        CWnd *pScrollBar );
afx_msg void OnVScroll( UINT nSBCode,
                        UINT nPos,
                        CWnd *pScrollBar );
```

Der wichtigste Parameter ist `nSBCode`. Er gibt an, welches Element eines Schiebereglers der Benutzer bedient hat. `nSBCode` kann die Werte aus Tabelle 7.1 annehmen.

nSBCode	Benutzeraktion	Reaktion des Programms
SB_LINEUP	nach Drücken des oberen (linken) Buttons	Der Bildausschnitt soll um den kleinstmöglichen Schritt nach oben (links) verschoben werden, bei Texten beispielsweise um eine Zeile (Spalte).
SB_LINEDOWN	nach Drücken des unteren (rechten) Buttons	Der Bildausschnitt soll um den kleinstmöglichen Schritt nach unten (rechts) verschoben werden.
SB_PAGEUP	nach Drücken der Fläche zwischen oberem (linkem) Button und Schieber	Der Bildausschnitt soll um eine Seite nach oben (links) verschoben werden, bei Texten um so viele Zeilen (Spalten), wie in dem aktuellen Fensterausschnitt Platz haben.
SB_PAGEDOWN	nach Drücken der Fläche zwischen unterem (rechtem) Button und Schieber	Der Bildausschnitt soll um eine Seite nach unten (rechts) verschoben werden, bei Texten um so viele Zeilen, wie in dem aktuellen Fensterausschnitt Platz haben.
SB_THUMBPOSITION	nach Bewegen des Schiebers	Der Bildausschnitt soll an die Position in den Daten verschoben werden, die dem Verhältnis zwischen aktueller Schieberposition und gesamtem Regelbereich entspricht. Hierbei gibt <i>nPos</i> die aktuelle Schieberposition an.
SB_THUMBTRACK	während der Bewegung des Schiebers	Wie SB_THUMBPOSITION.

Tabelle 7.1: Werte für `nSBCode`

Neben diesen Werten kann `nSBCode` noch die Werte `SB_TOP`, `SB_BOTTOM` und `SB_ENDSCROLL` annehmen. Die ersten beiden sollen (gemäß offizieller Dokumentation) den Anfang oder das Ende des Scrollingbereichs anzeigen, werden aber in *SuchText* nicht berücksichtigt. `SB_ENDSCROLL` zeigt dagegen das Ende einer zusammenhängenden Aktion von Benutzermanipulationen an. `nSBCode` wird beispielsweise nach dem Loslassen der linken Maustaste mit diesem Wert belegt, wenn vorher auf dem unteren oder oberen Schiebereglerknopf dauerhaft gedrückt wurde, ansonsten aber auch nach Abschluß jeder Interaktion des Benutzers mit dem Schieberegler, die eine `WM_SCROLL`-Nachricht erzeugt hat.

Der letzte Parameter beider Methoden, `pScrollBar`, wird nicht genutzt, falls die Schieberegler wie hier feste Bestandteile eines Fensters sind. Sind sie hingegen Steuerungen in einer Dialogbox, so zeigen sie auf ein Objekt vom Typ `CScrollBar`, das den Schieberegler repräsentiert.

7.4 Reaktion des Programms auf Scroll-Nachrichten

Die Reaktion auf die WM_?SCROLL-Nachrichten obliegt vollständig der Anwendung, sogar die Neupositionierung des Schiebers muß manuell vorgenommen werden. Dazu wird mit dem Klassen-Assistenten eine neue Member-Funktion angelegt (Abbildung 7.1).



Abbildung 7.1: Anlegen von UpdateDisplay

Am Ende wird die private Methode `UpdateDisplay` aufgerufen und erledigt das Positionieren der Schieber und den Neuaufbau des Bildschirms:

```
void CChildView::UpdateDisplay()
{
    if (nErsteZeile < 0) nErsteZeile = 0;
    if (nErsteZeile > nZeilen)
        nErsteZeile = nZeilen;
    SetScrollPos(SB_VERT, nErsteZeile, TRUE);
    InvalidateRect(NULL, FALSE);
    UpdateWindow();
}
```

Listing: Die private Funktion UpdateDisplay

Darüber hinaus führt `UpdateDisplay` alle nötigen Bereichsüberprüfungen der Variablen `nErsteZeile` durch. Das hat den Vorteil, daß diese Tests an einer einzigen Stelle zusammengefaßt sind und nicht hinter jedem *case-Zweig* in `OnHScroll` und `OnVScroll` ausprogrammiert werden müssen. Die Routine `UpdateDisplay` ist sicherlich ein gutes Beispiel für eine Gratwanderung zwischen *notwendiger Länge* (gleich Lesbarkeit) und *erwünschter Kürze* (gleich Produktivität). Ob es immer sinnvoll ist, so zu programmieren, mag dahingestellt bleiben; wegen der Einfachheit von *SuchText* (und seiner umfassenden Dokumentation) ist es hier jedoch vertretbar.

Etwas anders sieht die Methode `OnVScroll` aus, die für den vertikalen Schieberegler zuständig ist:

```
void CSuchTextView::OnVScroll(UINT nSBCode, UINT nPos,
                             CScrollBar* pScrollBar)
{
    switch (nSBCode)
    {
        case SB_LINEUP :   OnZeileZurueck(); break;
        case SB_LINEDOWN : OnZeileVor(); break;
        case SB_PAGEUP :   OnAnsichtSeitezurueck(); break;
        case SB_PAGEDOWN : OnAnsichtSeitevor(); break;
        case SB_THUMBPOSITION :
        case SB_THUMBTRACK :
            nErsteZeile = nPos;
            UpdateDisplay();
            break;
    }
    CWnd::OnVScroll(nSBCode, nPos, pScrollBar);
}
```



Listing: Reaktion auf die Nachricht WM_VSCROLL

Der Unterschied besteht darin, daß beim schrittweisen Scrollen die Veränderung der Variablen `nErsteZeile` indirekt durch Aufruf anderer Methoden erledigt wird. Das liegt darin begründet, daß diese Methoden sowieso realisiert werden mußten, um die *Blättern*-Funktionen über die Tastatur und das ANSICHT-Menü zu implementieren. Also können sie auch von *OnVScroll* aufgerufen werden, was zusätzlich den Vorteil hat, daß sich die »durchschnittliche Wiederverwendungszahl« unserer Methoden erhöht.

Diese Vorgehensweise gilt nicht für die direkte Manipulation des Schiebers. In diesem Fall wird `nErsteZeile` an Ort und Stelle auf den gewünschten Wert eingestellt. Da der Wertebereich des vertikalen Schiebereglers mit der Anzahl der eingelesenen Zeilen übereinstimmt, bleibt der Textausschnitt immer im sichtbaren Bereich.

Eine Besonderheit findet sich in den Methoden *OnAnsichtSeitezurueck* und *OnAnsichtSeitevor* von *CChildView*, dort wird `nErsteZeile` nämlich um den Wert `nZeilenProSeite` verändert. Dieser Wert ist eine Programmvariable, die immer die Anzahl der Zeilen im Client-Bereich angibt. Diese kann sich durchaus während der Laufzeit des Programms ändern, nämlich genau dann, wenn eine andere Schriftart ausgewählt oder die Höhe des Hauptfensters verändert wurde.

Um nun nicht nach jedem dieser Ereignisse `nZeilenProSeite` neu errechnen zu müssen, erledigt *SuchText* dies in *OnPaint* – dort fällt der Wert praktisch als Nebenprodukt an: Es ist nämlich der ganzzahlige Quotient aus Höhe des Client-Bereichs und Höhe der aktuellen Schriftart.

```
void CChildView::OnPaint()
{
    ...
    nZeilenProSeite = winrect.bottom /
        (tm.tmHeight + tm.tmExternalLeading);
    ...
}
```

Das bedeutet allerdings, daß `nZeilenProSeite` bis zum ersten *OnPaint*-Aufruf undefiniert ist. Das macht aber nichts, denn beim Aufbau des Hauptfensters – also garantiert vor dem ersten Aufruf von *OnAnsichtSeitezurueck* oder *OnAnsichtSeitevor* – wird schon eine `WM_PAINT`-Nachricht an das Fenster gesendet.

Bisher ist noch nicht deutlich herausgestellt worden, worin der Unterschied zwischen `SB_THUMBPOSITION` und `SB_THUMBTRACK` besteht, denn beide *nSBCodes* werden beim Bewegen des Schiebers gegeben. Der Unterschied besteht darin, daß `SB_THUMBPOSITION` erst *nach* der Positionierung gesendet wird (also nach dem Loslassen der linken Maustaste), während die Nachricht `SB_THUMBTRACK` bereits *während* der Bewegung der Maus erfolgt. `SB_THUMBTRACK` dient dazu, dem Benutzer ein gewisses Feedback beim Bewegen des Schiebers zu geben, indem die Ausgabe synchron mitgescrollt wird.

Die Implementierung von `SB_THUMBTRACK` macht aber nur dann Sinn, wenn der Client-Bereich schnell genug neu aufgebaut werden kann, andernfalls behindert sie den Benutzer zu sehr. Da die gefundenen Textzeilen in einem Array im Hauptspeicher gehalten werden, ist die Ausgabeperformance in unserem Fall ausreichend, um `SB_THUMBTRACK` ohne Probleme zu implementieren.

Ausblick

In diesem Abschnitt wurden die wesentlichen Dinge erklärt, die man über Bildlaufleisten im Hauptfenster wissen muß. Das hier vorgestellte Verfahren, um Text zu scrollen, kann prinzipiell ebenso auf grafische Ausgaben angewendet werden. Dabei kann es aber im Einzelfall wesentlich schwieriger sein, eine geeignete Datenstruktur zur Speicherung der Ausgabeelemente zu finden.

Neben den Schiebereglern im Hauptfenster gibt es Schieberegler auch als Steuerungen in Dialogboxen, die Programmierung ist allerdings fast vollkommen identisch. Zudem gibt es noch einige Möglichkeiten, Schieberegler zu manipulieren, die hier nicht erwähnt wurden (beispielsweise das Deaktivieren der Buttons), aber in der Online-Dokumentation nachgelesen werden können (siehe *EnableScrollBar* und *CScrollBar*).

Weiterhin existiert die vordefinierte Ansichts-Klasse mit Bildlaufleisten – *CScrollView*.

7.5 Beispielprogramm

Das Beispielprogramm liegt in SUCHTEXT\STEP4. Falls noch nicht geschehen, sollten sie noch Beschleunigertasten zum Blättern im Text definieren. Des weiteren wurden noch zwei Funktionen zum zeilenweisen Blättern angelegt. Die IDs wurden in der Beschleunigertabelle (ID_ZEILE_VOR, ID_ZEILE_ZURUECK) für die Cursorbewegung definiert, und mit dem Klassen-Assistenten wurden die zugehörigen Member-Funktionen angelegt.

Implementieren Sie das Scrollen in horizontaler Richtung. Dies ist besonders bei großen Schriftarten erforderlich, wenn nicht der gesamte Text auf eine Zeile paßt. Gehen Sie dabei in ähnlicher Art und Weise wie beim vertikalen Scrollen vor. Eine Variable dient zum Speichern der linken Position, die über die Bildlaufleiste festgelegt wird. Der Wertebereich liegt zwischen Null und der maximalen Zeilenlänge. Bei der Ausgabe in *OnPaint* wird der scheinbar nicht sichtbare Teil der Zeichenkette einfach abgeschnitten. Dazu kann die Funktion *Mid* der Klasse *CString* verwendet werden (siehe Kapitel 11).



Der gesamte Quelltext zu *SuchText* ist auf der CD zu finden. Im Verzeichnis SUCHTEXT\STEP5 finden Sie auch die Lösung der Aufgabe.





Fehlersuche mit dem Browser

8

Kapitelübersicht

8.1	Arbeitsweise	243
8.2	Das Hauptfenster des Browsers	244
8.3	Vererbungshierarchie darstellen	245
8.3.1	Abgeleitete Klassen	245
8.3.2	Basisklassen	246
8.4	Aufrufabhängigkeiten darstellen	246
8.4.1	Aufgerufene Funktionen	246
8.4.2	Aufrufende Funktionen	247
8.5	Definitionen und Referenzen anzeigen	248
8.6	Integration des Browsers in das Developer Studio	249

Die Zeiten, in denen Computerprogramme aus einigen Hunderten oder vielleicht Tausenden Zeilen Quertet bestanden, sind vorbei. Heute sind Programme in der Regel sehr viel größer: Sie bestehen aus Hunderten oder Tausenden von Funktionen, mehreren zehntausend oder hunderttausend Zeilen Quelltext und werden von vielen Programmierern parallel entwickelt.

Um gute Programme zu schreiben, ist es erforderlich, die Beziehungen zwischen den Programmelementen zu kennen. Bevor die Welle der objektorientierten Programmierung begann, wollte man vor allem die Abhängigkeiten zwischen aufrufenden und aufgerufenen Funktionen oder zwischen Definition und Verwendung von Variablen kennen, was üblicherweise mit Hilfe von Cross-Referenzlisten untersucht wurde. Mit der objektorientierten Programmierung und der Möglichkeit, Eigenschaften von Klassen zu vererben, kam eine weitere bedeutungsvolle Beziehung ins Spiel, nämlich die zwischen Basisklassen und abgeleiteten Klassen.

Gewöhnlich wird in einer guten objektorientierten Entwicklungsumgebung die Vererbungshierarchie der Klassen mit Hilfe eines Quelltext-Browsers untersucht, der in der Lage ist, die Relationen zwischen Vater- und Sohnklassen grafisch darzustellen. Neben diesen Fähigkeiten bietet der Browser von Visual C++ die Möglichkeit, alle anderen der genannten Beziehungen darzustellen. So zeigt er neben der Vererbungshierarchie auf Wunsch auch die Aufrufhierarchie an oder findet die Definition eines Symbols – und zwar jeweils unabhängig davon, ob es dabei um Quelldateien des aktuellen Projekts oder der Foundation Classes geht. Im Detail bietet der Browser folgende grafische Darstellungsmöglichkeiten:

- ▼ alle Klassen, die aus einer beliebigen Basisklasse abgeleitet wurden;
- ▼ alle Basisklassen einer beliebigen Klasse;
- ▼ alle Funktionen, die von einer beliebigen Funktion aufgerufen werden;
- ▼ alle Stellen, von denen aus eine beliebige Funktion aufgerufen wird;
- ▼ alle Definitionen und Referenzen eines beliebigen Symbols.

Mit diesen Fähigkeiten geht der Browser von Visual C++ deutlich über das hinaus, was in objektorientierten Systemen normalerweise üblich ist, und stellt damit eines der wichtigsten Hilfsmittel für den Entwickler dar.

Microsoft geht in der Enterprise Edition noch einen Schritt weiter. Zur Unterstützung des objektorientierten Entwurfsprozesses liegt der *Visual Modeler* dem Paket bei. Dieser erlaubt es, mit der UML¹-Sprache Klassen grafisch zu modellieren und die Modelle dann in fertigen Quellcode zu überführen.

¹ UML – Unified Modeling Language

8.1 Arbeitsweise

Der Browser läuft als normales, nicht modales Fenster innerhalb des Developer Studios ähnlich dem *Eigenschaften*-Fenster. Er verfügt ebenfalls über einen *Push-Pin*, mit dem das Fenster bei Verlust des Fokus sichtbar bleibt. Der Browser arbeitet mit einer *Informationsdatei*. Bei dieser handelt es sich um die *Browser-Datenbank*, die beim Übersetzen eines Projekts erstellt wird. Sie hat denselben Namen wie das Projekt, aber die Erweiterung *.BSC*, also *SUCHTEXT.BSC* für das *SuchText*-Projekt.

Üblicherweise wird die Browser-Datenbank am Ende des Erstellungsprozesses aus den SBR-Dateien erzeugt. Diese werden für jede Quelldatei angelegt und dann zur BSC-Datei zusammengefaßt. Ohne diese Dateien ist der Browser nicht nutzbar.

Standardmäßig werden die Browser-Informationen nicht angelegt. Um sie anzulegen, muß im Menü PROJEKT der Punkt EINSTELLUNGEN (**Alt** **F7**) aufgerufen werden. Im Register *Browse Information* muß das Kontrollkästchen *Browse Informationsdatei erstellen* aktiviert werden. Damit wird die BSC-Datei angelegt. Sicherzustellen ist noch, daß auch die SBR-Files angelegt werden. Das erfolgt im Register *C/C++* und der Kategorie *Listing Dateien*. *Browse-Info generieren* ist dort zu aktivieren.

Leider wird durch diese Einstellung der Erstellungszyklus verlangsamt, so daß es aus Gründen der Performance sinnvoll sein kann, sie vorübergehend zu deaktivieren. Auch wenn überhaupt nicht mit dem Browser gearbeitet werden soll, kann diese Option deaktiviert werden. Ein Mittelweg könnte darin bestehen, die BSC-Datei nur bei Bedarf zu erstellen. So werden immer die aktuellen SBR-Dateien erzeugt. Der Browser arbeitet so mit einer vorhandenen Datenbank, die nicht aktuell ist. Für viele Zwecke reicht dies aus.

Nach dem Erstellen der Browser-Datenbank muß sie noch aktiviert werden. Dies kann entweder über den Menüpunkt EXTRAS|QUELLCODE BROWSER erfolgen oder manuell, indem die BSC-Datei per DATEI|ÖFFNEN geöffnet wird. Mit EXTRAS|QUELLCODE BROWSER oder **Alt** **F12** wird gleichzeitig der Browser bzw. ein Fenster zur Auswahl der Kategorie des Browsers geöffnet (Abbildung 8.1).

Nach der Auswahl der gewünschten Abfragekategorie der Browser-Datenbank wird der Browser mit dem als *Bezeichner* angegebenen Parameter angezeigt. Dieser wird entweder dem Wort entnommen, in dem der Cursor steht, oder entstammt einem markierten Bereich. Er kann aber auch manuell eingegeben werden.

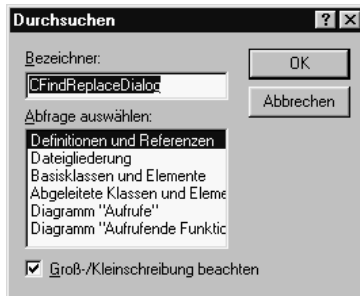


Abbildung 8.1: Auswahl einer Browser-Kategorie (in deutsch: Durchsuchen)

8.2 Das Hauptfenster des Browsers

Das *Browser*-Fenster besteht im wesentlichen aus drei Teilen (siehe Abbildung 8.2):

- ▼ der Darstellung der Quelle im linken Teil
- ▼ der Darstellung der Member-Funktionen und -Variablen (falls vorhanden)
- ▼ der Darstellung der Referenzen und Definition im rechten Teil

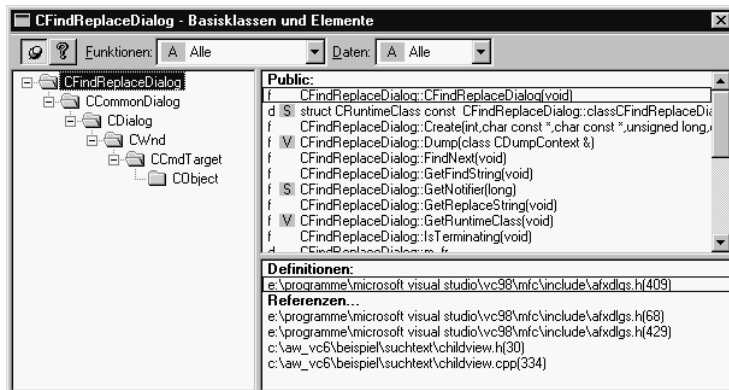


Abbildung 8.2: Browser – Basisklassen und Elemente-Ansicht

Während die üblichen Browser objektorientierter Sprachen im ersten Fenster die Vererbungshierarchie, im zweiten die Methoden der ausgewählten Klasse und im dritten den zugehörigen Quelltext anzeigen, ist der Aufbau der Informationsfenster im Visual-C++-Browser etwas anders. Da er neben Vererbungshierarchien noch andere Abfragearten kennt, muß man zunächst die gewünschte Abfrage formulieren (Abbildung 8.1). Das grafische Ergebnis dieser Abfrage wird immer im ersten (ganz linken) Informa-

tionsfenster dargestellt, der Aufbau der übrigen Fenster variiert je nach Art der Abfrage. Bei einem Vererbungsgraphen zeigt das zweite beispielsweise die Member der ausgewählten Klasse und das dritte die Definition und Verwendung des ausgewählten Members an. In keinem Fall enthält das letzte Fenster den tatsächlichen Quelltext, sondern immer nur Referenzen darauf – allerdings ist es damit sehr einfach, ein Quelltextfenster genau an der angegebenen Position zu öffnen. Dazu muß nur ein Doppelklick mit der Maus auf die entsprechende Zeile erfolgen.

Zur grafischen Darstellung werden im ersten Fenster Ordner verwendet. Diese können entweder geschlossen oder offen sein. Die Anzeige erfolgt in einer Baumansicht und entspricht der Philosophie des Explorers bei der Anzeige von Verzeichnissen. Sind unter dem aktuellen Pfad noch weitere vorhanden, so wird in der Struktur ein Pluszeichen dargestellt. Durch einen Mausklick darauf läßt sich die Struktur expandieren. Natürlich kann auch eine der Basisklassen mit der Maus angewählt werden, woraufhin dessen Member-Funktionen, -Variablen sowie Definitionen und Referenzen im rechten Fenster angezeigt werden.

8.3 Vererbungshierarchie darstellen

8.3.1 Abgeleitete Klassen

Das wichtigste Diagramm in einer komplexen Klassenbibliothek ist zunächst das der Klassenhierarchie. Dazu ist in der Liste die Abfrageart *Abgeleitete Klassen und Elemente* auszuwählen und unter *Bezeichner* der Name der gewünschten Basisklasse anzugeben. Danach wird ein Diagramm der Basisklasse und aller daraus abgeleiteten Klassen im ersten Informationsfenster angezeigt (siehe Abbildung 8.3).

Durch Anklicken einer der angezeigten Klassen können nun weitere Informationen beschafft werden. Das zweite Fenster (rechts oben) zeigt alle Methoden und Member-Variablen der Klasse, und das dritte gibt an, wo die ausgewählte Klasse definiert und wo sie verwendet wurde. Durch Anklicken einer der Methoden oder Member-Variablen im zweiten Fenster wird die Information im dritten Fenster für das ausgewählte Element spezifiziert. Zusätzliche Filtereinstellungen können in den Listboxen *Funktionen* und *Daten* vorgenommen werden. Damit läßt sich die Anzeige zum Beispiel auf *static member functions* begrenzen.

Soll von einer der im dritten Fenster angezeigten Referenzen direkt auf die zugehörige Stelle im Quelltext gesprungen werden, so wird nur die entsprechende Zeile doppelgeklickt, und im Editor des Developer Studios wird die Datei geladen und korrekt positioniert.

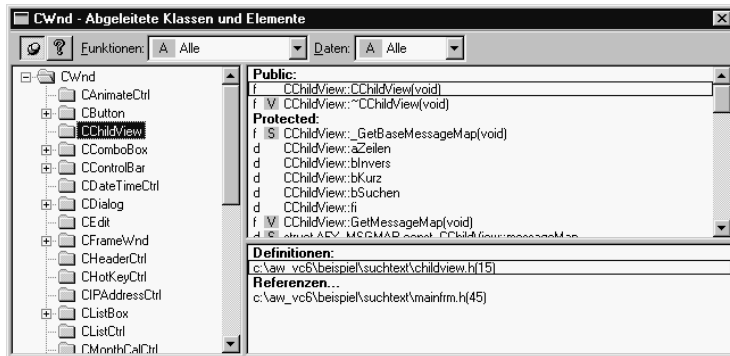


Abbildung 8.3: Ansicht der abgeleiteten Klassen und Elemente

8.3.2 Basisklassen

Neben der Anzeige aller abgeleiteten Klassen ist es auch möglich, alle Basisklassen einer bestimmten Klasse anzuzeigen. Hier geht man genauso wie bei den abgeleiteten Klassen vor, lediglich ist bei der Abfrageart *Basisklassen und Elemente* anzugeben. Nach der Abfrage wird im ersten Informationsfenster die gewählte Klasse mit ihrer Vaterklasse, deren Vaterklasse usw. angezeigt, bis das Ende der Vererbungskette erreicht ist. Falls keine mehrfache Vererbung verwendet wurde, ist der Graph eine lineare Liste, mehrfache Vaterklassen werden baumartig dargestellt (Abbildung 8.2).

Die Darstellung und der Inhalt der anderen Informationsfenster stimmt mit denen von *Abgeleitete Klassen* überein.

8.4 Aufrufabhängigkeiten darstellen

8.4.1 Aufgerufene Funktionen

Wie eingangs erwähnt, bietet der Browser die Möglichkeit, Abhängigkeiten zwischen Funktionsaufrufen zu untersuchen. Mit der Abfrageart *Diagramm Aufrufe* werden alle Funktionen angezeigt, die innerhalb der Funktion aufgerufen werden, die unter *Bezeichner* angegeben wurde.

Bei der Interpretation des Ergebnisses darf man sich nicht von der optischen Ähnlichkeit mit den vorigen Abfragearten verwirren lassen. Was hier angezeigt wird, hat nichts mit Abhängigkeiten zwischen Vater- und Kindklassen zu tun. Das Diagramm gibt lediglich eine Liste aller Funktionsaufrufe aus, die im Quelltext einer Funktion zu finden sind. Falls der angezeigte Graph mehr als zweistufig ist, so bedeutet dies nichts anderes, als daß eine aufgerufene Funktion ihrerseits wieder Funktionen aufruft. Das Ende der Darstellung ist erreicht, wenn eine Funktion keine weiteren Funktionen mehr aufruft oder wenn für diesen Aufruf keine Quellen mehr zur Verfügung stehen.

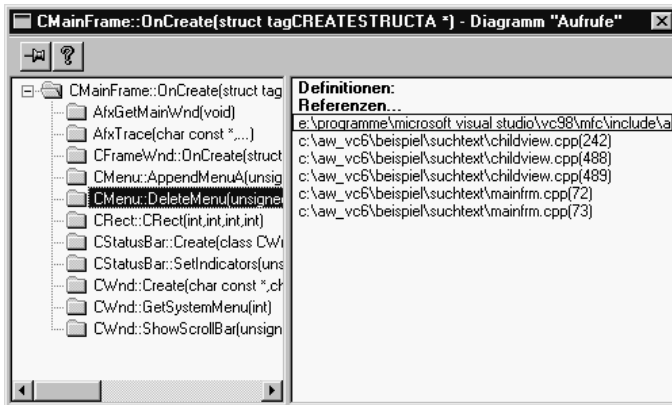


Abbildung 8.4: Ansicht der Funktionsaufrufe innerhalb einer Funktion

Bei dieser Abfrageart gibt es neben der grafischen Darstellung nur noch ein einziges Informationsfenster. Es zeigt die Definitionen und Referenzen der im ersten Fenster markierten Funktion an. Auch hier kann per Doppelklick direkt in einen der angezeigten Quelltexte gesprungen werden.

8.4.2 Aufrufende Funktionen

Ganz ähnlich wie die vorige Abfrageart funktioniert die Option *Diagramm Aufrufende Funktionen*. Sie zeigt für eine beliebige Funktion an, von welchen anderen Funktionen sie aufgerufen wird, und setzt dies rekursiv für alle gefundenen Aufrufer fort. Erst, wenn sich kein Aufrufer mehr findet – etwa weil die Funktion nur noch von MFC-internen Funktionen oder als Callback-Funktion aufgerufen wird –, ist die Darstellung zu Ende.

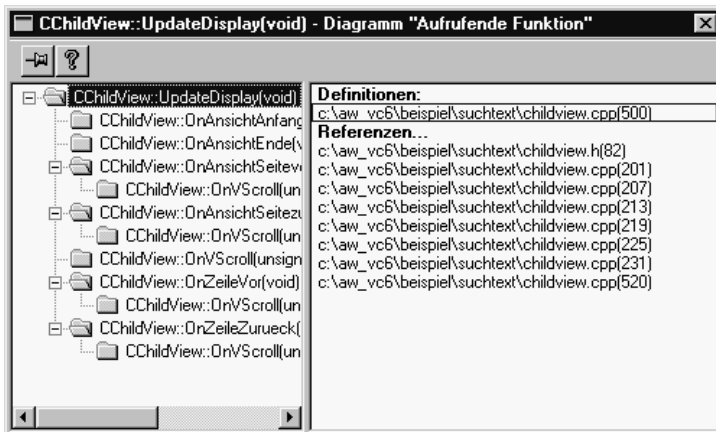


Abbildung 8.5: Ansicht der aufrufenden Funktionen einer Methode

8.5 Definitionen und Referenzen anzeigen

Die vorletzte Abfrageart des Browsers bietet die Möglichkeit, nach Definitionen oder Referenzen von Symbolen zu suchen. So kann für beliebige Funktionen, Klassen, Variablen, Typen oder Makros die Stelle gesucht werden, an der sie definiert, und die Stellen, an denen sie verwendet wurden.

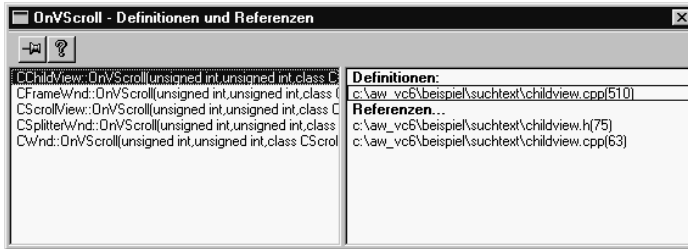


Abbildung 8.6: Definitionen und Referenzen von Variablen oder Funktionen

Dazu ist als Abfrageart *Definitionen und Referenzen* zu verwenden und unter *Bezeichner* der Name des gesuchten Symbols einzugeben. Auch diese Abfrageart besitzt nur zwei Informationsfenster: Das erste zeigt die gefundenen Vorkommen des Symbols an, und das zweite gibt an, wo Definition und Referenzen zu finden sind. Wird für die Abfrage der Begriff *On** eingegeben, so wirkt dieser wie ein Platzhalter in Dateinamen, so daß alle Bezeichner angezeigt werden, die mit *On* beginnen.

Die letzte Abfrage-Möglichkeit des Browsers bezieht sich auf eine komplette Datei. Wird für *Bezeichner* nichts eingetragen (oder der Name der Datei plus Erweiterung) und die Abfrageart *Dateigliederung* gewählt, so werden dem Filter entsprechend alle Elemente der Datei im Editor angezeigt. Mittels der Filter-Tasten können Klassen, Funktionen, Variablen, Makros und Typ-Definitionen ausgewählt werden.



Abbildung 8.7: Ansicht der Dateigliederung

8.6 Integration des Browsers in das Developer Studio

Da der Start des Browsers über die Tastenkombination **[Alt] [F12]** oder das Menü **EXTRAS|QUELLCODE BROWSER** und anschließender Auswahl der Abfrage-Kategorie sehr umständlich und zeitaufwendig ist, gibt es eine Reihe weiterer Möglichkeiten, den Browser zu starten.

Als erstes sei die *Durchsuchen*-Symbolleiste genannt. Sie kann bei Bedarf eingeblendet werden und bietet auf Tastendruck den Aufruf der entsprechenden Abfrage. Als Bezeichner wird die aktuelle Cursorposition ausgewertet.



Abbildung 8.8: Die Symbolleiste Durchsuchen

Die Symbolleiste umfaßt folgende Punkte:

- ▼ Sprung zur Definition des Bezeichners; Taste **[F12]**
- ▼ Sprung zur Referenz des Bezeichners; Tasten **[⇧] [F12]**
- ▼ Sprung zur vorhergehenden Definition/Referenz; Tasten **[Strg] [Num] [-]**
- ▼ Sprung zur nächsten Definition/Referenz; Tasten **[Strg] [Num] [+]**
- ▼ Rücksprung zu dem Punkt vor Beginn des letzten Durchsuchens **[Strg] [Num] [*]**
- ▼ Dateigliederung
- ▼ Definitionen und Referenzen
- ▼ abgeleitete Klassen und Elemente
- ▼ Basisklassen und Elemente
- ▼ Diagramm »Aufrufe«
- ▼ Diagramm »Aufrufende Funktionen«

Einige der Funktionen der Symbolleiste können direkt über Funktionstasten erreicht werden. Diese Tasten werden im *Tooltip* mitangezeigt. (Tool-tips sind die Fenster mit der Bezeichnung einer Funktion. Sie werden angezeigt, wenn die Maus eine gewisse Zeit über einem Element ruht.) Es können auch eigene Tastenzuordnungen für Funktionen des Browsers erstellt werden.

Des weiteren besteht an jeder Stelle im Quelltext oder im *Arbeitsbereich*-Fenster die Möglichkeit, mit der rechten Maustaste ein Kontextmenü zu öffnen. Darin sind auch die möglichen Funktionen des Browsers enthalten (Abbildungen 8.9 und 8.10).



Abbildung 8.9: Möglichkeiten des Kontextmenüs im Editor-Fenster

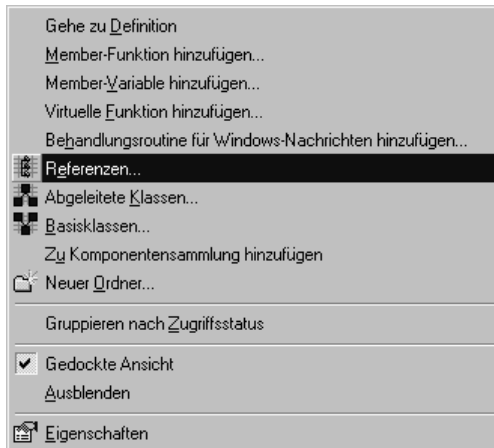


Abbildung 8.10: Das Kontextmenü einer Klasse im Arbeitsbereich-Fenster



Der integrierte Debugger

9

Kapitelübersicht

9.1	Haltepunkte (Breakpoints)	253
9.1.1	Standard-Haltepunkte	253
9.1.2	Komplexere Breakpoints	254
9.1.3	Haltepunkte im Register »Pfad«	255
9.1.4	Haltepunkte im Register »Daten«	255
9.1.5	Haltepunkte im Register »Nachrichten«	257
9.2	Programmausführung	257
9.3	Programmvariablen	258
9.3.1	Das Variablen-Fenster	258
9.3.2	Das Fenster Überwachung	259
9.3.3	Das QuickWatch-Fenster	260
9.4	Verschiedenes	261
9.4.1	Die Aufrufliste	261
9.4.2	Das Register-Fenster	261
9.4.3	Das Speicher-Fenster	261
9.4.4	Assembler-Listings einblenden	261
9.4.5	Ausnahmen	261
9.4.6	Threads	262
9.4.7	Module	262
9.5	Weitere Debugging-Möglichkeiten	263
9.5.1	TRACE-Makro	263
9.5.2	ASSERT-Makro	263
9.5.3	Memory Leaks	264

Bisher wurden nur Programme betrachtet, die von Anfang an fehlerfrei waren – was während der normalen Entwicklung einer Anwendung natürlich nicht zu erwarten ist. Statt dessen treten Laufzeitfehler auf, die nur mit Hilfe eines guten Debuggers gefunden werden können. Glücklicherweise gehört zum Developer Studio ein integrierter Debugger, mit dessen Hilfe sich Windows-Programme innerhalb der Entwicklungsumgebung auf etwaige Fehler hin überprüfen lassen.

Die Zeiten, zu denen man zum Debuggen noch einen zweiten Bildschirm brauchte oder mit einem statischen textorientierten *Debugger*-Fenster vorliebnehmen mußte, sind glücklicherweise vorbei. Der integrierte Debugger bietet einen Großteil der Fähigkeiten früherer Codeview-Versionen, ohne deren Nachteile zu haben. Mit seiner Hilfe können Programme im Einzelschrittmodus getestet werden. Es ist möglich, verschiedene Arten von Unterbrechungspunkten zu setzen, man kann sich Variablen ansehen oder verändern und noch vieles mehr.

Der Debugger arbeitet mit maximal sieben Fenstern, in denen verschiedene Informationen ausgegeben werden:

Fenster	Taste	Information
Variablen	(Alt)(4)	Variablen einer Funktion, lokale Variablen, <i>this</i> -Zeiger
Speicher	(Alt)(6)	zeigt einen Ausschnitt aus dem Hauptspeicher
Disassemblierung	(Alt)(8)	zeigt die Assembler-Funktionen an, die für jede C++-Zeile ausgeführt werden (im <i>Editor</i> -Fenster)
Register	(Alt)(5)	Inhalt der Prozessorregister
Aufrufliste	(Alt)(7)	Anzeige der Funktionen, die zum Aufruf dieser Funktion geführt haben
Überwachung	(Alt)(3)	Eine Menge von Variablen, die frei vorgegeben werden kann.
Ausgabe	(Alt)(2)	Ausgaben der Library-Funktionen <i>OutputDebugString</i> und <i>afxDump</i> sowie von Makros wie z.B. TRACE

Tabelle 9.1: Fenster während des Debuggens

Aufgerufen werden die Fenster entweder über das Menü ANSICHT oder über die in der Spalte »Taste« angegebene Zugriffstaste.

Auch hier handelt es sich wiederum um ganz normale *MDI-Kind*-Fenster des Developer Studios, die nach Belieben auf dem Bildschirm angeordnet werden können. Zur besseren Übersicht können die Fenster aneinandergefügt (angedockt) werden. Eine typische Debug-Session zeigt Abbildung 9.1. Hier wird besonders deutlich, wie vorteilhaft ein großer Monitor sein kann.

Um ein Programm debuggen zu können, muß es speziell übersetzt werden. Mit Visual C++ ist dies aber ganz einfach, denn dazu braucht lediglich unter ERSTELLEN|AKTIVE KONFIGURATION FESTLEGEN der *Debug-Modus* aktiviert zu werden (Standard nach dem Anlegen eines neuen Projekts). Nach einem erneuten Übersetzungslauf, bei dem *alle* Dateien neu erstellt werden müssen (ERSTELLEN|ALLES NEU ERSTELLEN), kann mit der Fehlersuche begonnen werden.

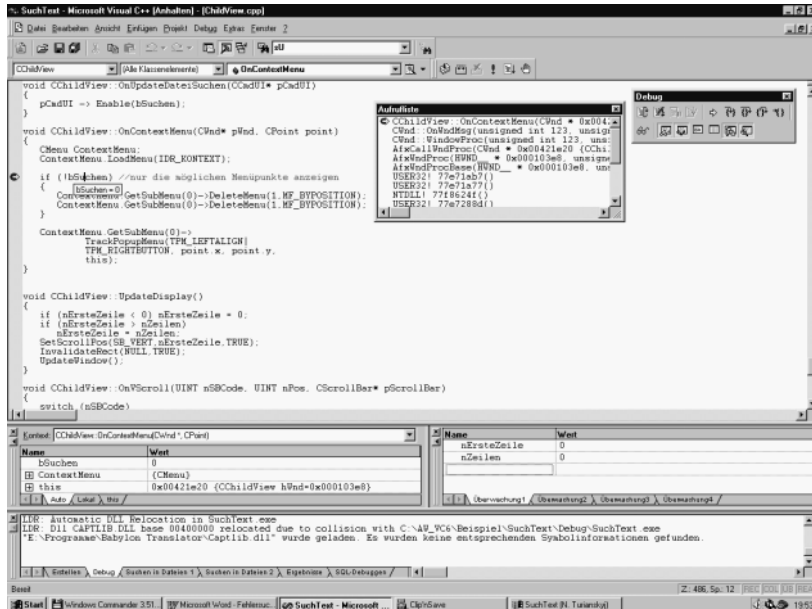


Abbildung 9.1: Eine typische Debug-Session

9.1 Haltepunkte (Breakpoints)

9.1.1 Standard-Haltepunkte

Eine der wichtigsten Eigenschaften eines Debuggers ist die, ein Programm an einer bestimmten Stelle unterbrechen zu können, also einen Haltepunkt (Breakpoint) zu setzen. Das Programm wird dabei normal gestartet und läuft so lange unter der Kontrolle des Debuggers, bis der Haltepunkt erreicht ist. Dann geht es in den Debug-Modus über, und es können Variablen untersucht oder verändert, Registerinhalte überprüft oder das Programm im Einzelschrittmodus fortgesetzt werden.

Die einfachste Form eines Haltepunkts bezieht sich nur auf eine bestimmte Stelle im Quelltext. Ein solcher Haltepunkt kann gesetzt werden, indem der Cursor an der betreffenden Stelle im Quelltext positioniert und dann die HALTEPUNKT-Schaltfläche aus der *Minileiste-Erstellen* (oder **F9**)

gedrückt wird. Auf dieselbe Weise ist es möglich, einen zuvor gesetzten Haltepunkt zurückzunehmen. Ein gesetzter Haltepunkt wird durch einen Punkt vor der Zeile im Quelltext kenntlich gemacht, so daß er auf einen Blick vom normalen Quelltext oder von der aktuellen Programmarke zu unterscheiden ist. Die beim nächsten Schritt auszuführende Programmzeile wird durch einen gelben Pfeil am linken Rand des *Editor*-Fensters markiert.

Ein Haltepunkt markiert immer die Stelle im Quelltext, an der das Programm anhält. Wird ein Programm durch einen Haltepunkt unterbrochen, so wurde das Statement mit dem Haltepunkt selbst noch nicht ausgeführt. Alle aktiven Haltepunkte bleiben zwischen Projektsitzungen erhalten, d.h., sie werden in der Projektdatei gespeichert und stehen beim nächsten Aufruf desselben Projekts wieder zur Verfügung.

Das Setzen dieser einfachen Unterbrechungspunkte reicht für die meisten Anwendungen aus. Allerdings ist zu beachten, daß an einigen Stellen im Quelltext das Einfügen von Haltepunkten den Programmablauf verändert. Dann führt das Debuggen zu falschen Schlüssen. Ein Beispiel dafür sind *SetFocus*- und *KillFocus*-Nachrichten in Dialogen.

9.1.2 Komplexere Breakpoints

Neben den bisher geschilderten positionalen Breakpoints gibt es noch eine Reihe komplexerer Unterbrechungen. Sie unterbrechen ein Programm nur dann, wenn zusätzlich eine Reihe weiterer Bedingungen erfüllt ist; typischerweise ein Ausdruck, der sich geändert oder einen ganz bestimmten Wert angenommen hat. Außerdem ist es möglich, eine Unterbrechung bei Auftreten einer ganz bestimmten Windows-Nachricht zu aktivieren. Diese komplexeren Haltepunkte werden über die Dialogbox *Haltepunkte* editiert (Abbildung 9.2).

Die Dialogbox unterscheidet drei Klassen von Breakpoints:

- ▼ *Pfad*: die genaue Position in einer Quelltextdatei (entspricht Standard-Breakpoint)
- ▼ *Daten*: wenn Daten von Variablen geändert werden oder bestimmte Werte annehmen
- ▼ *Nachrichten*: wenn Windows-Nachrichten eine Unterbrechung hervorrufen sollen

In der unteren Listbox werden alle definierten Haltepunkte angezeigt. Um sie temporär zu deaktivieren, genügt es, das Kontrollkästchen der entsprechenden Zeile auszuschalten. Im Quelltext wird der Haltepunkt dann als weißer Kreis dargestellt. Die Schaltfläche *ENTFERNEN* löscht den aktuellen Haltepunkt.

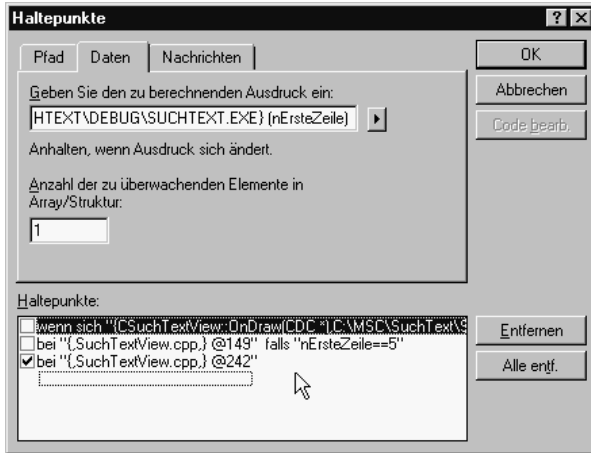


Abbildung 9.2: Definition komplexer Haltepunkte

9.1.3 Haltepunkte im Register »Pfad«

Wie schon beschrieben, entspricht das Setzen eines Haltepunkts unter dem Register *Pfad* dem Setzen eines Standard-Haltepunkts. Allerdings kann hier noch eine zusätzliche Bedingung eingegeben werden, so daß das Programm nur unterbrochen wird, wenn die Bedingung eintrifft. Hier können drei verschiedene Varianten zur Unterbrechung führen:

- ▼ wenn ein Wert geändert wurde
- ▼ wenn eine logische Bedingung erfüllt wurde
- ▼ wenn der Haltepunkt eine bestimmte Zahl an Durchläufen hinter sich hat

Eingegeben wird die Bedingung über die gleichnamige Schaltfläche. Danach wird ein Dialog geöffnet, der zusätzliche Eingaben für die Bedingung erlaubt. Es sind teilweise auch Kombinationen der drei Varianten möglich.

9.1.4 Haltepunkte im Register »Daten«

Das Setzen von bedingten Unterbrechungspunkten erfolgt auf dieser Seite. In die Zeile *Geben Sie den zu berechnenden Ausdruck ein...* wird der Ausdruck eingegeben, der überwacht werden soll. Die Auswertung der Bedingung nimmt der Debugger vor. Es existieren zwei Möglichkeiten, den Ausdruck zu formulieren:

- ▼ wenn ein Wert geändert wurde
- ▼ wenn eine logische Bedingung wahr wird



Abbildung 9.3: Bedingungen für einen Haltepunkt eingeben

Soll nur die Veränderung einer Variablen beobachtet werden, so braucht nur der Name der Variablen eingegeben zu werden. Es ist aber durch Formulierung einer Bedingung auch möglich, genau dann zu unterbrechen, wenn eine Variable einen bestimmten Wert angenommen hat (Abbildung 9.3). Die Ausdrücke müssen einen logischen Wert zurückgeben, also ähnlich einer *if*-Bedingung. Beispiele dafür sind *nColor==2* oder *nErsteZeile>10*. Zusätzlich müssen noch einige Daten im Dialog *Weitere Optionen* eingegeben werden, um genau festzulegen, in welcher Funktion die Bedingung überprüft werden soll (Abbildung 9.4).

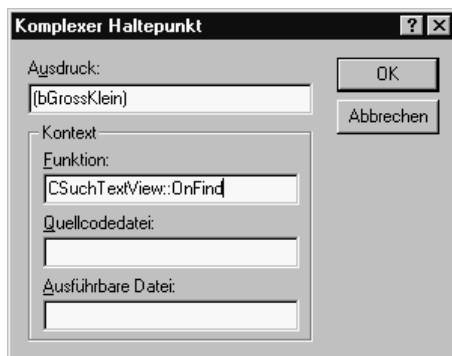


Abbildung 9.4: Eingeben weiterer Optionen für den Haltepunkt



Es ist wichtig zu wissen, daß diese bedingten Haltepunkte zwar sehr vielseitig anwendbar sind, aber die Ausführung des Programms sehr stark verlangsamen. Je mehr diese Haltepunkte vorhanden sind, um so langsamer wird das Programm. Man könnte denken, das Programm hat sich aufgehängt.

9.1.5 Haltepunkte im Register »Nachrichten«

Im Register *Nachrichten* wird festgelegt, daß bei Eintreffen einer bestimmten Windows-Nachricht eine Unterbrechung erfolgen soll. Dabei wird ein Haltepunkt auf den Anfang einer Fensterprozedur gesetzt, und das Programm hält genau dann an, wenn dort die angegebene Nachricht eintrifft. Um diesen Haltepunkt bei der MFC-Programmierung zu verwenden, muß unter *Anhalten bei WndProc* entweder *AfxWndProc* oder *AfxDlgProc* angegeben werden – das sind die Fensterfunktionen für *CWnd*-Fenster und Dialogboxen. Im Kombinationsfeld *...zu überwachende Nachricht* wird dann die gewünschte Nachricht ausgewählt.

Tritt eine der ausgewählten Nachrichten auf, so stoppt das Programm genau am Anfang der entsprechenden Callback-Funktion, also mitten in den Foundation Classes. Wenn die MFC-Quellen installiert sind, kann nun genau so fortgefahren werden, wie es auch bei den eigenen Programmen möglich wäre. Oft genügt es zu wissen, wie die Parameter *lParam* und *wParam* belegt sind oder wie oft eine solche Nachricht geschickt wird.

Falls Sie versuchen, *WM_PAINT* zu untersuchen, werden Sie aus einer Endlosschleife nicht herauskommen. Immer dann, wenn Sie vom Debugger zurück zum Programm schalten, wird *WM_PAINT* geschickt, um das Fenster des Programms neu zu zeichnen. Bevor das jedoch erfolgen kann, ist aber der Debugger vom Developer Studio schon wieder zur Stelle. Dieses Verhalten kann auch für andere Nachrichten zutreffen!



9.2 Programmausführung

Nachdem ein Programm in der Debug-Version übersetzt und gegebenenfalls mit Haltepunkten ausgestattet wurde, kann es gestartet werden. Am einfachsten geht dies über die *AUSFÜHREN*-Schaltfläche in der *Minileiste erstellen*. Alternativ kann aber auch **F5** gedrückt oder der Menüpunkt *ERSTELLEN|DEBUG STARTEN|AUSFÜHREN* ausgewählt werden. In allen Fällen läuft das Programm bis zum nächsten Haltepunkt oder bis es beendet ist.



Abbildung 9.5: Die Debug-Symbolleiste

Man kann auch über die Schaltfläche *AUSFÜHRUNG ANHALTEN* in der *Debug-Symbolleiste* das Programm unterbrechen, jedoch ist der Haltepunkt dann irgendwo in der MFC oder in einem Teil von Windows. Die *Debug-Symbolleiste* (Abbildung 9.5) wird automatisch bei Beginn einer Debug-Session geladen.



Soll nach einem Haltepunkt im Einzelschrittmodus fortgefahren werden, gibt es folgende Möglichkeiten:

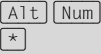
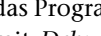
Taste	Button	Funktion
		<i>Zur nächsten Anweisung</i> springen kann dann nützlich sein, wenn man während einer Session in anderen Funktionen oder Dateien etwas sucht und dann zurück will.
		Das Programm läuft bis zur aktuellen <i>Cursorposition</i> und hält dann an.
		<i>In Aufruf springen</i> : Wenn es sich um einen Funktionsaufruf handelt, wird beim ersten Statement in der Funktion angehalten (Step into).
		<i>Aufruf als ein Schritt</i> : Wenn es sich um einen Funktionsaufruf handelt, wird dieser komplett abgearbeitet, und es wird erst unmittelbar danach angehalten.
		<i>Ausführen bis Rücksprung</i> : Das Programm arbeitet die aktuelle Funktion bis zum Ende ab und hält dann an.

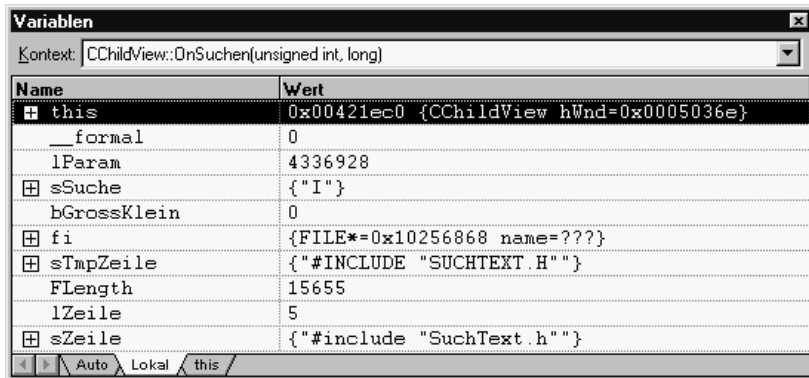
Tabelle 9.2: Einzelschrittvarianten

Alle Möglichkeiten können auch über das Menü **DEBUG** aufgerufen werden, welches während der Debug-Session anstelle des **ERSTELLEN**-Menüs eingeblendet wird. Neben diesen Varianten gibt es noch die Möglichkeit, das Programm mit *Erneut starten* () wieder zu beginnen oder mit *Debug beenden* () die Debug-Session abzubrechen. Diese beiden Optionen haben jedoch ihre Tücken, da sie möglicherweise die anwendungsspezifische Endebehandlung des laufenden Programms nicht korrekt durchführen. Sie sollten daher mit der entsprechenden Vorsicht angewandt werden.

9.3 Programmvariablen

9.3.1 Das Variablen-Fenster

Für die Variablen gibt es ein eigenes Fenster, in dem ihre Werte angezeigt oder verändert werden können (siehe Abbildung 9.6). Dieses Fenster läßt sich über den Menüpunkt **ANSICHT|DEBUG FENSTER|VARIABLEN** aufrufen und kann beliebig auf dem Bildschirm plziert werden. Auch hier kann der Status auf »angedockt« gesetzt werden. Es zeigt alle Variablen an, die an der aktuellen Halteposition entsprechend *Kontext* definiert sind. Im Kombinationsfeld sind alle Funktionen enthalten, die zuvor aufgerufen wurden (entsprechend der Aufrufliste). Das Register *Auto* zeigt Informationen zu den Variablen der vorhergehenden und der aktuell auszuführenden Anweisung an. *Lokal* zeigt alle lokalen Variablen an, und *this* gibt alle Informationen zu der aktuellen Klasse aus. Ist noch keine Debug-Session gestartet oder gibt es keine lokalen Variablen, so ist das Fenster leer.



Name	Wert
☒ this	0x00421ec0 {CChildView hWnd=0x0005036e}
__formal	0
lParam	4336928
☒ sSuche	{"I"}
bGrossKlein	0
☒ fi	{FILE#=0x10256868 name=???}
☒ sTmpZeile	{"#INCLUDE "SUCHTEXT.H"}
FLength	15655
lZeile	5
☒ sZeile	{"#include "SuchText.h"}

Auto Lokal this

Abbildung 9.6: Das Variablen-Fenster

Die angezeigten Variablen enthalten unmittelbar hinter ihrer Adresse möglicherweise ein ☒ oder ☐, um anzuzeigen, daß es sich um zusammengesetzte Werte handelt. Ist das ☒ angegeben, so wird die Variable insgesamt angezeigt, und ihre Struktur ist verborgen. Ist dagegen das ☐ angegeben, so beginnt in der nächsten Zeile die Darstellung der Struktur der Variablen. Zwischen den beiden Darstellungsarten kann durch einen Mausklick auf das Symbol umgeschaltet werden. Die Fähigkeiten des Debuggers, die Struktur von Variablen zu interpretieren, reichen sehr weit und schließen Zeiger, Arrays, Strukturen und Klassen ein.

Neben der Anzeige von Variablen können diese im *Variablen*-Fenster auch verändert werden. Dazu muß für die gewünschte Variable in der *Wert*-Spalte ein Mausklick erfolgen; nun kann dieser nach Belieben editiert werden. Nach dem Drücken der ENTER-Taste bzw. nach dem Entfernen des Cursors wird die Änderung in die Variable übertragen.

9.3.2 Das Fenster Überwachung

Das Fenster *Überwachung* (siehe Abbildung 9.7) arbeitet ähnlich wie das *Variablen*-Fenster: Es kann Variablen anzeigen, Strukturen expandieren und erlaubt es, deren Werte zu ändern. Im Unterschied zum *Variablen*-Fenster hat es aber den Vorteil, daß sein Inhalt frei bestimmt werden kann. Während das *Variablen*-Fenster die gerade gültigen Variablen anzeigt, kann das *Überwachungs*-Fenster auch globale oder statische Variablen anzeigen und sogar C++-Ausdrücke interpretieren.

Um einen Ausdruck einzufügen, muß man das *Überwachungs*-Fenster mit der Maus anwählen, die erste freie Zeile anklicken und den gewünschten Ausdruck eingeben. Nach Drücken von oder wird er ausgewertet, und das Ergebnis wird angezeigt. Falls der Ausdruck nicht ausgewertet werden kann, weil er fehlerhaft ist oder Variablen referenziert, die derzeit

nicht sichtbar sind, wird eine Fehlermeldung ausgegeben. Um einen Ausdruck wieder aus dem *Überwachungs*-Fenster zu entfernen, genügt es, ihn zu markieren und dann mit der Taste `[Entf]` zu löschen. Die zu überwachenden Variablen oder Ausdrücke können auf vier Seiten verteilt werden.



Abbildung 9.7: Das Überwachungs-Fenster



Sie können Variablen oder markierte Teile des Quelltextes auch per Drag & Drop in das Überwachungsfenster schieben.

9.3.3 Das QuickWatch-Fenster

Neben dem *Überwachungs*- und dem *Variablen*-Fenster gibt es noch eine dritte Möglichkeit, Variablen zu untersuchen. Durch Drücken der Tastenkombination `[⇧] [F9]` wird das Fenster *Schnellüberwachung* geöffnet und darin die unter dem Cursor befindliche Variable angezeigt. Alternativ kann auch die entsprechende Schaltfläche  in der Symbolleiste angeklickt werden. Das *Schnellüberwachungs*-Fenster zeigt die Variable an und erlaubt es, sie zu verändern, strukturell aufzugliedern oder in das *Überwachungs*-Fenster zu übertragen (Schaltfläche *ZUR ÜBERWACHUNG HINZUFÜGEN*).

Im Gegensatz zum *Überwachungs*-Fenster bleibt die Schnellüberwachung nicht geöffnet, sondern muß als modale Dialogbox vor dem Weiterarbeiten geschlossen werden. Es dient dazu, mit wenig Aufwand eine Variable zu untersuchen, die nicht dauernd von Interesse ist. Falls es erforderlich ist, diese Variable längere Zeit zu beobachten, sollte sie in das *Überwachungs*-Fenster übertragen werden.

Die *Schnellüberwachungs*-Funktion zum Anzeigen des Variableninhalts wurde auch in das *Editor*-Fenster integriert. Wenn der Cursor mit der Maus über eine Variable gezogen wird, wird während der Debug-Session der In-

halt der Variablen als Tooltip angezeigt. Man kann sich so die Werte von Variablen ausgeben lassen, indem man mit dem Mauszeiger darauf zeigt; einfacher und schneller geht es kaum noch.

9.4 Verschiedenes

9.4.1 Die Aufrufliste

Über den Menüpunkt ANSICHT|DEBUG FENSTER|AUFRUFLISTE kann die Liste der Funktionen angezeigt werden, die zum Aufruf derselben geführt haben. Das Fenster enthält eine Liste aller Funktionsaufrufe: von der Initialisierung bis zur aktuellen Programmstelle. Durch Doppelklick auf eines der Listenelemente kann in den Quelltext der angezeigten Funktion verzweigt werden.

9.4.2 Das Register-Fenster

Neben den bisher erläuterten Fenstern gibt es noch ein weiteres, mit dem die Registerinhalte der CPU angezeigt und modifiziert werden können. Es wird als normales MDI-Fenster über den Menüpunkt ANSICHT|DEBUG FENSTER|REGISTER aktiviert und kann beliebig auf dem Bildschirm positioniert werden. Der Umgang mit diesem Fenster erfordert einige Erfahrung, denn das Ändern von Registerinhalten kann leicht zu unerwünschten Nebenwirkungen führen.

9.4.3 Das Speicher-Fenster

Das *Speicher*-Fenster erlaubt es, einen Auszug aus dem Hauptspeicher ab einer bestimmten Adresse zu betrachten. Da es aber aufwendig ist, Adressen einzugeben, kann per Drag & Drop der Bezeichner einer Variablen aus den verschiedenen Fenstern (z.B. *Überwachung*, *Variablen* oder *Editor*) in das *Speicher*-Fenster geschoben werden. Daraufhin wird die Adresse der Variablen als Startadresse benutzt.

9.4.4 Assembler-Listings einblenden

In schwierigen Debug-Situationen hilft oft nur noch ein Blick auf den vom Compiler generierten Assembler-Code. Hier bietet der Debugger von Visual C++ über den Menüpunkt ANSICHT|DEBUG FENSTER|DISASSEMBLIERUNG die Möglichkeit, unter jeder Quelltextzeile den generierten Assembler-Code anzuzeigen. Besonders günstig ist es, daß auch die Einzelschrittfunktionen in diesem Modus auf Assembler-Ebene ablaufen, und es so zulassen, die Wirkung jedes einzelnen Maschinenbefehls zu kontrollieren.

9.4.5 Ausnahmen

Im Fenster *Ausnahmen* (DEBUG|AUSNAHMEN) werden alle definierten Ausnahmen aufgeführt. Hier ist es möglich, einzelne Ausnahmen zu entfernen oder zu definieren, wann das Programm angehalten werden soll.

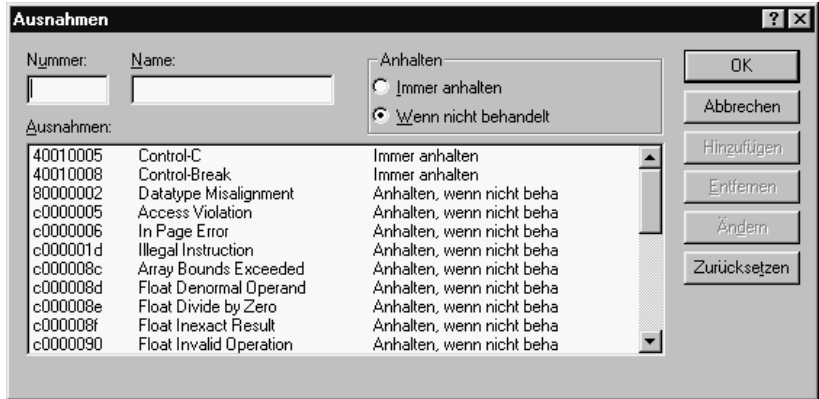


Abbildung 9.8: Das Fenster Ausnahmen

9.4.6 Threads

In diesem Fenster werden alle im Programm aktiven Threads angezeigt. Das Fenster kann über `DEBUG|THREADS` aufgerufen werden. Zusätzlich besteht die Möglichkeit, einzelne Threads anzuhalten und später wieder aufzunehmen.



Abbildung 9.9: Das Threads-Fenster während des Debuggens

9.4.7 Module

Ein weiteres interessantes Fenster kann über `DEBUG|MODULE` aufgerufen werden. Es zeigt alle vom Programm aufgerufenen Module auf, wie zum Beispiel DLLs. Solange diese über das Header-Control nicht sortiert wurden, werden sie in der Reihenfolge angezeigt, in der sie aufgerufen wurden. Zur Fehlersuche ist hier von Interesse, daß der Pfad des aufgerufenen Moduls mitangezeigt wird. Damit läßt sich erkennen, ob nicht doch eine DLL aus einem nicht gewünschten Verzeichnis aufgerufen wurde.

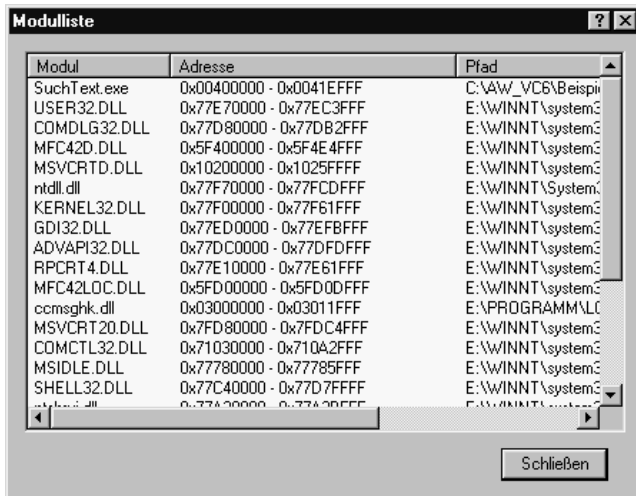


Abbildung 9.10: Die vom Programm aus aufgerufenen Module

9.5 Weitere Debugging-Möglichkeiten

In diesem Abschnitt sollen einige weitere Möglichkeiten angerissen werden, die zur Fehlersuche in Programmen gehören. Sie ermöglichen eine tiefergehende Suche nach Fehlern.

9.5.1 TRACE-Makro

Das TRACE-Makro ermöglicht es, während der Laufzeit des Programms Ausschriften im *Ausgabe-Fenster* (Register *Debug*) vorzunehmen. Dieses Makro wirkt nur in der Debug-Version und muß somit *nicht* für die Release-Version entfernt werden. Die Syntax ähnelt der *printf*-Funktion, d.h., es können auch Inhalte von Variablen ausgegeben werden. Mit diesem Makro läßt sich beispielsweise verfolgen, welche Funktionen wie oft aufgerufen werden, wenn es denn in jeder Member-Funktion aufgerufen wird. Im Quelltext von *SuchText* ist es an einigen Stellen enthalten. Der Aufruf könnte folgendermaßen aussehen:

```
TRACE( "Funktion OnSchriftFarbe: Farbe %d", nFarbe);
```

9.5.2 ASSERT-Makro

Diesem Makro wird ein Ausdruck übergeben, der anschließend ausgewertet wird. Ist das Ergebnis TRUE (oder größer Null), läuft das Programm normal weiter. Ergibt die Berechnung des Ausdrucks FALSE, so wird eine Dialogbox *Assertion failed ...* aufgerufen und das Programm danach beendet. ASSERT wird zum Beispiel verwendet, um zu überprüfen, ob Objekte auf dem Heap korrekt angelegt wurden.

```
CString *sName = new CString;  
ASSERT(sName);
```

Da das Makro nur in die Debug-Version integriert ist und nur dort ausgewertet wird, kann die Überprüfung zur Laufzeit nicht erfolgen. Soll der Ausdruck auch dann getestet werden, so ist das VERIFY-Makro zu benutzen.

9.5.3 Memory Leaks

Memory Leaks sind Fehler, die auftreten, wenn auf dem Heap Speicher allokiert wird (mit dem Operator *new*), aber nicht wieder freigegeben wird (mit *delete*). Erfahrungsgemäß ist die Arbeit mit Zeigern schwierig. Oft ist es schwer zu entscheiden, wann ein Objekt gelöscht werden muß und wann nicht, da einige Objekte automatisch freigegeben werden.

In der Debug-Version wird das Aufspüren dieser Fehler beträchtlich unterstützt. Wenn man zum Beispiel in der Funktion *OnFileOpen* die Zeile `delete mDateiWahl` entfernt, wird man am Ende der Debug-Session im Debug-Ausgabefenster eine Meldung finden. In ihr wird die Zeile genannt, in der das Objekt angelegt wurde, welches nicht wieder freigegeben wurde (Abbildung 9.11).



Abbildung 9.11: Ausgabe-Fenster bei Memory Leaks

Damit sollen die Beschreibungen zur Fehlersuche enden. Es kann kein allgemeingültiges Konzept zur Suche der verschiedensten Fehler in Programmen gegeben werden. Die richtige, effektive Technik muß jeder für sich selbst finden. Erfahrung spielt dabei eine große Rolle. Oft werden Fehlermeldungen ausgegeben, die nur eine Reaktion auf den Fehler sind, aber nicht zur Fehlerquelle selbst führen. An dieser Stelle ist dann die Arbeit des Programmierers gefragt.



Die
Benutzerschnittstellen

TEIL III



Steuerungen

10

Kapitelübersicht

10.1 Buttons (Schaltflächen)	270
10.1.1 Anwendungen	270
10.1.2 Attribute	273
10.1.3 Benachrichtigungscodes	279
10.1.4 Die Klasse CButton	283
10.2 Textfelder	288
10.2.1 Anwendungen	288
10.2.2 Attribute	289
10.2.3 Benachrichtigungscodes	292
10.2.4 Die Klasse CEdit	295
10.2.5 Die Klasse CStatic	303
10.3 Listenfelder und Kombinationsfelder	306
10.3.1 Listenfelder (Listboxen)	306
10.3.2 Kombinationsfelder (Comboboxen)	307
10.3.3 Attribute	309
10.3.4 Benachrichtigungscodes	312
10.3.5 Die Klassen CComboBox und CListBox	316
10.4 Bildlaufleiste/Schieberegler	323
10.4.1 Anwendungen	323
10.4.2 Attribute	324
10.4.3 Benachrichtigungscodes	324
10.4.4 Die Klasse CScrollBar	326
10.5 Regler (Slider)	328
10.5.1 Attribute	328
10.5.2 Benachrichtigungscodes	329
10.5.3 Die Klasse CSliderCtrl	329
10.6.1 Attribute	331
10.6.2 Benachrichtigungscodes	332

10.6.3	Die Struktur TV_INSERTSTRUCT	332
10.6.4	Anlegen einer Struktur	334
10.6.5	Die Klasse CTreeCtrl	336
10.6	Strukturansicht (Tree Control)	331
10.7	Drehfeld (Spin Button Control)	337
10.7.1	Attribute	337
10.7.2	Benachrichtigungscodes	338
10.7.3	Die Klasse CSpinButtonCtrl	339
10.8	Zugriffstaste (Hotkey Control)	339
10.8.1	Attribute	340
10.8.2	Benachrichtigungscodes	340
10.8.3	Die Klasse CHotKeyCtrl	340
10.9	Statusanzeige (Progress Control)	341
10.10	Eigenschaftfenster (Property Sheets)	342
10.10.1	Anlegen der Dialog-Ressourcen	342
10.10.2	Anlegen der Property Pages	343
10.10.3	Anlegen des Property Sheets	344
10.11	Registerkarte (Tab Control)	346
10.11.1	Attribute	346
10.11.2	Benachrichtigungscodes	347
10.11.3	Die Struktur TC_ITEM	348
10.11.4	Die Klasse CTabCtrl	349
10.11.5	Beispielprogramm	351
10.12	Listenelement (List Control)	354
10.12.1	Anwendungen	354
10.12.2	Attribute	355
10.12.3	Benachrichtigungscodes	356
10.12.4	Die Klasse CListControl	357
10.12.5	Beispielprogramm	362
10.13	Animation (Animate Control)	366
10.13.1	Anwendung	366
10.13.2	Attribute	367
10.13.3	Die Klasse CAnimateCtrl	367
10.13.4	Beispielprogramm	368
10.14	Datum/Uhrzeit-Steuerung	368
10.14.1	Anwendung	368
10.14.2	Attribute	369
10.14.3	Benachrichtigungscodes	370
10.14.4	Die Klasse CDateTimeCtrl	370
10.14.5	Beispielprogramm	375
10.15	Die Monatskalender-Steuerung	378
10.15.1	Anwendung	378
10.15.2	Attribute	378
10.15.3	Benachrichtigungscodes	379

10.15.4 Die Klasse CMonthCalCtrl	380
10.15.5 Beispielprogramm	384
10.16 Die IP-Adressensteuerung	390
10.16.1 Anwendung	390
10.16.2 Attribute	390
10.16.3 Benachrichtigungscodes	390
10.16.4 Die Klasse CIPAddressCtrl	391
10.16.5 Beispielprogramm	392
10.17 Das erweiterte Kombinationsfeld-Steuerelement	394
10.17.1 Anwendung	394
10.17.2 Attribute	394
10.17.3 Benachrichtigungscodes	395
10.17.4 Die Klasse CComboBoxEx	396
10.17.5 Beispielprogramm	398
10.18 ActiveX-Controls	400
10.18.1 Anwendung	400
10.18.2 Ablauf	400
10.18.3 Attribute	402
10.18.4 Beispielprogramm	402

Dieser Abschnitt erklärt verschiedene Arten der Steuerungen. Zu ihnen gehören Schaltfläche, Eingabefeld, Listbox, Combobox, Schieberegler sowie weitere Steuerungen, die seit Windows 95 bzw. NT 4.0 verfügbar sind. Hier hat Microsoft bei der Übersetzung der Namen ganze Arbeit geleistet. Wußten Sie schon, was ein Optionsfeld ist? Radio-Buttons sind Ihnen aber sicher bekannt. Es fällt uns hier etwas schwer, immer die neuen (deutschen) Begriffe zu benutzen. Nehmen Sie es uns deshalb bitte nicht übel, wenn wir anstelle des deutschen Begriffes ab und zu das englische Äquivalent benutzen.

Jede Steuerung wird in vier Abschnitten beschrieben:

1. Der Abschnitt *Anwendung* erklärt den Einsatz und die Anwendungsmöglichkeiten der Steuerung, gibt Tips für die Verwendung in eigenen Programmen und liefert Beispiele aus Standard-Windows-Programmen, in denen Dialogboxen mit der jeweiligen Steuerung arbeiten.
2. *Attribute* erläutert die verschiedenen Eigenschaften der Steuerung, die ihr Aussehen und Verhalten beeinflussen. Es wird erklärt, wie diese Eigenschaften mit Hilfe des Dialogeditors verändert werden können und welche Auswirkungen das hat.
3. Der Abschnitt *Benachrichtigungscodes* enthält eine Beschreibung der Nachrichten, die von Steuerungen des jeweils beschriebenen Typs an die Dialogbox gesendet werden. Es wird erläutert, wann ein Benachrichtigungscod auftaucht, wie die Anwendung darauf reagieren sollte und mit welchen Message-Map-Einträgen die Verbindung zu den Methoden der Dialogboxklasse hergestellt wird.
4. *Die Klasse ...* liefert eine Beschreibung der Klasse, die speziell für den Zugriff auf diese Art von Steuerung entworfen wurde. Hier wird erläutert, mit welchen Methoden man auf die Steuerung zugreifen kann, und welche Effekte dies hat.

Zusätzlich finden sich am Ende der Abschnitte spezielle Hinweise auf Besonderheiten der jeweiligen Dialogboxelemente oder Bemerkungen über verwandte Steuerungen. Nicht jede Steuerung wird in einem eigenen Abschnitt behandelt. *Optionsfelder*, *Kontrollkästchen* und *Schaltflächen* werden unter dem Oberbegriff *Buttons* zusammengefaßt, *Statischer Text* und *Eingabefelder* werden gemeinsam abgehandelt und *Listboxen*, *Comboboxen* und *Schieberegler* finden Sie ebenfalls gruppiert.

10.1 Buttons (Schaltflächen)

10.1.1 Anwendungen

Buttons finden in ganz unterschiedlichen Anwendungen Verwendung, die man am besten in zwei Gruppen unterteilt: das Auslösen von Ereignissen und das Speichern von Zuständen. Zur ersten Gruppe zählen vor al-

lem die *Schaltflächen*, während *Kontrollkästchen* und *Optionsfelder* der zweiten Gruppe angehören. Allerdings gibt es auch Mischformen, beispielsweise Optionsfelder, die Ereignisse auslösen oder Schaltflächen, die Zustandsspeicher besitzen.

Schaltflächen (Push-Buttons)

Ereignisse werden in der Regel durch Schaltflächen ausgelöst. Drückt der Anwender eine Schaltfläche, so rechnet er mit einer unmittelbaren Reaktion des Programms. In nahezu jeder Dialogbox finden sich Schaltflächen mit den Beschriftungen *FERTIG*, *OK*, *ABBRUCH* oder ähnliches, mit denen der Dialog beendet werden kann. Auch die meisten Fenster enthalten Schaltflächen zu ihrer Minimierung und Maximierung und eine Schaltfläche für das Systemmenü, also Schaltflächen, die eine unmittelbare Reaktion des Programms auslösen.

Dagegen besitzen Schaltflächen keinen inneren Zustandsspeicher, d.h., sie können sich keine Werte merken. Aus diesem Grund ist eine direkte Reaktion des Programms auf das Drücken einer Schaltflächen erforderlich, denn später – etwa beim Schließen der Dialogbox – wäre nicht mehr erkennbar, ob während des Dialogs der Button gedrückt wurde. Die Reaktionsfreudigkeit von Schaltflächen äußert sich in dem zugehörigen MFC-Programm darin, daß in der Regel zu jeder Schaltfläche eine Methode vorhanden ist, die dieses Ereignis pflegt. Die Implementierung der Methode bestimmt, welche Wirkung das Drücken der Schaltfläche hat: Soll nur ein innerer Zustand umgeschaltet werden, soll etwas an der Dialogbox verändert werden, oder soll das Programm selbst eine größere Aktion durchführen?

Kontrollkästchen (Checkboxes)

Einen anderen Einsatzbereich decken Kontrollkästchen und Optionsfelder ab, denn sie werden vorwiegend gebraucht, um unterschiedliche Zustände zu speichern und anzuzeigen. Dabei repräsentiert ein Kontrollkästchen einen Wahrheitswert. Ist es gesetzt, so speichert es den Wert WAHR, ist es nicht gesetzt, repräsentiert es den Wert FALSCH – oder JA und NEIN, 1 und 0, ERLAUBT und VERBOTEN usw. In der Praxis werden Kontrollkästchen eingesetzt, um zwischen zwei unterschiedlichen Zuständen umzuschalten. Winword verwendet ein Kontrollkästchen, beispielsweise im Dialog *BEARBEITEN|ERSETZEN*, um abzufragen, ob bei der Suche Groß- und Kleinschreibung berücksichtigt werden soll; oder in *FORMAT|ZEICHEN*, um Textauszeichnungen wie Fettschrift, Unterstreichen usw. ein- oder auszuschalten. Übrigens kann man sich Menüpunkte, die mit einem Häkchen versehen sind, auch als eine Art Kontrollkästchen vorstellen: Ist das Häkchen gesetzt, wird der Wert WAHR dargestellt, ist es nicht gesetzt, ergibt das den Wert FALSCH.

Kontrollkästchen besitzen einen inneren Zustandsspeicher, d.h., sie wissen, ob sie gerade gesetzt sind oder nicht. Dieser Zustand kann nicht nur

vom Benutzer verändert werden (durch Anklicken), sondern auch vom Programm. Darüber hinaus hat das Programm die Möglichkeit, den aktuellen Zustand des Kontrollkästchens abzufragen. In den meisten Fällen werden Kontrollkästchen am Anfang des Dialogs (also in *InitDialog*) initialisiert, während des Dialogs vom Benutzer verändert und beim Beenden des Dialogs (in *OnOK*) vom Programm abgefragt. Allerdings muß dies nicht zwangsläufig so sein, denn auch Checkboxes lassen sich so programmieren, daß sie sofort auf einen Mausklick reagieren.

Optionsfelder (Radio-Buttons)

Optionsfelder dienen zur Auswahl *eines* Wertes aus einer etwas größeren Menge, typischerweise mit drei, vier, fünf oder wenig mehr Elementen. Der Name Radio-Button (und nicht Optionsfeld) erinnert an die Zeit der Röhrenradios, die zur Umschaltung der verschiedenen Frequenzbänder eine Reihe von Knöpfen hatten, von denen immer nur einer gleichzeitig gedrückt sein konnte. Wurde ein anderer Knopf gedrückt, so sprang der vorige heraus. (Ein nettes Spiel ist es, die Mechanik zu überlisten, um doch mehr als einen Knopf zu arretieren – unter Windows müßte man also nur zwei Mäuse anschließen...)

Optionsfelder werden also immer in Gruppen verwendet, ein einzelner ist nicht mehr als ein Kontrollkästchen. Innerhalb dieser Gruppe ist genau ein Knopf gedrückt, die anderen nicht. Optionsfelder repräsentieren also eine 1-aus-n-Auswahl (Single-Choice). Beispiele für Optionsfelder finden sich im Standard-Windows-Programm zuhauf: Der wissenschaftliche Taschenrechner beispielsweise erlaubt auf diese Weise die Auswahl eines der vier Zahlensysteme, und in Winword wird in dem Menüpunkt FORMAT|TABULATOREN mit Optionsfeldern bestimmt, ob der Tabulator links- oder rechtsbündig, zentriert oder dezimal gesetzt wird.

Ebenso wie Kontrollkästchen besitzen Optionsfelder einen eigenen Zustandsspeicher. Jeder einzelne Button merkt sich, ob er aktiviert ist oder nicht, und verhält sich damit wie ein Kontrollkästchen. Der Röhrenradioeffekt kommt dadurch zustande, daß man im Dialogeditor mehrere Buttons zu einer Gruppe zusammenfaßt und ihnen das *Auto*-Attribut gibt. Auf diese Weise sorgt die Standard-Dialogboxfunktion automatisch dafür, daß immer nur ein Button aktiv ist und beim Drücken eines anderen wieder ausgeschaltet wird.

Setzt man diese Fähigkeiten von Windows nicht ein, sondern pflegt eine Gruppe von Optionsfeldern manuell, so könnte man auch eine n-aus-m-Auswahl realisieren (Multiple-Choice), also beispielsweise zulassen, daß immer genau *zwei* Buttons gedrückt sind. Diese Art und Weise, Optionsfelder einzusetzen, ist allerdings sehr ungewöhnlich und sollte daher besser vermieden werden – derselbe Effekt läßt sich auch mit einer Gruppe von Kontrollkästchen realisieren, die entsprechend gepflegt werden.

Optionsfelder werden im Programm meist ähnlich verwendet wie Kontrollkästchen, d.h., sie werden erst initialisiert, dann vom Benutzer verändert und nach Dialogende vom Programm abgefragt. Dennoch ist es möglich, auch Optionsfelder reaktiv zu programmieren, d.h., das Programm sofort antworten zu lassen, wenn der Anwender einen der Knöpfe drückt.

10.1.2 Attribute

Dieser Abschnitt erklärt, mit Hilfe welcher Attribute das Aussehen und Verhalten der verschiedenen Schaltflächen verändert werden kann. Am einfachsten ist es, diese Attribute mit Hilfe des Dialogeditors zu pflegen. Bei einem Klick mit der rechten Maustaste auf eine Steuerung und Wahl des Punktes EIGENSCHAFTEN, wenn man den Menüpunkt ANSICHT|EIGENSCHAFTEN wählt, oder nach Drücken von   erscheint das Eigenschaftsfenster, in dem alle Attribute der markierten Steuerung verändert werden können. Dieses Verfahren entspricht auch der üblichen Vorgehensweise, Dialogboxen nicht manuell, sondern mit dem Dialogeditor zu erzeugen.

Alternativ zur Verwendung des Dialogeditors kann man die Dialogboxschablone auch von Hand editieren und die gewünschten Attribute durch eine ODER-Verknüpfung geeigneter *Style*-Parameter (Attribute) festlegen. Dazu muß man wissen, von welchem Typ die Steuerung ist, welche Syntax dieser Typ hat und welche *Style*-Parameter erforderlich sind (siehe *button styles*, *styles* in der Online-Dokumentation). Neben diesen syntaktischen Einzelheiten liegt der eigentliche Aufwand, eine Dialogboxschablone von Hand zu editieren, darin, daß alle Koordinatenberechnungen von Hand vorgenommen werden müssen. Auch ist es viel schwieriger, die resultierende Dialogbox zu testen, denn dazu muß sie erst übersetzt und mit einem Programm verbunden werden, um dann schließlich mit Hilfe einer geeigneten Dialogboxklasse im Programm aufgerufen werden zu können.

Zur Erklärung der möglichen Arten von Steuerungen und deren Attributen wird in diesem und den folgenden Abschnitten eine dreispaltige Tabelle verwendet. Die linke Spalte gibt den Namen des Attributs an, so wie er auch im Dialogeditor verwendet wird, die rechte Spalte erklärt die Bedeutung des Attributs und seine Auswirkungen auf Aussehen und Verhalten der Steuerung, und die mittlere Spalte gibt an, welches Ressourcen-Statement und welcher *Style*-Parameter beim manuellen Editieren der Dialogboxschablone angewendet werden müssen, um dieselben Effekte zu erzielen. Soll ein bestimmtes Attribut, das Default-Attribut ist (z.B. `WS_VISIBLE`), nicht gegeben werden, so kann es nicht einfach weggelassen werden, sondern ist mit dem Präfix `NOT` zu versehen.

Allgemeine Eigenschaften (Basic Styles)

Es gibt vier Grundeigenschaften, die bei allen Steuerungen möglich sind, und die im Dialogeditor als *Basic Styles* bezeichnet werden.

Editor	Dialogboxschablone	Bedeutung
Sichtbar (Visible)	Ressource-Typ: alle Style: <i>WS_VISIBLE</i>	Die Steuerung ist sichtbar, wird also beim Aufruf der Dialogbox angezeigt. Falls dieses Attribut nicht gesetzt wird, ist die Steuerung während der Anzeige der Dialogbox unsichtbar. (Da <i>WS_VISIBLE</i> ein Default-Style ist, braucht er normalerweise nicht gegeben zu werden. Um ihn zu deaktivieren, ist <i>NOT WS_VISIBLE</i> zu verwenden.)
Deaktiviert (Disabled)	Ressource-Typ: alle Style: <i>WS_DISABLED</i>	Die Steuerung ist nach dem Aufbau der Dialogbox inaktiv, d.h., sie wird zwar angezeigt und kann vom Programm aus verändert werden, nimmt aber keine Benutzereingaben an. Dies kann vom Programm aus mit der Methode <i>CWnd::EnableWindow</i> verändert werden.
Gruppe (Group)	Ressource-Typ: alle Style: <i>WS_GROUP</i>	Gibt das erste Element einer Gruppe von Steuerungen an. Innerhalb einer Gruppe können die einzelnen Steuerungen mit den Cursortasten ausgewählt werden. Eine Gruppe endet immer dort, wo die nächste Gruppe beginnt. Es können also nur hintereinanderliegende Steuerungen zu einer Gruppe zusammengefaßt werden.
Tabstop	Ressource-Typ: alle Style: <i>WS_TABSTOP</i>	Die Steuerung ist durch das Drücken der  -Taste (bzw. rückwärts durch  ) erreichbar. Dieses Attribut sollte in der Regel für alle aktiven und sichtbaren Steuerungen vergeben werden, es sei denn, sie liegen in einer Gruppe und sind nicht deren erstes Element. In diesem Fall bekommt nur das erste Element der Gruppe das Tabstop-Attribut, so daß die Gruppe durch Drücken von  verlassen werden kann.
Hilfe-ID (Help-ID)		Wird dieser Schalter aktiviert, bekommt die Steuerung automatisch eine Hilfe-ID zugeordnet. Diese ID wird aus dem Namen der Steuerung durch Voranstellen des Buchstaben <i>H</i> abgeleitet. Bei Anwendungen mit kontextsensitiver Hilfe kann so zum richtigen Hifetext verzweigt werden.

Tabelle 10.1: Die allgemeinen Styles

In Abbildung 10.1 ist die Dialogbox zur Einstellung der allgemeinen Eigenschaften zu sehen.

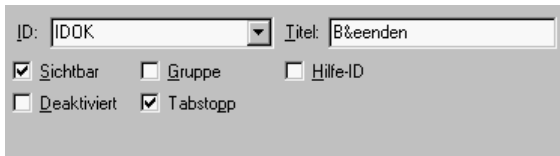


Abbildung 10.1: Die Seite zur Einstellung der allgemeinen Eigenschaften

Erweiterte Eigenschaften (Extended Styles)

Die erweiterten Eigenschaften von Steuerungen (bei Microsoft »Erweiterte Formate« genannt) werden auf einer separaten Dialogseite eingestellt. Sie können ebenfalls bei fast allen Steuerungen eingestellt werden. In Abbildung 10.2 ist die Dialogseite für erweiterte Eigenschaften am Beispiel eines Eingabefeldes dargestellt. Die Tabelle 10.2 enthält eine Zusammenstellung der Attribute und Ihrer Wirkung. Da viele der Attribute Auswirkungen auf das Aussehen der Steuerung haben, kann deren Wirkung gleich im Dialogeditor begutachtet werden. Die für ein Element zugelassenen Attribute werden beim Anklicken mit der Maus im Eigenschaftsfenster aufgeführt.

Das Eigenschaftsfenster wird bei jedem Mausklick geschlossen und muß über das Kontextmenü oder   neu aufgerufen werden. Sie können es aber auch mit dem Push-Pin in der linken oberen Ecke feststecken. Bei einem Mausklick werden dann die Eigenschaften des neu ausgewählten Elements aufgelistet.

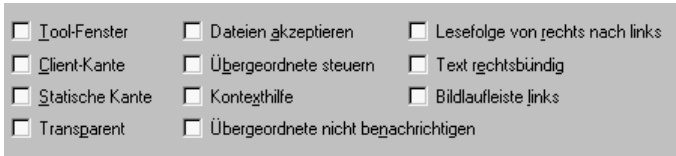


Abbildung 10.2: Die Seite zur Einstellung der erweiterten Eigenschaften

Editor	Dialogboxschablone	Bedeutung
Client Kante (Client Edge)	Style: WS_EX_CLIENTEDGE	Die Umrandung der Steuerung wird mit einer optischen Vertiefung versehen, so daß ein 3D-ähnlicher Anblick entsteht.
Statische Kante (Static Edge)	Style: WS_EX_STATICEDGE	Die Steuerung wird ebenfalls mit einer optischen Vertiefung versehen, die allerdings nicht so stark auffällt wie bei der Client Kante. Sie sollte dazu benutzt werden, um dem Bearbeiter zu zeigen, daß dieses Feld für Änderungen gesperrt ist.
Modaler Rahmen (Modal Frame)	Style: WS_EX_DLGMODALFRAME	Mit einem modalen Rahmen wird eine optische Erhöhung des Elements erreicht. Bei Eingabefeldern führt dieses Attribut zu einem erhöhten Rahmen um das Element.
Transparent	Style: WS_EX_TRANSPARENT	Steuerungen mit diesem Attribut überdecken andere Steuerungen, die eigentlich darunter liegen, nicht.
Dateien akzeptieren (Accept Files)	Style: WS_EX_ACCEPTFILES	Eine Steuerung mit diesem Attribut akzeptiert Drag & Drop-Dateien. Wenn ein Benutzer eine Datei aus dem Explorer in dieses Feld legt, wird die Nachricht WM_DROPFILES an das Element gesendet. Am Mauscursor wird sichtbar, daß dieses Element Dateien per Drag & Drop annimmt.

Tabelle 10.2: Die erweiterten Eigenschaften von Dialogen und Steuerungen

Editor	Dialogboxschablone	Bedeutung
Übergeordnete nicht benachrichtigen(Disabled)	Style: <i>WS_EX_NOPARENTNOTIFY</i>	Mit Hilfe dieses Attributes wird verhindert, daß die Nachricht WM_PARENTNOTIFY gesendet wird, wenn ein Kindfenster erzeugt oder zerstört wird.
Tool-Fenster (Group)	Style: <i>WS_EX_TOOLWINDOW</i>	Ein Tool-Fenster hat eine schmalere Titelleiste, die auch eine kleinere Schriftart enthält. Das Fenster kann dann als verschiebbare Symbolleiste verwendet werden.
Kontexthilfe (Group)	Style: <i>WS_EX_CONTEXTHELP</i>	Das Dialog-Fenster wird über eine Schaltfläche mit einem Fragezeichen in der Titelleiste mit einer kontextsensitiven Hilfe versehen. Wenn der Benutzer die Schaltfläche betätigt, kann er anschließend das Element wählen, zu dem er Hilfe benötigt.
Bildlaufleiste links (Group)	Style: <i>WS_EX_LEFTSCROLLBAR</i>	Eine vertikale Bildlaufleiste wird im Fenster links angeordnet, sofern vorhanden. Standard ist das Attribut WS_EX_RIGHTSCROLLBAR.
Übergeordnete steuern (Group)	Style: <i>WS_EX_CONTROLPARENT</i>	Ermöglicht dem Benutzer das Wechseln der Kindfenster mit der  -Taste.
Text rechtsbündig (Group)	Style: <i>WS_EX_RIGHT</i>	Ordnet Text in einem Element rechtsbündig an. Standard ist das Attribut WS_EX_LEFT.
Lesefolge von rechts nach links(Help-ID)	Style: <i>WS_EX_RTLCREADING</i>	Für einige Sprachen ist es erforderlich, den Text von rechts nach links zu schreiben. Standard ist das Attribut WS_EX_LTRREADING.

Tabelle 10.2: Die erweiterten Eigenschaften von Dialogen und Steuerungen

In Abbildung 10.2 sind nicht alle der in der Tabelle 10.2 enthaltenen Attribute anwählbar. Das liegt daran, daß je nach ausgewählter Steuerung immer nur zugelassene Attribute in das Fenster aufgenommen werden. Für den Dialog selbst gibt es noch die Seite WEITERE FORMATE, die im Kapitel »Dialoge« beschrieben ist.

Spezielle Attribute

Spezielle Attribute der verschiedenen Steuerelemente werden auf der Registerseite FORMATE dargestellt. Diese sind für jede Steuerung unterschiedlich.

Schaltflächen: Bei den speziellen Attributen, die nur für Buttons gültig sind, muß zwischen den einzelnen Arten von Buttons unterschieden werden.

Dialogeditor	Dialogboxschablone	Bedeutung
Standardschaltfläche (Default button)	Ressource-Typ: DEFPUSHBUTTON Style: -	Eine Standardschaltfläche wird mit einem dicken Rahmen versehen. Drückt man in der Dialogbox die  -Taste, so wird ein Druck auf die Standardschaltfläche simuliert, gleichgültig wo sich der Cursor gerade befindet. In einem Dialog ist nur eine Standardschaltfläche zulässig.
Besitzerzeichnung (Owner-Draw)	Ressource-Typ: PUSHBUTTON DEFPUSHBUTTON Style: <i>BS_OWNERDRAW</i>	Ein Button, dessen Aussehen von der Anwendung bestimmt wird. Er sendet zusätzlich zu den normalen Push-Button-Nachrichten immer dann eine Nachricht an die Dialogboxroutine, wenn er gezeichnet, invertiert oder deaktiviert werden muß. Er wird realisiert, indem eine eigene OnDrawItem-Routine geschrieben wird.
Symbol (Icon)	Ressource-Typ: PUSHBUTTON DEFPUSHBUTTON Style: <i>BS_ICON</i>	Auf dem Button wird anstelle von Text ein Symbol aus den Ressourcen dargestellt.
Bitmap	Ressource-Typ: PUSHBUTTON DEFPUSHBUTTON Style: <i>BS_BITMAP</i>	Auf dem Button wird anstelle von Text ein Bitmap aus den Ressourcen dargestellt.
Mehrzeilig (Multiline)	Ressource-Typ: PUSHBUTTON DEFPUSHBUTTON Style: <i>BS_MULTILINE</i>	Falls sehr viel Text auf einer Schaltfläche untergebracht werden muß, kann diese entweder entsprechend lang gemacht oder mit diesem Attribut versehen werden. Der Text wird dann automatisch in eine neue Zeile umgebrochen, wenn er zu lang wird.
Benachrichtigung (Notify)	Ressource-Typ: PUSHBUTTON DEFPUSHBUTTON Style: <i>BS_NOTIFY</i>	Das übergeordnete Fenster wird benachrichtigt, wenn die Taste gedrückt wurde.
Flach (Flat)	Ressource-Typ: PUSHBUTTON DEFPUSHBUTTON Style: <i>BS_FLAT</i>	Durch dieses Attribut wird eine Schaltfläche zweidimensional dargestellt.
Horizontale/vertikale Ausrichtung (Horizontal/ vertical alignment)	Ressource-Typ: PUSHBUTTON DEFPUSHBUTTON Style: <i>BS_LEFT, BS_RIGHT, BS_CENTER</i> <i>BS_TOP, BS_BOTTOM,</i> <i>BS_VCENTER</i>	In zwei Kombinationsfeldern kann die Ausrichtung des Textes der Schaltfläche horizontal und vertikal bestimmt werden.

Tabelle 10.3: Attribute von Push-Buttons

Kontrollkästchen: Bei Kontrollkästchen können unabhängig voneinander drei Attribute gesetzt werden. Sie betreffen sowohl das Aussehen der Kontrollkästchen als auch ihr Verhalten. Neben den hier angegebenen Attributen besitzt ein Kontrollkästchen einen Text, der entweder links oder rechts neben dem Markierungskästchen angezeigt wird.

Dialogeditor	Dialogboxschablone	Bedeutung
Auto	Ressource-Typ: CONTROL Style: <i>BS_AUTOCHECKBOX</i> (bzw. <i>BS_AUTO3STATE</i> , wenn zusätzlich das Attribut <i>Tristate</i> aktiviert ist)	Das Kontrollkästchen wird automatisch an- bzw. ausgeschaltet, wenn der Benutzer es anklickt. Fehlt dieses Attribut, muß die Anwendung selbst für das An- und Ausschalten des Kontrollkästchens sorgen.
Tri-State	Ressource-Typ: CONTROL Style: <i>BS_3STATE</i> (bzw. <i>BS_AUTO3STATE</i> , wenn zusätzlich das Attribut <i>Auto</i> aktiviert ist)	Erzeugt eine Checkbox mit drei statt zwei Zuständen. Der dritte Zustand zeigt an, daß die Checkbox inaktiv ist und keinen Wahrheitswert repräsentiert.
Text links	Ressource-Typ: CONTROL Style: <i>BS_LEFTTEXT</i>	Der Text der Checkbox erscheint auf der linken Seite. Wird dieses Attribut nicht gegeben, so erscheint er auf der rechten Seite. Besitzt die Checkbox weder das Attribut <i>Auto</i> noch <i>Tristate</i> , ist die Ressource vom Typ <i>CHECKBOX</i> .
Drucktaste	Ressource-Typ: CONTROL Style: <i>BS_PUSHLIKE</i>	Macht aus einem Kontrollkästchen eine Taste, die aber entgegengesetzt zur Schaltfläche in gedrücktem Zustand verharren kann. Damit werden die Wahrheitswerte der Steuerung repräsentiert.
Mehrzeilig, Benachrichtigung, Flach, Symbol, Bitmap, Ausrichtung	Ressource-Typ: CONTROL	Diese Attribute wirken genauso wie bei den Schaltflächen beschrieben.

Tabelle 10.4: Attribute von Kontrollkästchen

Optionsfelder: Optionsfelder besitzen ähnliche Attribute wie Kontrollkästchen, können aber keinen dritten Zustand annehmen.

Dialogeditor	Dialogboxschablone	Bedeutung
Auto	Ressource-Typ: CONTROL Style: <i>BS_AUTORADIOBUTTON</i>	Der Radio-Button wird automatisch an- bzw. ausgeschaltet, wenn der Benutzer ihn anklickt. Fehlt dieses Attribut, muß die Anwendung selbst für das An- und Ausschalten sorgen.
Text links	Ressource-Typ: RADIOBUTTON oder CONTROL Style: <i>BS_LEFTTEXT</i>	Der Text des Radio-Buttons erscheint auf der linken Seite. Wird dieses Attribut nicht gegeben, so erscheint er auf der rechten Seite. Besitzt die Steuerung nicht das Attribut <i>Auto</i> , ist die Ressource vom Typ RADIOBUTTON.
Drucktaste	Ressource-Typ: CONTROL Style: <i>BS_PUSHLIKE</i>	Macht aus einem Radio-Button eine Taste, die aber anders als bei der Schaltfläche in gedrücktem Zustand verharren kann. Damit werden die Wahrheitswerte der Steuerung repräsentiert.
Mehrzeilig, Benachrichtigung, Flach, Symbol, Bitmap, Ausrichtung	Ressource-Typ: CONTROL	Diese Attribute wirken genauso wie bei den Schaltflächen beschrieben.

Tabelle 10.5: Attribute von Optionsfeldern

10.1.3 Benachrichtigungscode

Wenn eine Schaltfläche gedrückt wird, erzeugt sie eine WM_COMMAND-Nachricht mit dem Benachrichtigungscode BN_CLICKED in HIWORD(1Param) und dem Bezeichner der Steuerung in wParam.

Drücken Sie die Taste F12, wenn der Cursor auf BN_CLICKED steht, um den Klassenbrowser zu starten und zur Definition des Makros zu springen.

Um auf diese Nachricht zu reagieren, wird ein ON_COMMAND-Eintrag mit der folgenden Syntax in die Message-Map eingefügt:

```
ON_COMMAND(<id>, <memberFxn>)
```

Dieser Eintrag ruft die Methode `memberFxn` genau dann auf, wenn die Taste mit dem Bezeichner `id` gedrückt wurde.

In HEXE gibt es eine ganze Reihe von Schaltflächen, mit denen die Tasten des Rechners dargestellt werden. Wird beispielsweise der Button HILFE gedrückt, führt dies zum Aufruf der Methode *OnHilfe*, denn der Button hat den Bezeichner IDD_HILFE und wird über den Message-Map-Eintrag

```
ON_BN_CLICKED(IDD_HILFE, OnHilfe)
```

mit der Methode verbunden:



```
void CHEXEDlg::OnHilfe()
{
    CDialog dlg(IDD_HILFE, this);
    dlg.DoModal();
}
```

Listing: Reaktion auf Drücken von Hilfe

OnHilfe ruft ihrerseits durch Instanziierung der Methode *CDialog* den Hilfe-Dialog von HEXE auf.

Ähnlich wie schon bei Menü-Nachrichten ist es nicht unbedingt nötig, für jede Nachricht eine eigene Methode vorzusehen. Statt dessen kann man mehrere Nachrichten auf eine einzige Methode umleiten und in dieser mit Hilfe von *CWnd::GetCurrentMessage* die erforderliche Differenzierung vornehmen. Auf diese Weise werden in HEXE beispielsweise alle Zifferntasten auf die Methode *OnNumber* umgeleitet:

```
ON_BN_CLICKED(IDD_BUTTON_0, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_1, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_2, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_3, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_4, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_5, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_6, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_7, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_8, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_9, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_A, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_B, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_C, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_D, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_E, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_F, OnNumber)
...
```

Listing: Ausschnitt aus der Message-Map

Am einfachsten werden die Einträge der Message-Map mit dem Klassen-Assistenten angelegt. Neben dem Eintrag in der Message-Map werden dann auch gleich der Funktionsrumpf sowie die Deklaration der Funktion eingefügt. Der Programmierer kann dabei keine Fehler mehr machen und muß nur noch die Funktion implementieren. Die Voraussetzung zur Benutzung des Klassen-Assistenten ist allerdings, daß die Dialog-Klasse ebenfalls mit dem Klassen-Assistenten angelegt wurde.

Wird dann ein Message-Map-Eintrag erstellt, so wird für den ausgewählten Button keine WM_COMMAND-Nachricht benutzt, sondern der Benachrichtigungscode BN_CLICKED wird ausgewertet und ein ON_BN_CLICKED-Eintrag erstellt.

```
BEGIN_MESSAGE_MAP(CHEXEDlg, CDialog)
    //{{AFX_MSG_MAP(CHEXEDlg)
    ON_BN_CLICKED(IDD_BUTTON_0, OnNumber)
    ...
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Dieser Eintrag gleicht aber von der Übergabe der Parameter her einer ON_COMMAND-Nachricht. In der zugehörigen Funktion `OnNumber` muß dann zuerst untersucht werden, welcher Button gedrückt wurde.

```
void CHEXEDlg::OnNumber()
{
    int num = GetCurrentMessage()->
                    wParam - IDD_BUTTON_0;
    if (num >= nbase)
    {
        MessageBeep(0);
    }
    else
    {
        if (new_number)
        {
            new_number = FALSE;
            areg = num;
        }
        else
        {
            areg = areg * nbase + num;
        }
    }
    UpdateCalculatorDisplay();
}
```

Listing: Reaktion auf Eingabe einer Ziffer

Es wurde bereits erwähnt, daß auch Kontrollkästchen und Optionsfelder reaktiv sein können, d.h., unmittelbar auf einen Mausklick reagieren. In HEXE gibt es beispielsweise eine Gruppe von Optionsfeldern, mit denen zwischen den Zahlensystemen *binär*, *oktal*, *dezimal* und *hexadezimal* umgeschaltet werden kann. Auf das Drücken eines dieser Optionsfelder muß

das Programm natürlich *sofort* reagieren, um den Eingabemodus an das neue Zahlensystem anzupassen – es reicht keinesfalls aus, den Zustand der Buttons erst nach dem Ende der Dialogbox abzufragen.

Da auch Optionsfelder und Kontrollkästchen beim Drücken eine WM_COMMAND-Nachricht mit ihrem Bezeichner senden, ist es sehr einfach, das Programm sofort darauf reagieren zu lassen; es müssen nur eine Methode und ein dazu passender Message-Map-Eintrag angelegt werden. Die vier Radio-Buttons haben in HEXE die Bezeichner IDD_INPUT_16, IDD_INPUT_10, IDD_INPUT_8 und IDD_INPUT_2 und werden gemeinsam auf die Methode `OnBase` umgeleitet:

```
ON_BN_CLICKED(IDD_INPUT_2, OnBase)
ON_BN_CLICKED(IDD_INPUT_8, OnBase)
ON_BN_CLICKED(IDD_INPUT_10, OnBase)
ON_BN_CLICKED(IDD_INPUT_16, OnBase)
```

In dieser wird dann mit `GetCurrentMessage` überprüft, welcher der Radio-Buttons aktiv ist, und die Variable `nbase` wird auf den neuen Wert gesetzt:

```
void CCalculatorDlg::OnBase()
{
    switch (GetCurrentMessage()->wParam)
    {
        case IDD_INPUT_16 : nbase = 16; break;
        case IDD_INPUT_10 : nbase = 10; break;
        case IDD_INPUT_8  : nbase = 8; break;
        case IDD_INPUT_2  : nbase = 2; break;
    }
    new_number = TRUE;
}
```

Listing: Funktion zum Umschalten der Zahlensysteme

Da die Optionsfelder mit dem Attribut *Auto* definiert wurden, braucht sich `OnBase` nicht mehr darum zu kümmern, den gedrückten Button auch visuell anzuschalten; dies wird ebenso automatisch von der Standard-Dialogboxfunktion erledigt wie das Ausschalten des vorher aktivierten Optionsfeldes.

Die wichtigsten Schaltflächen einer Dialogbox – der OK- und der ABBRUCH-Button – brauchen nicht über Message-Map-Einträge mit einer Methode verbunden zu werden. Sie sind bereits in der Basisklasse *CDialog* an die virtuellen Methoden `OnOK` und `OnCancel` gekoppelt und brauchen bei Bedarf in der abgeleiteten Klasse nur noch überlagert zu werden. Obwohl es den Anschein hat, die Beziehung zwischen den Buttons und den Methoden würde über den Namen des Buttons hergestellt, ist dies nicht der

Fall. Vielmehr ist für den Aufruf der richtigen Methode ausschlaggebend, daß die Buttons den Bezeichner 1 (für *OnOK*) oder 2 (für *OnCancel*) haben. Weil diese Bezeichner so oft gebraucht werden, wurden sie in *WINDOWS.H* als symbolische Konstanten *IDOK* und *IDCANCEL* vordefiniert. Um die Tastaturbearbeitung der Dialogbox korrekt durchzuführen, sollten auch tatsächlich der Bezeichner *IDOK* für den OK- und *IDCANCEL* für den ABBRUCH-Button verwendet werden. In diesem Fall führt das Drücken der -Taste zum Aufruf von *OnOK* und das Drücken der -Taste zum Aufruf von *OnCancel*.

10.1.4 Die Klasse *CButton*

Überblick

Die Klasse *CButton* ist vorgesehen, um zur Laufzeit Dialogboxelemente vom Typ *Schaltfläche*, *Kontrollkästchen* oder *Optionsfeld* zu verändern. Sie besitzt Methoden, mit denen die Eigenschaften und das Aussehen eines Buttons zur Laufzeit verändert werden können. Um auf einen Button zuzugreifen zu können, der mit dem Dialogeditor erzeugt wurde (also über eine Ressource-Datei eingebunden wurde), kann die Methode *CWnd::GetDlgItem* aufgerufen werden; sie liefert einen Zeiger auf ein temporäres *CButton*-Objekt.

Alle Aufrufe von Methoden dieses Objekts wirken sich dann auf die konkreten Eigenschaften der angesprochenen Steuerung aus. Da der Rückgabewert von *GetDlgItem* »formal« ein Zeiger auf ein *CWnd*-Objekt ist, können zunächst nur Methoden der Klasse *CWnd* aufgerufen werden. Um die speziellen Methoden von *CButton* aufrufen zu können, muß der Rückgabewert in einen *CButton*-Zeiger umgewandelt werden, wie beispielsweise im Rechnerdialog von *HEXE* in *OnInitDialog*:

```
BOOL CCalculatorDlg::OnInitDialog()
{
    ...
    switch (nbase) {
        case 16 : ((CButton*)GetDlgItem (IDD_INPUT_16))->SetCheck(1);
                break;
        case 10 : ((CButton*)GetDlgItem (IDD_INPUT_10))->SetCheck(1);
                break;
        ...
    }
}
```

Listing: Anwendung von *GetDlgItem*

GetDlgItem liefert einen Zeiger auf ein temporäres *CWnd*-Objekt, dieser wird in einen Zeiger auf ein *CButton*-Objekt konvertiert und der wiederum wird verwendet, um die Methoden der Klasse *CButton* auf die ausgewählte

Steuerung anzuwenden. Da der Rückgabewert auf ein *temporäres* Objekt zeigt, bedeutet dies, daß seine Lebensdauer begrenzt ist. Beim Lesen der nächsten Nachricht, also in diesem Beispiel nach Ende der Methode *OnInitDialog*, führen die Foundation Classes eine *Garbage Collection* durch, d.h., die temporären Objekte werden gelöscht. Es macht also keinen Sinn, den Zeiger längere Zeit, etwa in einer globalen Variablen, zu speichern.

Alternativ dazu können mit dem ClassWizard Member-Variablen für die Radio-Buttons angelegt werden. Die Definition wird in der zugehörigen Header-Datei der Dialogboxklasse eingetragen. Die Variable ist dann vom Typ *CButton*.

```
class CHEXEDlg : public CDialog
{
// Construction
public:
    CHEXEDlg(CWnd* pParent = NULL);    // standard constructor
// Dialog Data
    #ifdef AFX_DATA
    enum { IDD = IDD_HEXE_DIALOG };
    CButton m_Input_16;
    ...
    #endif
};
```

Listing: Anlegen von Member-Variablen für die Radio-Buttons

Die Verbindung zwischen Member-Variablen und Steuerung wird über die Methode *DoDataExchange* hergestellt. In der MFC wird *Dialog Data Exchange* (DDX) dazu benutzt, um Daten zwischen den Steuerungen und den Variablen der Dialogbox zu transferieren. Diese Methode wird automatisch aufgerufen. Sie darf *nie* direkt aufgerufen werden. In einer von *CDialog* abgeleiteten Klasse muß sie überlagert werden. Das sieht auf den ersten Blick alles sehr kompliziert aus. Durch die Verwendung des Klassen-Assistenten wird dem Benutzer aber die gesamte Arbeit abgenommen. Er legt bei Bedarf die Methode an und nimmt auch die entsprechenden Eintragungen für die einzelnen Steuerungen vor. Genauso verhält es sich mit dem zweiten Mechanismus, *Dialog Data Validation* (DDV). Er sorgt dafür, daß Daten in einem bestimmten Wertebereich liegen.

```
void CHEXEDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    #ifdef AFX_DATA_MAP
    DDX_Control(pDX, IDD_INPUT_16, m_Input_16);
    #endif
}
```

Die Methode *OnInitDialog* würde sich unter Verwendung des DDX-Mechanismusses wie folgt verändern:

```
BOOL CCalculatorDlg::OnInitDialog()
{
...
    switch (nbase) {
        case 16 : m_Input_16.SetCheck(1); break;
        case 10 : m_Input_10.SetCheck(1); break;
        ...
    }
}
```

Listing: Datenaustausch (DDX) von Steuerungen mit der Anwendung

Beschriftung

Die Beschriftung eines Buttons kann zur Laufzeit geändert werden. Obwohl dazu Methoden aus der Klasse *CWnd* erforderlich sind (und nicht aus *CButton*), sollen sie der Vollständigkeit halber an dieser Stelle erläutert werden:

```
class Cwnd:
    void SetWindowText(LPCTSTR lpszString);
    int GetWindowText( LPTSTR lpszStringBuf,
                      int nMaxCount );
    int GetWindowText( CString& rString ) const;
```

Ein Aufruf von *SetWindowText* setzt die Beschriftung des Buttons auf die in *lpszString* übergebene Zeichenkette. Bei Schaltflächen wird dadurch der Text *auf* dem Button verändert, bei Kontrollkästchen und Optionsfeldern derjenige *neben* dem Button. Ist nicht genügend Platz für den übergebenen String, wird er am Rand abgeschnitten.

Die Beschriftung eines Buttons kann mit der Methode *GetWindowText* auch ausgelesen werden. Dazu ist in *lpszStringBuf* ein Zeiger auf ein Character-Array zu übergeben, das mindestens die Länge *nMaxCount* hat. Nach dem Aufruf der Methode steht in *lpszStringBuf* die Beschriftung des Buttons. Ist die tatsächliche Beschriftung länger als *nMaxCount*, wird nur die erlaubte Anzahl an Zeichen kopiert. Der Rückgabewert von *GetWindowText* ist die Anzahl der tatsächlich zurückgegebenen Zeichen.

Dieselbe Aufgabe wie *SetWindowText* und *GetWindowText* erfüllen bei Buttons die Methoden *SetDlgItemText* und *GetDlgItemText*: Sie verändern die Beschriftung oder lesen diese aus.

```
class Cwnd:
    void SetDlgItemText( int nID,
                        LPCTSTR lpzString );
    int GetDlgItemText( int nID, LPCTSTR lpStr,
                      int nMaxCount ) const;
    int GetDlgItemText( int nID, CString&
                      rString ) const;
```

Das Konzept dieser Methoden unterscheidet sich allerdings etwas von dem von *Set-/GetWindowText*. *Set-/GetDlgItemText* werden nicht auf das gewünschte *CButton*-Objekt angewendet, sondern auf die Dialogbox als Ganzes (die ja auch aus *CWnd* abgeleitet wurde), in der die gesuchte Steuerung enthalten ist. Nun muß der Methode natürlich auf irgendeine Weise mitgeteilt werden, welches Dialogboxelement sie verändern soll, und dazu dient der Parameter *nID*. Er muß beim Aufruf der Methode den Bezeichner des Buttons enthalten, dessen Beschriftung geändert bzw. abgefragt werden soll; die übrigen Parameter entsprechen denen von *Set-/GetWindowText*.

Markierungszustand

Optionsfelder und Kontrollkästchen haben eine Markierung, die gesetzt oder nicht gesetzt sein kann. Diese Markierung ist nicht nur auf dem Bildschirm sichtbar, sondern wird auch innerhalb der Buttons gespeichert, also besitzen diese Buttons einen inneren Markierungszustand. Der Zustand kann entweder vom Benutzer durch Anklicken oder programmgesteuert durch Aufruf der Methode *SetCheck* der Klasse *CButton* verändert werden.

```
class Cbutton:
    void SetCheck( int nCheck );
    int GetCheck();
```

Das Verhalten von *SetCheck* wird durch den Parameter *nCheck* gesteuert: Ist er 0, so wird die Markierung entfernt, ist er 1, wird sie gesetzt. Bei 3-State-Checkboxes darf er zusätzlich den Wert 2 annehmen, damit das Kontrollkästchen in den dritten Zustand übergeht. Auch hier muß vor der Anwendung der Methode zunächst mit *GetDlgItem* ein Zeiger auf ein temporäres *CButton*-Objekt geholt werden, um dann die Markierung setzen zu können.

Die Gruppe von Optionsfeldern zur Einstellung des Zahlensystems wird auf diese Weise initialisiert: Nur der Radio-Button, der dem eingestellten Zahlensystem entspricht, bekommt eine Markierung, alle anderen bleiben unmarkiert (dies ist die Voreinstellung nach dem Aufrufen des Dialogs).

Neben dem Setzen des Markierungszustands ist es auch möglich, diesen mit Hilfe der Methode *GetCheck* abzufragen. Dazu muß lediglich *GetCheck* auf das gewünschte *CButton*-Objekt angewendet werden, und der Rückgabewert liefert den aktuellen Markierungszustand des Buttons, nämlich 0, 1 oder 2 entsprechend den Konventionen, die auch bei *SetCheck* verwendet wurden.

Die Methoden *GetCheck* und *SetCheck* sollten nur auf Kontrollkästchen und Optionsfelder, nicht aber auf Schaltflächen angewendet werden. Da Schaltflächen keine Markierung und keinen internen Zustandsspeicher besitzen, ist das Verhalten der Methoden, wenn sie auf diese angewendet werden, nicht definiert.

Weitere Methoden

Neben den bisher besprochenen Methoden der Klasse *CButton* gibt es noch einige andere, die nicht so häufig gebraucht werden, und daher in Kurzform abgehandelt werden sollen.

Mit *SetState* kann die Hervorhebung des Buttons gesetzt (*bHighlight*=TRUE) oder entfernt (*bHighlight*=FALSE) werden. Die Hervorhebung eines Buttons wird zum Beispiel gesetzt, wenn der Anwender den Button anklickt und die linke Maustaste gedrückt hält. Um in diesem Fall das Niederdrücken des Knopfes zu visualisieren, erhalten Optionsfelder und Kontrollkästchen einen fetten Rand, während bei Schaltflächen der 3D-Rahmen invertiert wird. Nach dem Loslassen der Maustaste wird die Hervorhebung wieder rückgängig gemacht.

```
class CButton:

    void SetState( BOOL bHighlight );
    UINT GetState();
    void SetButtonStyle( UINT nStyle,
                        BOOL bRedraw = TRUE );
    UINT GetButtonStyle();
```

Mit Hilfe der Methode *GetState* kann abgefragt werden, ob der Button hervorgehoben oder markiert ist, oder ob er den Eingabefokus hat. Dazu muß der Rückgabewert jeweils mit einer passenden Bitmaske UND-verknüpft werden, um die gewünschten Informationen zu erhalten (siehe *GetState* in der Online-Hilfe).

Interessant sind auch die Methoden *GetButtonStyle* und *SetButtonStyle*. Mit *SetButtonStyle* läßt sich der Typ des Buttons verändern. Aus einer Schaltfläche kann man ein Optionsfeld machen, aus einem Kontrollkästchen eine Schaltfläche usw. Normalerweise sind derart grobe Veränderungen aber höchstens geeignet, den Benutzer zu verwirren. Der eigentliche Nutzen

dieser Methode liegt in *kleinen* Veränderungen, beispielsweise der Umwandlung eines Kontrollkästchens in eine 3-State-Checkbox oder einer Schaltfläche in einen Standard-Schaltfläche. Den aktuellen Typ des Buttons kann man mit *GetButtonStyle* abfragen.

10.2 Textfelder

10.2.1 Anwendungen

Eingabefeld (Edit Control)

Eingabefelder haben eine große historische Bedeutung in der Entwicklung von Programmierschnittstellen. Vor nicht allzulanger Zeit waren sie das einzige Mittel eines dialogorientierten Programms, um mit dem Anwender zu kommunizieren. Die Eingabe einer Zahl zur Auswahl eines Menüpunkts, eines Kommandos zum Starten eines Programms oder einer Zeichenkette zur Speicherung mußte über (im weitesten Sinne) textuelle Eingaben realisiert werden. Statt eines Kontrollkästchens wurde ein Textfeld mit einem »x« beschriftet, statt eines Optionsfeldes eine Zahl eingegeben.

Aufgrund der Vielzahl der verfügbaren Benutzerschnittstellenobjekte unter Windows ist die Bedeutung von Textfeldern stark zurückgegangen, sie werden fast nur noch gebraucht, um Text (im eigentlichen Wortsinn) zu erfassen und zu editieren. Dabei ist der Begriff *Text* im Sinne von *Zeichenkette* zu verstehen, es geht nicht nur um Buchstaben, sondern auch um Zahlen, Punktierungen, Sonderzeichen usw. Ein Eingabefeld ist also ganz allgemein ein Eingabemedium für Zeichenketten.

Sieht man sich Windows-Programme an, so findet man Eingabefelder in fast allen Dialogen. In Winword kann der linke Rand des Textes nicht nur über das Absatzlineal, sondern auch in der Dialogbox FORMAT|ABSATZ als numerischer Wert in einem Eingabefeld eingegeben werden.

Das interessanteste Beispiel eines Eingabefeldes ist aber sicherlich das Programm *Editor*, mit dem Textdateien angezeigt und bearbeitet werden können. Editor ist nicht viel mehr als ein Hauptprogramm mit einem Menü und einem riesigen, mehrzeiligen Eingabefeld. Dieses Eingabefeld stellt dem Anwender eine ganze Reihe von Funktionen zur Verfügung, die es erlauben, auch mehrseitige Texte komfortabel zu editieren. Es gibt vielfältige Möglichkeiten, den Cursor zu bewegen und Text einzugeben oder zu löschen. Textblöcke können markiert, kopiert und verschoben werden, außerdem gibt es eine UNDO-Funktion, einen automatischen Zeilenumbruch, eine Funktion zum Suchen von Textteilen und noch vieles mehr.

Der größte Teil dieser komfortablen Editierfunktionen ist in den Eingabefeldern bereits fest eingebaut, so daß er auch in eigenen Anwendungen zur Verfügung steht. Der Schlüssel zu dieser Funktionalität liegt in den At-

tributen eines Eingabefeldes verborgen. Hier entscheidet sich, ob ein Eingabefeld nur eine einzige Zeile bearbeiten kann, oder ob es sich wie ein ausgewachsener Editor verhalten soll. Darüber hinaus gibt es Eingabefelder für Spezialanwendungen, die beispielsweise die Eingabe direkt in Groß- oder Kleinbuchstaben konvertieren, eine verdeckte Eingabe (z.B. bei einem Paßwort) ermöglichen oder es erlauben, eingegebenen Text automatisch linksbündig, rechtsbündig oder zentriert zu formatieren.

Text (Static Text)

Neben den Eingabefeldern gibt es noch eine andere Gruppe von Steuerungen, die in der Lage sind, Text in einer Dialogbox anzuzeigen: den *statischen Text* oder auch nur *Text*. Im Gegensatz zu den Eingabefeldern ist statischer Text aber nicht reaktiv, er sendet keine Nachrichten an das Programm und kann auch nicht vom Anwender verändert werden. Der enthaltene Text wird in der Regel bereits beim Erstellen der Dialogbox festgelegt, kann aber auch zur Laufzeit verändert werden. Der Hauptanwendungsbereich von statischem Text liegt darin, die übrigen Steuerungen einer Dialogbox zu beschriften. Wegen der Veränderbarkeit ihres Inhalts kann statischer Text aber auch zur programmgesteuerten Ausgabe von Text verwendet werden.

Statischer Text hat bezüglich der Formatierung des auszugebenden Textes recht ausgefeilte Fähigkeiten, die denen der Eingabefelder ähneln. So gibt es neben der Möglichkeit, den Text linksbündig, rechtsbündig oder zentriert anzuzeigen, die Fähigkeit, mehrzeiligen Text darzustellen und einen automatischen Zeilenumbruch durchzuführen. Darüber hinaus interpretiert statischer Text eingebettete Tabulatoren zur spaltengetreuen Darstellung und das &-Zeichen, um den nächsten Buchstaben zu unterstreichen.

Neben statischem Text gibt es noch andere statische Steuerungen, beispielsweise zur Ausgabe eines Rahmens, eines Rechtecks oder zum Anzeigen eines Symbols. Diese können mit dem Dialogeditor leicht in eine Dialogbox integriert werden, haben für das Programm aber keine Bedeutung und dienen lediglich der optischen Verschönerung. Am Ende dieses Abschnitts wird kurz erklärt, welche Effekte man mit statischen Symbolen erzielen kann, und wie man damit umgeht.

10.2.2 Attribute

Basic Styles

Die im vorigen Abschnitt erläuterten Grundeigenschaften *Sichtbar*, *Deaktiviert*, *Gruppe* und *Tabstop* lassen sich auch auf Eingabefelder anwenden. Jedes dieser Grundattribute hat bei einem Eingabefeld genau die gleichen Auswirkungen wie bei einem Button, sorgt also dafür, daß das Eingabefeld sichtbar, aktiv, mit der -Taste erreichbar oder erstes Element einer Gruppe ist.

Editierattribute

Neben den Standardattributen und erweiterten Eigenschaften, die alle Steuerungen besitzen, gibt es eine Vielzahl weiterer Eigenschaften, die das Verhalten des Eingabefeldes beeinflussen. Im Dialogeditor können sie über die Kategorie *Allgemein* und *Erweiterte Formate* des Eigenschaften-Fensters verändert werden.

Die meisten von ihnen können separat ein- oder ausgeschaltet werden, es gibt aber auch einige Abhängigkeiten zwischen den Eigenschaften. Die wichtigsten Attribute können Tabelle 10.6 entnommen werden.

Dialogeditor	Dialogboxschablone	Bedeutung
Rand (Border)	Ressource-Typ: EDITTEXT Style: WS_BORDER	Das Eingabefeld bekommt einen Rahmen. (Da WS_BORDER die Voreinstellung ist, braucht nur für den Fall, daß kein Rahmen gegeben werden soll, NOT WS_BORDER gesetzt zu werden.)
Großbuchstaben (Uppercase)	Ressource-Typ: EDITTEXT Style: ES_UPPERCASE	Beim Editieren werden eingegebene Buchstaben automatisch in Großschrift umgewandelt (nicht zusammen mit Kleinbuchstaben verwendbar).
Kleinbuchstaben (Lowercase)	Ressource-Typ: EDITTEXT Style: ES_LOWERCASE	Beim Editieren werden eingegebene Buchstaben automatisch in Kleinschrift umgewandelt (nicht zusammen mit Großbuchstaben verwendbar).
Kennwort (Password)	Ressource-Typ: EDITTEXT Style: ES_PASSWORD	Die Eingabe erfolgt verdeckt, indem jedes eingegebene Zeichen nur als »*« angezeigt wird.
Schreibgeschützt (Read Only)	Ressource-Typ: EDITTEXT Style: ES_READONLY	Erlaubt nur lesenden Zugriff auf das Feld, d.h., die Operationen <i>Anzeigen</i> , <i>Blättern</i> und <i>Kopieren</i> in die Zwischenablage.
Mehrzeilig (Multi-Line)	Ressource-Typ: EDITTEXT Style: ES_MULTILINE	Erzeugt ein mehrzeiliges Eingabefeld, dessen Funktionalität in etwa dem des Windows-Notizblocks entspricht.
Vertikaler Bildlauf (Vert. Scroll)	Ressource-Typ: EDITTEXT Style: WS_VSCROLL	Versieht das Eingabefeld mit einem vertikalen Schieberegler (nur bei mehrzeiligen Eingabefeldern). Dieser Schieberegler ist übrigens für die Anwendung weitgehend transparent, er wird von der Fensterklasse für Eingabefelder selbst bedient.
Horizontaler Bildlauf (Horiz. Scroll)	Ressource-Typ: EDITTEXT Style: WS_HSCROLL	Versieht das Eingabefeld mit einem horizontalen Schieberegler (nur bei mehrzeiligen Eingabefeldern). Auch dieser Schieberegler ist für die Anwendung transparent.

Dialogeditor	Dialogboxschablone	Bedeutung
Auto Hor. Bildlauf (Auto HScroll)	Ressource-Typ: EDITTEXT Style: ES_AUTOHSCROLL	Sorgt dafür, daß der Text beim Editieren am rechten Rand automatisch nach links gescrollt wird.
Auto Vert. Bildlauf (AutoVScroll)	Ressource-Typ: EDITTEXT Style: ES_AUTOVSCROLL	Sorgt dafür, daß der Text beim Anhängen einer neuen Zeile am unteren Rand automatisch nach oben gescrollt wird (nur bei mehrzeiligen Eingabefeldern).
Kein Ausblenden (No hide selection)	Ressource-Typ: EDITTEXT Style: ES_NOHIDSEL	Mit diesem Attribut wird dafür gesorgt, daß eine vorhandene Markierung auch beim Verlust des Eingabefokusses erhalten bleibt.
Numerisch (Number)	Ressource-Typ: EDITTEXT Style: ES_NUMBER	Mit diesem Style können nur numerische Werte in das Eingabefeld eingegeben werden.
Return möglich (Want Return)	Ressource-Typ: EDITTEXT Style: ES_WANTRETURN	Bei Drücken der  -Taste wird eine Zeilenschaltung eingefügt und nicht die Standard-Schaltfläche der Dialogbox ausgelöst. (Gilt nur für mehrzeilige Eingabefelder.)

Tabelle 10.6: Attribute eines Eingabefeldes

Ausrichtung

Zusätzlich kann bei einem mehrzeiligen Eingabefeld festgelegt werden, welche Ausrichtung der Text haben soll. Diese Attribute schließen sich gegenseitig aus, es kann also immer nur eines davon gesetzt sein.

Dialogeditor	Dialogboxschablone	Bedeutung
Linksbündig (Left)	Ressource-Typ: EDITTEXT Style: --	Der Text wird linksbündig ausgerichtet. Da dies die Voreinstellung ist, braucht kein separater Style angegeben zu werden.
Zentriert (Center)	Ressource-Typ: EDITTEXT Style: ES_CENTER	Der Text wird zentriert. Dieses Attribut steht nur in mehrzeiligen Eingabefeldern zur Verfügung.
Rechtsbündig (Right)	Ressource-Typ: EDITTEXT Style: ES_RIGHT	Der Text wird rechtsbündig ausgerichtet. Auch dieses Attribut steht nur in mehrzeiligen Eingabefeldern zur Verfügung.

Tabelle 10.7: Ausrichtungsattribute eines Eingabefeldes

10.2.3 Benachrichtigungscodes

Ein Eingabefeld sendet eine ganze Reihe von Benachrichtigungscodes, über die es der Dialogbox mitteilt, daß Veränderungen stattgefunden haben. Diese Nachrichten können mit Hilfe von Message-Map-Einträgen auf Methoden der Dialogboxklasse umgeleitet werden. Zusammen mit der Möglichkeit, das Verhalten eines Eingabefeldes mit Hilfe der Klasse *CEdit* zur Laufzeit zu manipulieren, ergeben sich damit sehr weitreichende Konfigurationsmöglichkeiten.

Windows sendet die Benachrichtigungscodes eines Eingabefeldes im HI-WORD des Parameters *lParam* einer WM_COMMAND-Nachricht. Bei der SDK-Programmierung ist man als Programmierer selbst dafür verantwortlich, die WM_COMMAND-Nachrichten zu zerlegen, den jeweiligen Absender zu identifizieren und die einzelnen Benachrichtigungscodes herauszufiltern. Mit den Microsoft Foundation Classes ist dies viel einfacher: Hier steht für jeden Benachrichtigungscode ein eigener Message-Map-Eintrag bereit, der diese Arbeit erledigt und die gewünschte Nachricht direkt auf eine Methode umleitet.

Soll die Dialogboxklasse beispielsweise auf den Benachrichtigungscode EN_SETFOCUS des Eingabefeldes mit dem Bezeichner IDD_EDIT reagieren, muß dazu folgendes ausgeführt werden:

1. In der Dialogboxklasse wird eine neue Methode deklariert (z.B. mit dem Namen *OnSetFocus*), die aufgerufen werden soll, wenn die Steuerung IDD_EDIT den Benachrichtigungscode EN_SETFOCUS sendet.
2. In der Message-Map der Dialogboxklasse wird ein neuer Eintrag ON_EN_SETFOCUS(IDD_EDIT, OnSetFocus) eingefügt.
3. Die Methode *OnSetFocus* wird implementiert.

Tabelle 10.8 gibt eine Übersicht von möglichen Benachrichtigungscodes und der zugehörigen Message-Map-Einträge. In den nachfolgenden Abschnitten werden sie detailliert erklärt.

Benachrichtigungscode	Message-Map-Eintrag
EN_SETFOCUS	ON_EN_SETFOCUS (<id>, <memberFxn>)
EN_KILLFOCUS	ON_EN_KILLFOCUS (<id>, <memberFxn>)
EN_CHANGE	ON_EN_CHANGE (<id>, <memberFxn>)
EN_UPDATE	ON_EN_UPDATE (<id>, <memberFxn>)
EN_HSCROLL	ON_EN_HSCROLL (<id>, <memberFxn>)

Benachrichtigungscode	Message-Map-Eintrag
EN_VSCROLL	ON_EN_VSCROLL (<id>, <memberFxn>)
EN_ERRSPACE	ON_EN_ERRSPACE (<id>, <memberFxn>)
EN_MAXTEXT	ON_EN_MAXTEXT (<id>, <memberFxn>)

Tabelle 10.8: Benachrichtigungscode eines Eingabefeldes

Auch hier ist es so, daß bei Verwendung des Klassen-Assistenten zum Anlegen von Message-Map-Einträgen dieser Eintrag entsprechend der Nachricht automatisch erzeugt wird.

Eingabefokus

Bekommt ein Eingabefeld den Eingabefokus, so teilt es dies seiner zugehörigen Dialogbox über den Benachrichtigungscode EN_SETFOCUS mit. Der Eingabefokus wird immer dann erteilt, wenn der Anwender das Eingabefeld mit der Maus oder über die Tastatur anwählt. Verliert ein Eingabefeld den Eingabefokus, d.h., wählt der Benutzer eine andere Steuerung oder schließt die Dialogbox, so wird der Dialogboxklasse der Benachrichtigungscode EN_KILLFOCUS zugesendet.

Änderungen

Wenn der Benutzer innerhalb des Eingabefeldes eine Aktion durchführt, die den Text verändert haben könnte, sendet das Eingabefeld die Benachrichtigungscode EN_CHANGE und EN_UPDATE. Der Unterschied zwischen den beiden besteht darin, daß EN_UPDATE gesendet wird, *bevor* die Änderung auf dem Bildschirm sichtbar ist, während die Meldung EN_CHANGE erst *danach* erfolgt. EN_UPDATE wird aber erst gesendet, wenn der Text intern vom Eingabefeld formatiert wurde. Damit ist es möglich, die Größe des Eingabefeldes zu verändern, bevor der Text tatsächlich angezeigt wird.

Beide Benachrichtigungscode werden gesendet, wenn das Eingabefeld der Meinung ist, daß sich sein Inhalt geändert haben könnte. Dies ist beispielsweise dann der Fall, wenn der Anwender ein neues Zeichen eingegeben oder gelöscht hat, die Markierung überschrieben oder der Inhalt der Zwischenablage eingefügt wurde. Ruft der Anwender die UNDO-Funktion auf ( ), wird jeder Benachrichtigungscode sogar zweimal gegeben.

Ist beispielsweise ein einzelnes Zeichen markiert, und wird dieses mit *demselben* Zeichen überschrieben, sendet das Eingabefeld ebenfalls zweimal die Benachrichtigungscode EN_UPDATE und EN_CHANGE – und das, obwohl sich faktisch gar nichts verändert hat. Der Grund dafür ist, daß sich etwas verändert haben *könnte*, denn die Aktion bestand aus den zwei Schritten *Löschen der Markierung* und *Einfügen eines neuen Zeichens*, die

beide für sich jeweils eine Änderung vorgenommen haben. Eine Anwendung geht beim Empfang eines dieser Benachrichtigungscodes also immer davon aus, daß sich tatsächlich etwas am Text verändert hat.

Eine typische Anwendung von `EN_UPDATE` und `EN_CHANGE` wäre es, eine Markierung (Flag) zu setzen, die festhält, daß der Text geändert wurde. Dies ist insbesondere bei längeren Texten sinnvoll, um zu entscheiden, ob die Daten gespeichert werden sollen oder nicht.

Der Dialog von `HEXE` besitzt ebenfalls ein Eingabefeld, nämlich die Ergebnisanzeige im Dezimalformat. Dieses Feld dient jedoch nicht nur zur Anzeige des Ergebnisses, sondern erlaubt auch die Eingabe von Zahlen. Dazu überwacht die Dialogboxklasse mögliche Eingaben durch Abfangen des `EN_CHANGE`-Benachrichtigungscodes und leitet sie auf die Methode *OnEditChanged* um:

```
void CHEXEDlg::OnEditChanged()
{
    char buf[10+1];
    GetDlgItemText(IDD_OUTPUT_10,buf,10);
    areg = atol(buf);
    new_number = TRUE;
    UpdateCalculatorDisplay(FALSE);
}

...
BEGIN_MESSAGE_MAP(CCalculatorDlg,CDialog)
...
    ON_EN_CHANGE(IDD_OUTPUT_10,OnEditChanged)
END_MESSAGE_MAP()
```

Listing: Nachrichtenbehandlung bei Änderung eines Eingabefeldes

Immer, wenn der Benutzer im Eingabefeld eine Änderung vornimmt, erfährt dieses die Methode *OnEditChanged*. Sie liest daraufhin den Inhalt des Feldes aus, wandelt ihn in ein *long* um und speichert das Ergebnis in der Variablen `areg`. Auf diese Weise kann der Benutzer Zahlen nicht nur über die Push-Buttons eingeben, sondern (wenigstens in dezimaler Form) auch direkt über die Tastatur.

Scrollen

Besitzt das Eingabefeld vertikale oder horizontale Schieberegler, so kann der Benutzer damit auf einfache Weise durch den angezeigten Text scrollen. Um diesen Vorgang auch dem Programm bekanntzumachen, gibt es die Benachrichtigungscodes `EN_HSCROLL` und `EN_VSCROLL`. Damit wird der Dialogbox angezeigt, daß der Benutzer den horizontalen oder vertikalen Schieberegler betätigt hat.

Verwendet der Benutzer die Buttons mit den Pfeilen oder klickt er zwischen diesen und dem Schieber, so wird der Benachrichtigungscode unmittelbar nach dem Mausklick übertragen. Manipuliert er den Schieber direkt, erfolgt die Benachrichtigung erst nach Abschluß der Positionierung und dem Loslassen des Knopfes. Diese BenachrichtigungsCodes werden ebenfalls übermittelt, wenn der Text durch Eingaben oder Cursorbewegungen des Benutzers automatisch gescrollt wird, und zwar auch dann, wenn das Eingabefeld gar keine Schieberegler besitzt.

Fehler

Während des Editiervorgangs können einige Fehler auftauchen, die das Eingabefeld nicht von alleine beheben kann; es benachrichtigt in diesem Fall die Dialogbox durch Senden eines der BenachrichtigungsCodes `EN_MAXTEXT` oder `EN_ERRSPACE`.

`EN_MAXTEXT` wird übermittelt, wenn der Benutzer durch eine Änderung des Textes die maximal erlaubte Größe überschritten hat, beispielsweise nach der Eingabe eines neuen Zeichens oder nach dem Einfügen von Text aus der Zwischenablage. Die maximal erlaubte Länge des Textes kann vom Programm verändert werden. Dieser Benachrichtigungscode wird auch dann übertragen, wenn beim Editieren in einem Multiline-Eingabefeld der Cursor am rechten oder unteren Rand angekommen ist, und das zugehörige Attribut *Auto Hor. Bildlauf* bzw. *Auto Vert. Bildlauf* nicht vorhanden ist.

Der Benachrichtigungscode `EN_ERRSPACE` wird gesendet, wenn das Textfeld eine Aktion aus Speichermangel nicht mehr ausführen kann.

10.2.4 Die Klasse *CEdit*

Die Klasse *CEdit* erlaubt es, programmgesteuert auf ein Eingabefeld zuzugreifen. Sie stellt Methoden zur Verfügung, mit denen der Inhalt des Eingabefeldes abgefragt oder modifiziert, Informationen über den Text beschafft oder Eigenschaften des Eingabefeldes verändert werden können. Darüber hinaus liefert sie eine Schnittstelle zur Zwischenablage.

Um in einer Dialogbox, die nicht vom Programm, sondern durch eine Dialogboxschablone erzeugt wurde, auf ein Eingabefeld zuzugreifen, wird die Methode *GetDlgItem* verwendet. Analog zur Vorgehensweise bei Schaltflächen liefert sie einen Zeiger auf ein temporäres *CWnd*-Objekt, der in einen Zeiger auf ein *CEdit*-Objekt umgewandelt wird. Dieser kann dann zum Zugriff auf das Eingabefeld und den Aufruf von Methoden der Klasse *CEdit* verwendet werden. Auch hier gibt es die Alternative, über eine Member-Variable die Eigenschaften des Eingabefeldes zu ändern bzw. abzufragen.

CEdit besitzt sehr viele Methoden. Es ist an dieser Stelle nicht möglich, auf jede einzelne all ihrer Parameter-Varianten einzugehen. Statt dessen liefern die nachfolgenden Abschnitte eine Auswahl der Methoden, mit denen ein Einsteiger gut zurechtkommen kann. Weitere Informationen finden sich in der elektronischen Hilfe unter *CEdit*.

Textoperationen

SetWindowText und GetWindowText: Man könnte meinen, die wichtigsten Methoden, die von der Klasse *CEdit* angeboten werden, erledigen das Schreiben und Lesen des Inhalts eines Eingabefeldes. Falsch, genau diese Methoden sucht man in *CEdit* vergeblich! Das Lesen und Schreiben erfolgt nämlich mit den aus *CButton* bekannten Methoden *SetWindowText* und *GetWindowText* der Klasse *CWnd*. Während diese Methoden bei Schaltflächen die Beschriftung ändern, sind sie bei Eingabefeldern dafür verantwortlich, den textuellen Inhalt zu bearbeiten – eine Beschriftung oder eine Titelzeile gibt es bei Eingabefeldern ja nicht. Alternativ dazu könnten – ebenso wie bei Schaltflächen – auch die Methoden *SetDlgItemText* und *GetDlgItemText* verwendet werden.

Da es keine syntaktischen Unterschiede gibt, die einen Hinweis darauf geben, welche der beiden Aufgaben ein Aufruf einer dieser Methoden gerade ausführt, brauchen sowohl das Programm als auch dessen Benutzer Kontextverständnis, d.h., jeder der beiden muß wissen, ob das Ziel der Aktion eine Schaltfläche oder ein Eingabefeld ist. Wenn man sich an das Hauptfenster eines Programms erinnert, erkennt man sogar noch eine dritte Aufgabe, die mit diesen Methoden erledigt wird: die Bearbeitung der Titelzeile des Fensters.

```
class Cwnd:  
    void SetWindowText( LPCTSTR lpszString );  
    int GetWindowText( LPTSTR lpszStringBuf, int nMaxCount ) const;
```

Es ist ohne weiteres möglich, auch mehrzeilige Eingabefelder mit diesen beiden Methoden zu beschreiben oder auszulesen. Dazu muß man nur wissen, daß eine Zeilenschaltung als CRLF-Kombination dargestellt wird, also durch die aufeinanderfolgenden Zeichen mit dem ASCII-Code 13 und 10. In C- oder C++-Programmen entspricht dies der Zeichenkettenkonstanten »\r\n«.

Man muß allerdings aufpassen, wenn der Inhalt mehrzeiliger Eingabefelder in einer Datei gespeichert werden soll oder aus einer solchen kommt. Öffnet man nämlich eine Textdatei, werden beim Einlesen alle CRLF-Sequenzen in einzelne LF-Zeichen umgewandelt, während dies beim Schreiben der Datei umgekehrt ist (der Grund ist die angestrebte Quellcodekompatibilität der C-/C++-Compiler mit UNIX). So wird der Inhalt eines

Eingabefeldes, das ohne weitere Konvertierungen in einer Datei gespeichert wurde, CRCRLF-Sequenzen enthalten (die von vielen Editoren als doppelte Zeilenschaltungen interpretiert werden), und beim Einlesen einer korrekten Datei in ein Eingabefeld werden anstelle der Zeilenumbrüche nur senkrechte Striche angezeigt. Zur Lösung dieses Problems kann man die Datei entweder im Binärmodus öffnen oder beim Lesen und Schreiben der Daten eine geeignete Konvertierung durchführen.

GetLine: Manchmal will man nicht den ganzen Text lesen, sondern nur einzelne Zeilen. Hierzu gibt es die Methode *GetLine*, mit der das sehr einfach möglich ist:

GetLine liefert die *nIndex*-te Zeile aus dem Eingabefeld (die Zählung beginnt bei 0, die Anzahl der Zeilen kann mit *GetLineCount* bestimmt werden, s.u.) und schreibt sie in den Puffer *lpszBuffer*. Dabei werden insgesamt höchstens *nMaxLength* Zeichen in den Puffer übertragen. Der Rückgabewert gibt die Anzahl der tatsächlich übertragenen Zeichen an; er kann kleiner als *nMaxLength* sein, wenn die übertragene Zeile kürzer ist.

```
class Cedit:
    int GetLine( int nIndex, LPTSTR lpszBuffer,
                int nMaxLength ) const;
    int GetLine( int nIndex, LPTSTR lpszBuffer )
                const;
```

Zu beachten ist, daß die Zeilenschaltung (also die CRLF-Sequenz) nicht mitübertragen wird! In einem einzeiligen Eingabefeld wird diese Funktion nicht ausgeführt.



ReplaceSel: Um einen Teil des Textes zu verändern, kann die Methode *ReplaceSel* verwendet werden:

```
class Cedit:
    void ReplaceSel(LPCTSTR lpszNewText,
                   BOOL bCanUndo = FALSE);
```

ReplaceSel ersetzt die aktuelle Selektion, d.h., alle Zeichen des Eingabefeldes, die invers dargestellt werden, durch die nullterminierte Zeichenkette *lpszNewText*. Ist der gesamte Text markiert, enthält das Eingabefeld nach Aufruf der Methode nur noch den Inhalt von *lpszNewText*, ist hingegen gar kein Text markiert, so wird *lpszNewText* an der aktuellen Cursorposition eingefügt.

Schnittstelle zur Zwischenablage: Eine der angenehmsten *allgemeinen* Eigenschaften von Textfeldern ist ihre eingebaute Schnittstelle zur Zwischenablage. Mit der Tastenkombination **[Strg][X]** wird die Markierung ausgeschnitten und in die Zwischenablage übertragen, mit **[Strg][C]** wird

sie ohne Änderung des Eingabefeldes in die Zwischenablage kopiert. Zusätzlich kann durch Drücken der Tastenkombination `[Strg][V]` der Inhalt der Zwischenablage in das Eingabefeld eingefügt werden. Außerdem ist es möglich, durch Drücken von `[Entf]` die Markierung insgesamt zu löschen, ohne ihren Inhalt in die Zwischenablage zu übertragen.

Neben der Möglichkeit, diese Interaktion mit der Zwischenablage benutzergesteuert aufzurufen, kann dies mit Hilfe der Klasse *CEdit* auch programmgesteuert erfolgen:

```
class Cedit:
    void Copy();
    void Cut();
    void Paste();
```

Copy kopiert die Markierung in die Zwischenablage, ohne im Eingabefeld eine Veränderung vorzunehmen, *Cut* kopiert sie erst und löscht sie dann im Eingabefeld, und *Paste* fügt den Inhalt der Zwischenablage in das Eingabefeld ein.

Textanzeige

SetLimitText: Die Anzahl der Zeichen, die der Benutzer in einem Eingabefeld eingeben kann, läßt sich mit der Methode *SetLimitText* begrenzen:

```
class Cedit:
    void LimitText( int nChars = 0 );
    void SetLimitText( UINT nMax );
```

Wird *LimitText* nicht aufgerufen oder ist der Parameter `nChars` gleich 0, so ist die Länge des Textes auf 65.535 (UINT_MAX) Zeichen limitiert (genau ein Segment). Andernfalls wird die Anzahl der Zeichen, die der Benutzer eingeben darf, auf `nChars` begrenzt. Ein Aufruf von *LimitText* schneidet keine Zeichen von bereits eingegebenem Text ab (falls dieser schon länger als `nChars` ist), sondern verhindert nur, daß weitere Zeichen eingegeben werden. Auch Einfügungen aus der Zwischenablage sind nur erlaubt, wenn die vorgegebene Beschränkung eingehalten wird. Jede Zeilenschaltung zählt zwei Zeichen, da sie intern als CRLF-Sequenz dargestellt wird.



SetLimitText wird anstelle von *LimitText* in 32-Bit-Systemen (nicht Win32s!) verwendet, da keine Beschränkungen auf Segmentgrenzen gelten.

SetSel: Die Markierung in einem Eingabefeld wird normalerweise vom Benutzer gesetzt, indem er mit der Maus über einen Bereich zieht oder bei gedrückter `[⇧]`-Taste Cursorbewegungen durchführt. Mit einer kleinen (aber unschönen) Einschränkung ist es auch programmgesteuert möglich, die Markierung zu setzen:

```
class Cedit:
    void SetSel( int nStartChar, int nEndChar,
                BOOL bNoScroll = FALSE);
    void SetSel( DWORD dwSelection,
                BOOL bNoScroll = FALSE );
```

Die Methode *SetSel* setzt die Markierung ab dem Zeichen mit der Position *nStartChar* (das erste Zeichen im Eingabefeld hat dabei die Position 0) bis *vor* das Zeichen mit der Position *nEndChar*, also von *nStartChar* bis *nEndChar-1*. Soll der gesamte Text markiert werden, ist *nStartChar* auf 0 und *nEndChar* auf -1 zu setzen; soll die bisherige Markierung aufgelöst werden, ist *nStartChar* auf -1 zu setzen. Der Parameter *bNoScroll* gibt an, ob der Einfügepunkt gescrollt wird (FALSE) oder nicht (TRUE).

Der Schönheitsfehler von *SetSel* besteht nun leider darin, daß die Markierung zwar intern registriert wird, aber für den Benutzer unsichtbar bleibt, wenn das Eingabefeld nicht den Eingabefokus hat. Der markierte Text wird nicht hervorgehoben. Gleichwohl funktionieren alle Methoden, die mit der Markierung arbeiten, korrekt; auch jene, die mit der Zwischenablage kommunizieren. Wird die Methode aufgerufen, während das Eingabefeld den Eingabefokus hat, so ist die Hervorhebung sichtbar, und sogar der Cursor wird korrekt an das Ende der Markierung positioniert.

Nützlich ist *SetSel* beispielsweise, um eine Eigenschaft von Eingabefeldern auszuschalten, die manchmal etwas störend ist: das Hervorheben des gesamten Textes, unmittelbar nachdem das Eingabefeld durch Anwählen mit der -Taste den Eingabefokus erhalten hat. Sorgt man dafür, daß als Reaktion auf den Benachrichtigungscode `EN_SETFOCUS` die Methode *SetSel(0,0)* aufgerufen wird, so wird die Markierung zurückgenommen, und der Cursor steht am Anfang des Textes:

```
void CTestDlg::OnEnSetFocus()
{
    CEdit *pEdit = (CEdit*)GetDlgItem(IDD_EDIT);
    pEdit->SetSel(0,0);
}

...
BEGIN_MESSAGE_MAP(CTestDlg, CDialog)
    ...
    ON_EN_SETFOCUS(IDD_EDIT, OnEnSetFocus)
    ...
END_MESSAGE_MAP()
```

Listing: Zurücksetzen der Markierung des Eingabefeldes bei Fokus-Erhalt

LineScroll: Der Text in einem mehrzeiligen Eingabefeld kann nicht nur vom Benutzer gescrollt werden, sondern auch vom Programm. Dazu verfügt *CEdit* über die Methode *LineScroll*, mit der sowohl ein vertikales als auch ein horizontales Scrollen des Textes möglich ist:

```
class Cedit:
    void LineScroll(int nLines,int nChars = 0 );
```

Beide Parameter enthalten einen Offset, der angibt, wie weit der Text gescrollt werden soll. Dabei gibt *nLines* an, um wie viele Zeilen der Text nach unten gescrollt werden soll, und *nChars*, um wie viele Spalten nach rechts. Beide Werte können negativ sein, dann wird nach oben bzw. links gescrollt, oder sie können 0 sein, dann wird in diese Richtung nicht gescrollt. In vertikaler Richtung wird das Scrollen durch den Textanfang und das Textende begrenzt.

SetTabStops: Innerhalb von Eingabefeldern werden auch Tabulatoren interpretiert. Standardmäßig sind sie im Abstand von 32 horizontalen *Dialogboxeinheiten* angeordnet, sie können aber mit Hilfe der Methode *SetTabStops* verändert werden:

```
class Cedit:
    BOOL SetTabStops( int nTabStops,
                     LPINT rgTabStops );
    void SetTabStops();
    BOOL SetTabStops( const int& cxEachStop );
```

Bevor *SetTabStops* erklärt wird, soll kurz etwas zu den Dialogboxeinheiten gesagt werden, mit denen Koordinatenangaben innerhalb von Dialogboxen spezifiziert werden. Eine Dialogboxeinheit ist abhängig von der Größe des aktuellen Systemfonts; diese wiederum ist vom Bildschirmtreiber und von der Grafikauflösung abhängig. Eine Dialogboxeinheit in x-Richtung entspricht dabei einem Viertel der mittleren Breite eines Zeichens des Systemzeichensatzes, während eine Dialogboxeinheit in y-Richtung einem Achtel der Höhe des Systemzeichensatzes entspricht.

Der Grund für diese ungewöhnlichen Koordinatenangaben liegt darin, daß die Anzeige einer Dialogbox weitgehend von der Bildschirmauflösung unabhängig gemacht werden soll, um die Größe der Dialogbox und ihrer Steuerungen immer im richtigen Verhältnis zum verwendeten Zeichensatz zu setzen. Aus diesem Grund entsprechen immer vier x-Einheiten in einer Dialogbox der Breite eines durchschnittlichen Zeichens und acht y-Einheiten der Höhe eines Zeichens. Die Voreinstellung der Tabulatorweite von 32 Dialogboxeinheiten bedeutet also, daß etwa alle acht Zeichen ein Tabstop steht.

Da der Systemzeichensatz proportional ist, müssen nicht unbedingt *genau* acht Zeichen zwischen zwei Tabulatoren Platz finden. Stehen überwiegend breitere Zeichen (wie z. B. ein »w« oder ein »M«) an der Stelle, so passen weniger als acht Zeichen zwischen die Tabulatoren, bei schmalere Zeichen (»i« oder »l«) entsprechend mehr.

SetTabStops ist eine überladene Methode und existiert in drei Varianten. Die parameterlose Version ist am einfachsten zu erklären: Ein Aufruf führt zur Voreinstellung von 32 Dialogboxeinheiten je Tabulator, also etwa acht Zeichen. Wird die Version von *SetTabStops* mit dem Parameter *cxEachStop* aufgerufen, werden in einem gleichbleibenden Abstand von *cxEachStop* so viele Tabulatoren gesetzt, wie das Eingabefeld breit ist.

Die letzte Variante ermöglicht es sogar, unterschiedlich breite Spalten zu erzeugen. Dazu werden die gewünschten Tabulatorpositionen in der Reihenfolge der Spalten in einem *int*-Array abgelegt und dieses in dem Parameter *rgTabStops* an die Methode übergeben; in *nTabStops* muß die Anzahl der gültigen Array-Elemente übergeben werden. Wichtig bei dieser Version ist, daß nicht die Breiten der Spalten übergeben werden, sondern die absoluten Tabulatorpositionen. Sollen beispielsweise drei Spalten à 32 Zeichen angelegt werden, muß das Array die Werte {32, 64, 96} enthalten.

Der Aufruf einer der *SetTabStops*-Methoden wirkt sich zunächst nicht sichtbar auf das Eingabefeld aus, sondern ist lediglich für neu eingegebene Zeichen gültig. Nur wenn das Eingabefeld neu gezeichnet wird, werden die neuen Tabulatorpositionen auch in den vor dem Methodenaufruf eingegebenen Textteilen sichtbar. Will das Programm sicherstellen, daß die Änderungen sofort sichtbar werden, kann es auf dem Eingabefeld die Methode *InvalidateRect* anwenden.

Informationen

Neben den Methoden, mit denen der Text oder die Optik eines Eingabefeldes verändert werden können, gibt es noch eine ganze Reihe von Methoden, die Informationen über das Eingabefeld liefern. Dazu gehört beispielsweise die Anzahl der Zeilen, der Änderungszustand oder die Ausdehnung der aktuellen Markierung. Mit Hilfe dieser Methoden kann man die Funktionalität eines Eingabefeldes weiter erhöhen und zu einem System ausbauen, das flexibel an unterschiedliche Anforderungen angepaßt werden kann. Die wichtigsten dieser Methoden sollen im folgenden kurz vorgestellt werden.

GetModify: Hat man ein größeres Eingabefeld, in dem der Anwender einen längeren Text bearbeitet, kann es sinnvoll sein, feststellen zu lassen, ob der Text tatsächlich geändert wurde. Dies spielt insbesondere dann eine Rolle, wenn nach einer Textänderung ein längerer Programmteil ablaufen müßte, der eine Wartezeit verursachen würde.

Man kann herausfinden, ob der Text geändert wurde, indem man einen der Benachrichtigungscodes `EN_CHANGE` oder `EN_UPDATE` auf eine eigene Methode umleitet und darin ein entsprechendes Flag setzt. Soviel Aufwand braucht man aber gar nicht zu betreiben, denn genau diese Information wird bereits im Eingabefeld selbst gespeichert und kann mit der Methode *GetModify* von *CEdit* abgefragt werden:

```
class Cedit:
    BOOL GetModify();
```

GetModify liefert genau dann `TRUE` zurück, wenn der Inhalt des Eingabefeldes geändert wurde, andernfalls ist der Rückgabewert `FALSE`.

Es gibt zudem eine Möglichkeit, dieses interne Flag von außen zu verändern: Ein Aufruf der Methode *SetModify*, die einen booleschen Parameter benötigt, erlaubt es, das Flag zu setzen oder zu löschen.

GetSel: Um herauszufinden, über welchen Bereich sich die aktuelle Markierung des Textes erstreckt, kann die Methode *GetSel* aufgerufen werden:

```
class Cedit:
    DWORD GetSel();
    void GetSel( int& nStartChar, int& nEndChar ) const;
```

Ihr Rückgabewert ist ein Doppelwort, welches in der unteren Hälfte den Index des ersten Buchstabens der Markierung und in der oberen Hälfte den Index des ersten Buchstabens hinter der Markierung enthält. Die zweite Variante von *GetSel* entspricht dem Aufruf von *SetSel* mit zwei Integer-Werten.

GetLineCount: Mit *GetLineCount* kann bestimmt werden, wie viele Zeilen ein mehrzeiliges Eingabefeld enthält. Für einzeilige Eingabefelder ist ein Aufruf dieser Methode nicht definiert. Falls das Eingabefeld leer ist, wird 1 zurückgegeben.

```
class Cedit:
    int GetLineCount() const;
```

LineLength: Um die Länge einer Zeile zu bestimmen, kann die Methode *LineLength* verwendet werden.

```
class Cedit:
    int LineLength( int nLine = -1 ) const;
```

Sie erwartet als Parameter *nLine* den Index irgendeines Zeichens, das in der Zeile liegt, deren Länge bestimmt werden soll, oder -1, falls die Länge der Zeile bestimmt werden soll, in der der Cursor gerade steht. Der Rückgabewert entspricht der Länge der Zeile *ohne* die angehängte CRLF-Sequenz. Da bei der Berechnung des Parameters *nLine* die zwei zusätzlichen Zeichen je Zeilenschaltung aber berücksichtigt werden müssen, kann leicht Verwirrung entstehen, denn die Summe der mit dieser Methode ermittelten Zeilenlängen ist keinesfalls gleich der Länge des gesamten Textes. Statt dessen muß bei jeder Zeile, die eine CRLF-Sequenz enthält (normalerweise alle, aber wie steht es mit der letzten?), der Wert »zwei« addiert werden, um auf den korrekten Wert zu kommen.

LineIndex: Soll die Länge einer Zeile anhand ihres Zeilenindex bestimmt werden, müßte aus diesem erst mit einigem Aufwand der Index eines Zeichens in dieser Zeile ermittelt werden. Glücklicherweise gibt es hierfür die Methode *LineIndex* in *CEdit*:

```
class Cedit:
    int LineIndex( int nLine = -1 ) const;
```

LineIndex erwartet als Parameter *nLine* die laufende Nummer einer Zeile (beginnend mit 0) und liefert den Index des ersten Zeichens in der Zeile. Dabei werden Zeilenschaltungen korrekt berücksichtigt, so daß der Rückgabewert dieser Methode direkt als Input für *LineLength* verwendet werden kann, um die Länge der *nLine*-ten Zeile zu ermitteln. Ist *nLine* -1, so wird der Index des ersten Zeichens der Zeile mit dem Cursor zurückgegeben.

LineFromChar: Genau die umgekehrte Aufgabe von *LineIndex* erledigt *LineFromChar*:

```
class Cedit:
    int LineFromChar( int nIndex = -1 );
```

Sie liefert die Nummer der Zeile (bei 0 beginnend), in der das Zeichen mit dem Index *nIndex* steht. Dabei geht die Methode davon aus, daß bei der Bestimmung von *nIndex* die Zeilenschaltungen der vorhergehenden Zeilen mitgezählt wurden.

10.2.5 Die Klasse *CStatic*

Mit Hilfe statischer Dialogboxelemente lassen sich auf einfache Weise spezielle Ausgabebereiche in einer Dialogbox erzeugen. Dabei gibt es mehrere unterschiedliche Arten statischer Steuerungen, von denen zwei sogar vom Programm veränderbar sind. Als erstes gilt dies für statischen Text, dessen

Wert des Bezeichners (ID) ungleich -1 ist; hier ist es möglich, den Text zur Laufzeit zu verändern und die statische Steuerung als textuelles Ausgabefeld zu benutzen. Zweitens gibt es Statics, die ein Symbol anzeigen können; auch diese können zur Laufzeit verändert werden, indem das angezeigte Symbol gewechselt wird – damit lassen sich dann in Dialogboxen ein ganze Reihe einfacher grafischer Effekte erzielen.

Im folgenden sollen diese beiden Arten, statische Steuerungen zu verwenden, vorgestellt werden. Die übrigen Statics, etwa schwarze oder graue Kästen oder Rahmen, werden nicht weiter behandelt.

Textausgaben

Attribute: Statics für Textausgaben werden im Dialogeditor mit Hilfe des »Aa«-Symbols aus der Toolbox angelegt. Sie besitzen einen Bezeichner, einen anfänglichen Text und ein paar weitere Attribute, von denen die wichtigsten in Tabelle 10.9 aufgelistet sind.

Die Steuerung muß dabei genau eines der Attribute *Linksbündig*, *Zentriert* oder *Rechtsbündig* haben, um damit anzugeben, auf welche Weise der Text ausgerichtet werden soll. Optional kann das Attribut *Kein Präfix* gegeben werden, um dafür zu sorgen, daß das &-Zeichen nicht interpretiert wird.

Dialogeditor	Dialogboxschablone	Bedeutung
Text ausrichten Linksbündig, Zentriert, Rechtsbündig (Left, Center, Right)	Ressource-Typ: LTEXT, CTEXT, RTEXT Style: --	Der Text wird linksbündig, zentriert oder rechtsbündig ausgegeben. Wenn »Kein Umbruch« aktiv ist, kann der Text nur linksbündig ausgerichtet sein.
Vertikal zentrieren (Center vertically)	Ressource-Typ: ?TEXT Style: SS_CENTERIMAGE	Der Text wird vertikal in der Steuerung zentriert. So kann der Text einfach mittig vor ein Eingabefeld gesetzt werden, wenn Eingabefeld und statischer Text die gleiche Größe haben.
Kein Umbruch (No wrap)	Ressource-Typ: ?TEXT Style: SS_LEFTNOWORDWRAP	Der Text wird linksbündig und einzeilig dargestellt sowie ggf. rechts abgeschnitten.
Einfach (Simple)	Ressource-Typ: ?TEXT Style: SS_SIMPLE	Der Text wird weder abgeschnitten noch umgebrochen.
Vertieft (Sunken)	Ressource-Typ: ?TEXT Style: SS_SUNKEN	Der Text wird mit einer optischen Vertiefung dargestellt.

Dialogeditor	Dialogboxschablone	Bedeutung
Benachrichtigung (Notify)	Ressource-Typ: ?TEXT Style: SS_NOTIFY	An das Elternfenster wird eine Nachricht geschickt, wenn auf die Steuerung geklickt wurde.
Rand (Border)	Ressource-Typ: ?TEXT Style: WS_BORDER	Der Text wird mit einem schwarzen Rahmen versehen.
Kein Präfix (No Prefix)	Ressource-Typ: ?TEXT Style: SS_NOPREFIX	In den Text eingebettete &-Zeichen werden nicht interpretiert, d. h., sie führen nicht dazu, daß der nächste Buchstabe unterstrichen wird, sondern werden literal angezeigt.

Tabelle 10.9: Attribute von statischen Steuerungen

Programmierung: Statics besitzen keine Benachrichtigungs-codes, so daß es nicht erforderlich ist, Methoden zu entwickeln, die auf Ereignisse von Statics reagieren. Die wichtigste Art, auf ein statisches Textfeld zuzugreifen, besteht darin, den angezeigten Text zu verändern. Dazu dient keine Methode der Klasse *CStatic*, sondern – wieder einmal – die alte Bekannte *SetWindowText*, der Text auf dieselbe Weise in ein Textfeld schreibt wie in ein Eingabefeld. Bezüglich des Aufrufs dieser Methode sei auf die Ausführungen im Kapitel »Textoperationen« der Klasse *CEdit* verwiesen.

Insbesondere Zeilenschaltungen und Tabulatoren werden dabei genauso interpretiert wie in einem Eingabefeld. Darüber hinaus besitzen statische Textfelder die Fähigkeit, Text automatisch umzubrechen, und können damit sehr lange Zeichenketten automatisch formatieren und leserlich auf dem Bildschirm darstellen.

Bilder in Dialogboxen

Um ein Bild in einer Dialogbox darzustellen, ist mit Hilfe des Bild-Symbols aus der Symbolleiste *STEUERELEMENTE* ein Bild-Static einzubinden. Der Dialogeditor übersetzt dieses Bild in ein Ressource-Statement vom Typ *ICON* oder *CONTROL*. Die Bild-Steuerung ist sehr vielseitig. Sie läßt sich vom Typ her als Rahmen, Rechteck, Symbol oder Bitmap darstellen. Bei Symbol oder Bitmap kann aus den Ressourcen eines ausgewählt werden, das dann angezeigt wird. Typischerweise zeigt man in einer Info-Box zum Programm neben dessen Version auch noch das Programmsymbol an. Die verschiedenen Varianten probiert man am besten aus.

Interessant wird es erst, wenn man mit Hilfe der Methode *SetIcon* der Klasse *CStatic* ein Symbol zur Laufzeit lädt:

```
class Cstatic:
    HICON SetIcon( HICON hIcon );
```

SetIcon erwartet in *hIcon* einen Handle auf das Icon, das in der Steuerung angezeigt werden soll. Um diesen Handle zu bekommen, kann die globale Funktion *LoadIcon* aufgerufen werden. Sie erwartet zwei Parameter, den Instance-Handle des laufenden Programms und den Namen einer Symbol-Ressource:

```
HICON LoadIcon( HINSTANCE hInstance,
                LPCTSTR lpIconName );
```

Der Instance-Handle kann sehr einfach mit der globalen Funktion *AfxGetInstanceHandle* beschafft werden, und der Name ist die Zeichenkette, unter dem die Symbol-Ressource in der Ressource-Datei des Programms abgelegt wurde.

Um beispielsweise in einer Dialogbox, die ein Bild-Symbol mit dem Bezeichner *IDD_AMPEL* besitzt, die Symbol-Ressource *AMPELROT* anzuzeigen, kann folgender Code verwendet werden:

```
void CxyzDlg::SetMyIcon()
{
    ...
    CStatic *pStatic =
        (CStatic*)GetDlgItem(IDD_AMPEL);
    pStatic->SetIcon(::LoadIcon
        (AfxGetInstanceHandle(),"AMPELROT"));
    ...
}
```

Listing: Setzen eines Symbols für eine Static-Steuerung

Voraussetzung für das Gelingen dieser Aktion ist allerdings, daß mit dem Ressource-Editor eine Symbol-Datei *AMPEL.ICO* erstellt wurde, und in der Ressource-Datei das Icon korrekt definiert worden ist:

```
AMPELROT ICON ampelr.ico
```

10.3 Listenfelder und Kombinationsfelder

10.3.1 Listenfelder (Listboxen)

Aus *Listboxen* wurden im Microsoft-Deutsch *Listenfelder*. Diese sind wiederum nicht zu verwechseln mit *Listenelementen* (*List Controls*). Um diesem Begriffs-Wirrwarr aus dem Weg zu gehen, soll im vorliegenden Abschnitt einfach der englische Begriff *Listbox* verwendet werden.

Listboxen werden sehr häufig in Windows-Dialogen verwendet. Sie werden benötigt, um ein oder auch mehrere Elemente aus einer Liste auszuwählen, und haben damit eine ähnliche Funktion wie Optionsfelder. Allerdings gibt es zwei wichtige Unterschiede: Erstens enthalten Listboxen in der Regel wesentlich mehr Elemente als eine Gruppe von Optionsfeldern, und zweitens ist ihr Inhalt zur Compile-Zeit meist noch nicht bekannt, sondern wird erst zur Laufzeit des Programms generiert – typischerweise während oder unmittelbar vor der Anzeige der Dialogbox. Die *List Controls*, die diesen sehr ähnlich sind, bieten eine Reihe weiterer Möglichkeiten, auf die später eingegangen wird. Deshalb verdrängen sie auch zunehmend die »alten« Listboxen.

Beide Arten von Steuerungen dienen also dazu, dem Benutzer eine Auswahl zu ermöglichen. Dabei werden Listboxen vorzugsweise dann verwendet, wenn der dynamische Charakter der Elementzusammensetzung überwiegt oder sehr viele Elemente anzutreffen sind, Optionsfelder zu meist dann, wenn die Auswahlmenge konstant und sehr klein ist. Diese Dynamik wird dadurch unterstützt, daß eine Listbox mehr Elemente enthalten kann als das Fenster selbst auf einmal aufnehmen kann. Mit Hilfe eines Schiebereglers kann der angezeigte Ausschnitt in kleinen oder großen Schritten verschoben oder durch Bewegen des Schiebers wahlfrei positioniert werden. Dabei wird die visuelle Veränderung des angezeigten Ausschnitts in Echtzeit vorgenommen und ermöglicht so auch in großen Datenbeständen eine schnelle Positionierung.

Eine der bekanntesten Anwendungen von Listboxen findet sich in den Dialogen zum Öffnen oder Speichern einer Datei bei 16-Bit Windows-Programmen. Hier zeigen Listboxen die verfügbaren Verzeichnisse und Dateien an, und der Benutzer kann den gesuchten Namen durch einfaches Positionieren der Maus auswählen. Andere Anwendungen für Listboxen finden sich bei der Suche im Hilfeindex von Windows-Anwendungen.

10.3.2 Kombinationsfelder (Comboboxen)

Das Prinzip der *Auswahl aus einer Menge von Möglichkeiten* findet sich bei den meisten Steuerungen. Egal, ob es sich um ein Menü, ein Optionsfeld, ein Kontrollkästchen oder eine Listbox handelt, immer wird dem Benutzer auf dem Bildschirm angezeigt, wie er weiter vorzugehen hat. Im Gegensatz dazu bestehen *kommandoorientierte Benutzerschnittstellen* im wesentlichen aus einer (leeren) Texteingabezeile, innerhalb derer der Benutzer dem System seine Wünsche in Form syntaktisch korrekter Kommandos mitzuteilen hat.

Vor allem auf der konsequenten Realisierung auswahlorientierter Dialoge beruht die Einschätzung, daß Windows-Programme leichter zu bedienen seien als ihre entsprechenden Gegenstücke. Das einzige Benutzerschnitt-

stellenobjekt aus der Zeit der Kommandozeilen ist das Eingabefeld, das vom Benutzer die freie Eingabe einer Zahl, einer Zeichenkette oder eines Kommandos verlangt, ohne ihm auch nur die kleinste Auswahlmöglichkeit zu bieten.

Leider lassen sich Eingabefelder in Dialogen häufig nicht vermeiden, vor allem dann nicht, wenn Zeichenketten oder Dezimalzahlen verarbeitet werden müssen. Allerdings ist es manchmal möglich, die Notwendigkeit eines Eingabefeldes mit der Bequemlichkeit einer Listbox zu kombinieren, indem an der entsprechenden Stelle ein Kombinationsfeld verwendet wird. Weiß der Anwender, was er einzugeben hat, braucht er nur das Eingabefeld zu benutzen; weiß er es nicht, kann er den Listboxteil aufklappen und das gewünschte Element aus der angebotenen Liste auswählen. Als Alternative zu beiden Steuerungen wird in letzter Zeit sehr gern das *Drehfeld* benutzt. Damit können über kleine Buttons numerische Werte inkrementiert oder dekrementiert werden. Falls das einem Benutzer zu lange dauert, kann er den Wert immer noch über die Tastatur eingeben.

Kombinationsfelder kann man also entweder als Eingabefelder mit entschärftem Kommandozeilencharakter oder verbesserter Bedienung oder integrierter Hilfe ansehen. Andererseits ist auch die umgekehrte Betrachtungsweise möglich, nach der ein Kombinationsfeld nur eine Listbox ist, die zusätzlich die Möglichkeit der freien Eingabe bietet.

In der Praxis tauchen beide Varianten auf. Schon im Hauptfenster von Winword findet man die erste, wenn man die Zeichengröße auswählen will: Ein Eingabefeld, das die Eingabe einer Zahl zwischen 4 und 127 erlaubt, und das es zusätzlich gestattet, diese auch aus der integrierten Listbox auszuwählen. Die andere Variante findet sich bei der Auswahl eines Druckformats. Normalerweise wird die dort vorhandene Listbox bemüht, um eines der angebotenen Druckformate auszuwählen, alternativ kann der Name aber auch direkt in das zugehörige Eingabefeld eingegeben werden.

Untersucht man eine Reihe von Dialogen, so stellt man schnell fest, daß darin wesentlich mehr Kombinationsfelder als Listboxen auftauchen. Dies hat seinen Grund, denn Kombinationsfelder sind als Kombination aus den drei Steuerungen *Listbox*, *Eingabefeld* und *Schieberegler* sehr flexible Dialogboxelemente, die bei der Darstellung wenig Platz wegnehmen und (selten benötigte) Details nur bei Bedarf anzeigen. Zudem wird sich zeigen, daß Kombinationsfelder bezüglich der *Eigenschaften*, *Benachrichtigungscodes* und *Methoden* die mächtigeren Steuerungen sind. Listboxen bieten nur ganz wenige Fähigkeiten, die nicht auch von Kombinationsfeldern zur Verfügung gestellt werden könnten.

10.3.3 Attribute

Basic Styles

Die in Kapitel 10.1.2 erläuterten Grundeigenschaften *Sichtbar*, *Deaktiviert*, *Gruppe* und *Tabstop* sind sowohl Listboxen als auch Kombinationsfeldern eigen. Jedes dieser Grundattribute hat bei diesen Steuerungen genau die gleichen Auswirkungen wie bei einem Button, d. h., es sorgt dafür, daß die Box sichtbar, aktiv, mit der -Taste erreichbar oder erstes Element einer Gruppe ist. Es gelten ebenfalls die erweiterten Eigenschaften.

Es macht übrigens keinen Sinn, eine Gruppe von Kombinationsfeldern oder Listboxen zu bilden. Wäre – wie bei Gruppen üblich – nur das erste Element über die -Taste zu erreichen, so könnten das zweite und alle anderen Gruppenelemente nicht per Tastatur angesprochen werden, denn die dafür benötigten Cursortasten werden schon in den Listboxen bzw. Kombinationsfeldern selbst benötigt. Jede Listbox und jedes Kombinationsfeld eines Dialogs sollte also das Tabstop-Attribut haben.

Standardattribute von Listboxen

Die folgende Tabelle erläutert die gebräuchlichsten Attribute für Listboxen. Viele davon lassen sich in ähnlicher Form auch in Kombinationsfeldern wiederfinden, die Style-Konstanten beginnen dann jedoch mit dem Präfix CBS_ statt LBS_. Es gibt auch einige qualitative Unterschiede zwischen Listboxen und Kombinationsfeldern. So können Kombinationsfelder keinen horizontalen Schieberegler haben, interpretieren keine Tabulatoren und erlauben es nicht, mehrspaltige Listen darzustellen. Des weiteren bieten sie dem Benutzer nicht die Möglichkeit, mehrere Elemente gleichzeitig zu markieren.

Dialogeditor	Dialogboxschablone	Bedeutung
Rand (Border)	Ressource-Typ: LISTBOX Style: WS_BORDER	Versieht die Listbox mit einem Rahmen. (Da WS_BORDER die Voreinstellung ist, muß für den Fall, daß kein Rahmen gegeben werden soll, NOT WS_BORDER gesetzt werden.)
Sortieren (Sort)	Ressource-Typ: LISTBOX Style: LBS_SORT	Der Inhalt der Listbox wird alphabetisch sortiert, bei jedem Aufruf von <i>AddString</i> wird das neue Element lexikalisch einsortiert.

Tabelle 10.10: Attribute für Listboxen

Dialogeditor	Dialogboxschablone	Bedeutung
Benachrichtigung (Notify)	Ressource-Typ: LISTBOX Style: LBS_NOTIFY.	Die Dialogboxfunktion erhält einen Benachrichtigungscode, wenn auf ein Element der Listbox geklickt oder doppelgeklickt wird. (Da LBS_NOTIFY die Voreinstellung ist, braucht nur für den Fall, daß kein Rahmen gegeben werden soll, NOT LBS_NOTIFY gesetzt zu werden.)
Vert. Bildlauf Horiz. Bildlauf (Vert./Horiz. Scroll Bar)	Ressource-Typ: LISTBOX Style: WS_VSCROLL WS_HSCROLL	Die Listbox bekommt einen vertikalen/ horizontalen Schieberegler, wenn sie mehr Elemente enthält, als auf einmal dargestellt werden können.
Tabstops (Use Tabs)	Ressource-Typ: LISTBOX Style: LBS_USETABSTOPS	Bei der Darstellung der Zeichenketten werden Tabulatoren interpretiert, andernfalls als senkrechte Balken angezeigt.
Keine Gesamthöhe (No Integral Height)	Ressource-Typ: LISTBOX Style: LBS_NOINTEGRALHEIGHT	Die Höhe der Listbox entspricht bei der Darstellung genau derjenigen beim Erstellen mit dem Dialogeditor. Wird dieses Attribut nicht gegeben, so paßt Windows die Höhe der Listbox so an, daß keine unvollständigen Zeilen ausgegeben werden.
Auswahl Keine, Einfach, Mehrfach, Erweitert (None, Single, Multiple, Extended)	Ressource-Typ: LISTBOX Style: LBS_MULTIPLESEL LBS_EXTENDESEL LBS_NOSEL	Die Listbox erlaubt es, kein, ein oder mehr als ein Element zu selektieren. Bei <i>Mehrfach</i> wird bei jedem Mausklick der Markierungszustand des angewählten Elements umgeschaltet. Der selbe Effekt kann auch durch Drücken der Leertaste über dem gewünschten Element erreicht werden. <i>Erweitert</i> erlaubt es, zusammenhängende Bereiche von Elementen durch Mausklick bei gedrückter ☐ -Taste zu markieren sowie einzelne Elemente durch Mausklick bei gedrückter ☐ -Taste.
Mehrspaltig (Multicolumn)	Ressource-Typ: LISTBOX Style: LBS_MULTICOLUMN	In der Listbox werden die Einträge in mehreren Spalten dargestellt. Die Einträge werden von oben nach unten Spalte für Spalte gefüllt. In diesem Fall kann ein horizontaler Schieberegler von Nutzen sein.

Tabelle 10.10: Attribute für Listboxen

Standardattribute von Kombinationsfeldern

Kombinationsfelder besitzen sehr viel weniger Attribute als Listboxen. Zunächst muß der Typ eines Kombinationsfeldes aus einer der folgenden drei Möglichkeiten ausgewählt werden:

Dialogeditor	Dialogboxschablone	Bedeutung
Einfach (Simple)	Ressource-Typ: COMBOBOX Style: CBS_SIMPLE	Die Listbox des Kombinationsfeldes ist immer aufgeklappt. Die Bedienung kann entweder durch Auswählen eines Elements aus der Listbox oder durch Eingeben eines Wertes in das Eingabefeld erfolgen.
Dropdown (Drop-Down)	Ressource-Typ: COMBOBOX Style: CBS_DROPDOWN	Die Listbox wird nur aufgeklappt, wenn der Benutzer den Button mit dem Pfeil anklickt. Ansonsten gleicht die Bedienung derjenigen des einfachen Kombinationsfeldes.
Dropdown Listenfeld (Drop-Down List)	Ressource-Typ: COMBOBOX Style: CBS_DROPDOWNLIST	Diese Listbox hat dasselbe Aussehen wie die vorheriger. Es ist jedoch nicht möglich, Texteingaben vorzunehmen. Die Bedienung erfolgt ausschließlich über die Auswahl von Elementen aus der Listbox. Ein Kombinationsfeld dieses Typs ist nichts weiter als eine besonders platzsparende Listbox mit eingeschränkter Funktionalität.

Tabelle 10.11: Typ-Attribute von Kombinationsfeldern

Darüber hinaus gibt es Attribute, die das Aussehen und Verhalten des Kombinationsfeldes beeinflussen. Sie können unabhängig voneinander ein- oder ausgeschaltet werden und sind in der Tabelle 10.12 enthalten:

Dialogeditor	Dialogboxschablone	Bedeutung
Sortieren (Sort)	Ressource-Typ: COMBOBOX Style: CBS_SORT	Der Inhalt der Combobox wird alphabetisch sortiert, bei jedem Aufruf von <i>AddString</i> wird das neue Element lexikalisch einsortiert.
Vertikaler Bildlauf (Vert. Scroll Bar)	Ressource-Typ: COMBOBOX Style: WS_VSCROLL	Das Kombinationsfeld bekommt einen vertikalen Schieberegler, wenn es mehr Elemente enthält, als gleichzeitig auf dem Bildschirm dargestellt werden können.
Kleinbuchstaben Großbuchstaben (Lowercase Uppercase)	Ressource-Typ: COMBOBOX Style: CBS_UPPERCASE CBS_LOWERCASE	Alle Elemente des Kombinationsfeldes werden in Klein- bzw. Großbuchstaben dargestellt, unabhängig davon, wie sie eingegeben wurden.
Keine Gesamthöhe (No Integral Height)	Ressource-Typ: COMBOBOX Style: CBS_NOINTEGRALHEIGHT	Die Höhe des Kombinationsfeldes entspricht bei der Darstellung derjenigen beim Erstellen mit dem Dialogeditor. Wird dieses Attribut nicht gegeben, paßt Windows die Höhe der Combobox so an, daß keine unvollständigen Zeilen ausgegeben werden.

Tabelle 10.12: Weitere Attribute von Kombinationsfeldern

Des weiteren können bei der Einstellung der Eigenschaften von Kombinationsfeldern auch schon die Daten eingegeben werden. Dafür existiert eine eigene Registerkarte. Dies ist natürlich nur dann sinnvoll, wenn die Elemente schon feststehen und sich nicht ändern.

HEXE besitzt nur ein einziges Kombinationsfeld, um dem Benutzer die Auswahl und erneute Bearbeitung eines alten Ergebnisses zu ermöglichen. Diese Combobox ist vom Typ *Dropdown Listenfeld* und besitzt das Attribut *Vert. Bildlauf*.

10.3.4 Benachrichtigungscode

Ebenso wie Eingabefelder oder Buttons senden auch Listboxen und Kombinationsfelder ganz bestimmte Nachrichten, um die Dialogbox über besondere Ereignisse oder Veränderungen zu unterrichten. Diese Nachrichten können ebenfalls mit Hilfe vordefinierter Message-Map-Einträge auf Methoden der Dialogboxklasse umgeleitet werden und so das Verhalten der Anwendung beeinflussen.

Um den Benachrichtigungscode einer Listbox oder eines Kombinationsfeldes auf eine eigene Methode umzuleiten, geht man so vor, wie es im letzten Abschnitt beschrieben wurde:

1. In der Dialogboxklasse wird eine neue Methode deklariert, die bei Auftreten des Benachrichtigungscode ausgeführt werden soll.
2. In der Message-Map der Dialogboxklasse wird ein neuer Eintrag eingefügt, der die Verbindung zwischen Nachricht und Methode herstellt.
3. Die neue Methode wird implementiert.

Schritt 1 und 2 läßt man vom Klassen-Assistenten erledigen, um die Implementierung muß man sich weiterhin selbst kümmern. Tabelle 10.13 und Tabelle 10.14 geben – jeweils für Listboxen und Kombinationsfelder – eine Übersicht der wichtigsten Benachrichtigungscode und der zugehörigen Message-Map-Einträge. Betrachtet man beide Tabellen genau, so zeigt sich, daß alle Benachrichtigungscode, die von einer Listbox gesendet werden können, im Prinzip auch von einem Kombinationsfeld stammen könnten. Der Unterschied besteht nur darin, daß sie einmal mit dem Präfix *LBN_* und zum anderen mit *CBN_* beginnen.

Der Eindruck allerdings, die Benachrichtigungscode von Listboxen seien eine echte Untermenge der Benachrichtigungscode von Kombinationsfeldern, täuscht, denn die beiden Tabellen listen nur die wichtigsten Codes auf. Es gibt beispielsweise einen Listbox-Benachrichtigungscode *LBN_SELCANCEL*, der ohne Kombinationsfeld-Pendant ist.

Benachrichtigungscode	Message-Map-Eintrag
<i>CBN_CLOSEUP</i>	<i>ON_CBN_CLOSEUP</i> (<id>, <memberFxn>)
<i>CBN_DBLCLK</i>	<i>ON_CBN_DBLCLK</i> (<id>, <memberFxn>)
<i>CBN_DROPDOWN</i>	<i>ON_CBN_DROPDOWN</i> (<id>, <memberFxn>)

Benachrichtigungscode	Message-Map-Eintrag
<i>CBN_EDITCHANGE</i>	<i>ON_CBN_EDITCHANGE (<id>, <memberFxn>)</i>
<i>CBN_EDITUPDATE</i>	<i>ON_CBN_EDITUPDATE (<id>, <memberFxn>)</i>
<i>CBN_ERRSPACE</i>	<i>ON_CBN_ERRSPACE (<id>, <memberFxn>)</i>
<i>CBN_KILLFOCUS</i>	<i>ON_CBN_KILLFOCUS (<id>, <memberFxn>)</i>
<i>CBN_SELCHANGE</i>	<i>ON_CBN_SELCHANGE (<id>, <memberFxn>)</i>
<i>CBN_SETFOCUS</i>	<i>ON_CBN_SETFOCUS (<id>, <memberFxn>)</i>

Tabelle 10.13: Benachrichtigungscode von Kombinationsfeldern

Benachrichtigungscode	Message-Map-Eintrag
<i>LBN_DBLCLK</i>	<i>ON_LBN_DBLCLK (<id>, <memberFxn>)</i>
<i>LBN_ERRSPACE</i>	<i>ON_LBN_ERRSPACE (<id>, <memberFxn>)</i>
<i>LBN_KILLFOCUS</i>	<i>ON_LBN_KILLFOCUS (<id>, <memberFxn>)</i>
<i>LBN_SELCHANGE</i>	<i>ON_LBN_SELCHANGE (<id>, <memberFxn>)</i>
<i>LBN_SETFOCUS</i>	<i>ON_LBN_SETFOCUS (<id>, <memberFxn>)</i>

Tabelle 10.14: Benachrichtigungscode von Listboxen

Falls in den folgenden Abschnitten Benachrichtigungscode erklärt werden, die sowohl für Listboxen als auch für Kombinationsfelder gültig sind, wird bei der Erklärung das Präfix ?BN_ verwendet, um anzuzeigen, daß es sowohl CBN_ als auch LBN_ lauten könnte.

Eingabefokus

Bekommt eine Listbox oder ein Kombinationsfeld den Eingabefokus, teilt sie dies der zugehörigen Dialogbox über den Benachrichtigungscode ?BN_SETFOCUS mit. Der Eingabefokus wird immer dann erteilt, wenn der Anwender die Box mit der Maus oder über die Tastatur anwählt. Verliert sie den Eingabefokus, d. h., wählt der Benutzer eine andere Steuerung oder schließt er die Dialogbox, wird der Dialogboxklasse der Benachrichtigungscode ?BN_KILLFOCUS übermittelt.

Benutzereingaben

Bei jeder Änderung eines selektierten Elements senden Listboxen und Kombinationsfelder den Benachrichtigungscode ?BN_SELCHANGE an die Dialogbox. In Kombinationsfeldern und Listboxen mit einfacher Selektion ist dies immer dann der Fall, wenn der Benutzer ein Element mit der Maus anklickt – selbst dann, wenn es bereits markiert ist. In Listboxen mit mehrfacher Selektionsmöglichkeit wird dieser Benachrichtigungscode auch dann gesendet, wenn ein markiertes Element deselektiert wird.

Falls auf ein Element der Listbox doppelgeklickt wurde, sendet die Listbox den Benachrichtigungscode LBN_DBLCLK an die Dialogbox (begleitet von LBN_SELCHANGE). Ein Kombinationsfeld sendet nur dann den Code CBN_DBLCLK, wenn sie vom Typ *Einfach* ist, d.h., wenn ihre Listbox dauernd geöffnet ist. Ein Kombinationsfeld, deren Listbox nicht dauernd geöffnet ist, akzeptiert dagegen keinen Doppelklick, sondern schließt bereits nach dem ersten Klick die Listbox, so daß der zweite ins Leere geht.

Der Dialog von HEXE überwacht die Auswahl eines früheren Ergebnisses aus dem Kombinationsfeld mit dem Bezeichner IDD_OUTPUT_LIST durch Umleiten des Benachrichtigungscode CBN_SELCHANGE auf die Methode OnOldResultSelected:

```
void CHEXEDlg::OnOldResultSelected()
{
    areg = m_cbListe.GetItemData(m_cbListe.GetCurSel());
    new_number = TRUE;
    GetDlgItem(IDD_BUTTON_EQUAL)->SetFocus();
    UpdateCalculatorDisplay();
}
BEGIN_MESSAGE_MAP(CHEXEDlg, CDialog)
...
    ON_CBN_SELCHANGE(IDD_OUTPUT_LIST, OnOldResultSelected)
...
END_MESSAGE_MAP()
```

Listing: Reaktion auf die Auswahl im Listen- oder Kombinationsfeld

In OnOldResultSelected wird zunächst das aktuelle Element des Kombinationsfeldes bestimmt und der darunter gespeicherte Wert gelesen. Anschließend wird der Eingabefokus auf die Gleichheitstaste gelegt und die Bildschirmanzeige durch Aufruf von UpdateCalculatorDialog auf den neuesten Stand gebracht. Die genaue Arbeitsweise der Methode (insbesondere die Bedeutung der Methoden GetCurSel und GetItemData der Klasse CComboBox) wird deutlich, wenn der Abschnitt über die Klassen CComboBox und CListBox durchgearbeitet wurde.

Bei Listboxen gibt es eine wichtige Besonderheit zu beachten. Besitzt eine Listbox nicht das Notify-Attribut, werden die Benachrichtigungscode LBN_SELCHANGE und LBN_DBLCLK unterdrückt, und die Dialogbox erfährt nichts von Benutzereingaben in der Listbox. Um diese beiden Nachrichten zu erhalten, muß also unbedingt das Benachrichtigungs-Attribut gesetzt werden. Bei Kombinationsfeldern gibt es das Notify-Attribut nicht, und die Benachrichtigungscode CBN_SELCHANGE und CBN_DBLCLK werden in jedem Fall gesendet, wenn der Benutzer entsprechende Eingaben vorgenommen hat.

Spezielle Kombinationsfeld-Benachrichtigungscodes

Die bisherigen Benachrichtigungscodes waren sowohl für Kombinationsfelder als auch für Listboxen gültig und konnten daher gemeinsam betrachtet werden. Darüber hinaus besitzt ein Kombinationsfeld weitere Benachrichtigungscodes, die aufgrund seiner besonderen Eigenschaften erforderlich sind und den Zustand der Listbox und des Eingabefeldes anzeigen.

Wird in einem Kombinationsfeld des Typs *Dropdown* oder *Dropdown Liste* der Listboxteil aufgeklappt, so sendet das Kombinationsfeld den Benachrichtigungscod `CBN_DROPDOWN` an die Dialogbox. Leider ist nicht genau definiert, ob der Benachrichtigungscod unmittelbar vor oder nach der visuellen Darstellung der Listbox gesendet wird. Wird die Listbox wieder geschlossen, so sendet das Kombinationsfeld den Benachrichtigungscod `CBN_CLOSEUP` an die Dialogbox. Falls das Kombinationsfeld vom Typ *Einfach* ist, wird weder der eine noch der andere Code gesendet, denn die Listbox ist ja die ganze Zeit aufgeklappt.

Auch das Eingabefeld eines Kombinationsfeldes ist in der Lage, Benachrichtigungscodes zu senden. Es verhält sich dabei genauso wie ein gewöhnliches Eingabefeld und sendet die Codes `CBN_EDITCHANGE` bzw. `CBN_EDITUPDATE`, wenn der Benutzer den Inhalt geändert hat. Der Unterschied zwischen den beiden Benachrichtigungscodes besteht darin, daß `CBN_EDITUPDATE` gesendet wird, *bevor* die Änderung auf dem Bildschirm sichtbar ist, während die Benachrichtigung `CBN_EDITCHANGE` erst *danach* erfolgt.

Ebenso wie bei gewöhnlichen Textfeldern kommen diese Nachrichten immer dann, wenn sich der Text geändert haben *könnte*, d.h., wenn der Benutzer Eingaben in dem Eingabefeld vorgenommen hat, die normalerweise zu Veränderungen führen würden. Wird der Inhalt des Eingabefeldes jedoch dadurch verändert, daß der Benutzer eines der Elemente aus der Listbox auswählt (und auf diese Weise in das Eingabefeld kopiert), wird keiner dieser Benachrichtigungscodes gesendet, sondern lediglich `CBN_SELCHANGE`. Um alle visuellen Änderungen des Eingabefeldes zu erkennen, reicht es also nicht aus, die Benachrichtigungscodes `CBN_EDITCHANGE` bzw. `CBN_EDITUPDATE` abzufangen, sondern die Dialogboxklasse muß auch eine Methode für den Benachrichtigungscod `CBN_SELCHANGE` zur Verfügung stellen.

Fehler

Ähnlich wie bei Eingabefeldern kann es auch bei Kombinationsfeldern oder Listboxen vorkommen, daß eine bestimmte Anforderung aus Speichermangel nicht ausgeführt werden kann. Taucht ein solches Problem auf, wird der Benachrichtigungscod `?BN_ERRSPACE` an die Dialogbox gesendet.

10.3.5 Die Klassen *CComboBox* und *CListBox*

Um aus dem Programm heraus auf eine Listbox oder ein Kombinationsfeld zuzugreifen, gibt es die Klassen *CListBox* und *CComboBox*. Die Methoden dieser Klasse ermöglichen das Einfügen oder Löschen von Elementen, das Lesen oder Setzen der Markierung (einschließlich mehrfacher Selektionen), bei einem Kombinationsfeld das Bearbeiten des Eingabefeldes, das Auf- und Zuklappen der zugehörigen Listbox und noch vieles andere mehr.

Um aus einer Dialogbox, die mit dem Dialogeditor erstellt wurde, zur Laufzeit des Programms auf ein Kombinationsfeld oder eine Listbox zuzugreifen, geht man genau wie bei den anderen Steuerungen vor. Man ruft die Methode *GetDlgItem* auf, konvertiert den zurückgegebenen *CWnd*- in einen *CComboBox*- oder *CListBox*-Zeiger und ruft die gewünschten Methoden auf. Danach besitzt man einen temporären Zeiger auf eine List- bzw. Combobox, mit dem die gewünschte Steuerung manipuliert werden kann. Die Rückgabe des temporären Objekts wird von den Foundation Classes automatisch beim nächsten Lauf des *Garbage Collectors* erledigt. Auch hier kann als Alternative zu *GetDlgItem* mit dem Klassen-Assistenten eine Member-Variable für die Steuerung angelegt werden.

Ebenso wie die Klasse *CEdit* besitzen auch *CComboBox* und *CListBox* sehr viele Methoden, die an dieser Stelle nicht alle besprochen werden können. Die nachfolgenden Abschnitte liefern jedoch einen Überblick über die wichtigsten und erklären, welche Aufgaben sich damit lösen lassen. Weitere Informationen finden sich in der elektronischen Hilfe (siehe *CListBox*, *CComboBox*).

Bearbeiten der Elemente

Die Angaben im vorliegenden Abschnitt sowie dem Abschnitt »Selektion von Elementen« gelten sowohl für Listboxen als auch für Kombinationsfelder. Um nicht ständig beide Begriffe anführen zu müssen, werden alle Erklärungen im Text auf Listboxen bezogen, gelten aber auch für Kombinationsfelder. Erst in den darauffolgenden Abschnitten werden Methoden besprochen, die entweder nur in *CListBox* oder nur in *CComboBox* definiert sind. Auf die eventuell vorhandenen kleinen Unterschiede zwischen beiden Arten von Steuerungen wird jeweils gesondert verwiesen.

AddString: Nach dem Anlegen ist eine Listbox leer, sie enthält also keine Elemente. Das Einfügen neuer Elemente ist ausschließlich zur Laufzeit möglich, es gibt also keine vordefinierten Listboxen, deren Elemente beispielsweise in der Ressource-Datei festgelegt werden könnten.

Jedes Element einer Listbox besteht aus zwei Teilen, einer Zeichenkette und einem *Daten-Langwort*, dessen Verwendung optional ist. Die Operationen zum Bearbeiten der Listbox sind etwas ambivalent: Bezüglich der

Einfügeoperationen entsprechen sie denen einer linearen Liste, bei der es möglich ist, an beliebiger Stelle ein Element einzufügen; beim lesenden Zugriff dagegen ähneln sie mehr einem Array, bei dem für jeden Zugriff auf ein Element dessen numerischer Index bekannt sein muß.

Die wichtigste Methode zum Einfügen eines neuen Elements ist *AddString*:

```
class CListBox:  
class CComboBox:  
    int AddString( LPCTSTR lpszItem );
```

Mit *AddString* wird die nullterminierte Zeichenkette, auf die der Zeiger *lpszItem* zeigt, in die Listbox eingefügt. Falls die Listbox das *Sort*-Attribut hat, wird sie an der Stelle eingefügt, die ihrer lexikografischen Einordnung entspricht, andernfalls am Ende der Listbox angehängt. Beim Einfügen der Zeichenkette wird diese *kopiert*, d. h. der Puffer, auf den der Zeiger *lpszItem* zeigt, kann anschließend weiterverwendet oder ungültig werden. Der Rückgabewert von *AddString* ist der nullbasierende Index der Zeichenkette in der Listbox bzw. *LB_ERR* oder *LB_ERRSPACE*, falls ein Fehler aufgetreten ist (*CB_ERR* bzw. *CB_ERRSPACE* bei Comboboxen).

Es ist ohne weiteres möglich, Zeichenketten doppelt einzufügen, Vorsicht ist allerdings geboten, wenn man dies mit Kombinationsfeldern tut. Öffnet der Benutzer die zugehörige Listbox und wählt eines der mehrfach vorhandenen Elemente aus, so wird dessen Name zwar korrekt in das Eingabefeld übertragen, beim nächsten Öffnen der Listbox wird die Markierung jedoch auf das *erste* Element mit dem im Eingabefeld angegebenen Namen gesetzt – unabhängig davon, ob dies wirklich das ursprünglich ausgewählte war.

InsertString: Will man ein neues Element ganz gezielt an einer bestimmten Stelle in die Listbox einfügen, so kann man dazu die Methode *InsertString* verwenden:

```
class CListBox:  
class CComboBox:  
    int InsertString( int nIndex,  
                     LPCTSTR lpszItem );
```

InsertString erwartet wie *AddString* einen Zeiger auf die einzufügende Zeichenkette in *lpszItem* und zusätzlich einen numerischen Index *nIndex*. Dieser gibt die Stelle an, an der das neue Element nach dem Einfügen stehen soll; alle weiter hinten stehenden Elemente werden um eine Position nach hinten geschoben. Die Zählung für *nIndex* beginnt bei 0; soll das neue Element an das Listenende angehängt werden, so kann -1 übergeben

werden. Auch *InsertString* gibt den nullbasierenden Index der Position zurück, an der die Zeichenkette eingefügt wurde – bzw. `LB_ERR` oder `LB_ERRSPACE`, wenn ein Fehler aufgetreten ist (`CB_ERR` bzw. `CB_ERRSPACE` bei Kombinationsfeldern).

Im Gegensatz zu *AddString* nimmt *InsertString* keine Rücksicht darauf, ob die Listbox mit dem *Sort*-Attribut versehen wurde. Das Element wird in jedem Fall an der angegebenen Position eingefügt, auch wenn dadurch eine eventuell vorhandene Sortierung zerstört werden sollte.

DeleteString und ResetContent: Natürlich ist es auch möglich, vorhandene Elemente zu löschen. Dabei gibt es einerseits die Möglichkeit, mit *ResetContent* alle Elemente zu entfernen, oder andererseits mit *DeleteString* ein ganz bestimmtes:

```
class CListBox:
class CComboBox:
    int DeleteString( UINT nIndex );
    void ResetContent();
```

DeleteString entfernt das Element mit dem Index `nIndex` und schiebt alle nachfolgenden Elemente eine Position nach unten. Der Rückgabewert von *DeleteString* ist die Anzahl der Elemente, die nach dem Löschen noch in der Liste verbleiben, bzw. `LB_ERR` (`CB_ERR` bei Kombinationsfeldern), wenn ein Fehler aufgetreten ist. *ResetContent* löscht alle Elemente aus der Listbox und stellt so deren Zustand wieder her, wie unmittelbar nach ihrem Anlegen.

SetItemData und GetItemData: Oben wurde bereits erwähnt, daß jedes Element einer Listbox neben der Zeichenkette noch einen zweiten Teil enthält: ein Langwort zum Speichern zusätzlicher Daten. Durch geschickte Verwendung dieses Langworts kann sehr oft vermieden werden, parallel zu der Listbox eine Datenstruktur mit analogen Einfüge- und Löschoperationen anzulegen, die für die Verwaltung zusätzlicher Daten zuständig ist. Oft ist der visuelle Inhalt der Listbox nämlich nur ein Bezeichner, der für den Bediener von Bedeutung ist – während die eigentlich interessanten Daten eine ganz andere Form haben.

Man kann jedem Element der Listbox ein zusätzliches Datum in Form eines Langworts zuordnen, mit dem die gewünschte Verbindung hergestellt wird. Zwar läßt sich in einem einzelnen Langwort nicht sehr viel unterbringen, aber für eine Referenz, d. h., einen Zeiger oder einen Schlüssel auf einen längeren Datensatz, reicht es allemal. Der Zugriff auf den Datenteil eines Elements erfolgt dabei mit den Methoden *SetItemData* und *GetItemData*:

```

class CListBox:
class CComboBox:
    int SetItemData( int nIndex,
                    DWORD dwItemData );
    DWORD GetItemData( int nIndex );

```

Der Zugriff beider Methoden auf die Elemente erfolgt über deren Position in der Listbox. Mit *SetItemData* bekommt der Datenteil des Elements mit dem Index *nIndex* den Wert *dwItemData* zugewiesen. Danach bleiben Daten- und Textteil dieses Elements für immer miteinander verbunden, unabhängig von zukünftigen Einfüge- oder Löschoperationen. Falls beim Aufruf der Methode ein Fehler aufgetreten ist, wird *LB_ERR* (*CB_ERR*) zurückgegeben.

Um umgekehrt auf den Datenteil des Elements mit dem Index *nIndex* zuzugreifen, ist die Methode *GetItemData* aufzurufen. Sie liefert als Rückgabewert die zugehörigen Daten bzw. *LB_ERR* (*CB_ERR*) bei einem Fehler. Falls *GetItemData* für ein Element aufgerufen wird, dem noch nicht mit *SetItemData* ein Wert zugewiesen wurde, ist der Rückgabewert undefiniert.

In HEXE wird nach jedem Betätigen der [=]-Taste das Ergebnis berechnet, in der Variablen *areg* gespeichert und für einen späteren Zugriff in die Combobox eingefügt:

```

void CHEXEDlg::OnEqual()
{
    //Ergebnis ermitteln und in areg ablegen
    ...
    //---Combobox aktualisieren---
    m_cbListe.EnableWindow(TRUE);
    m_cbListe.DeleteString(MAXRESULT-1);
    m_cbListe.InsertString(0,GetBasedString(areg,nbase,TRUE));
    m_cbListe.SetItemData(0,areg);
    m_cbListe.SetCurSel(0);
    UpdateCalculatorDisplay();
}

```

Listing: Einfügen von Elementen in ein Kombinationsfeld

Damit der Benutzer bei der Auswahl noch feststellen kann, welches Zahlensystem zum Zeitpunkt der Berechnung aktiv war, wird die Zahl bei der Umwandlung in eine Zeichenkette mit dem Präfix *x*, *d*, *o* oder *b* versehen. Um es später beim Wiedereinlesen des Wertes leichter zu haben, wird der Inhalt von *areg* durch Aufruf von *SetItemData* zusätzlich im Datenteil des Elements abgelegt. Beim Aufruf ist es also nicht erforderlich, den angezeigten Wert zu parsen, sondern es reicht aus, mit *GetItemData* den zusätzlich gespeicherten Wert auszulesen.

Selektion von Elementen

GetCurSel und SetCurSel: Die Auswahl eines Elements in einer Listbox wird vom Benutzer durch Setzen der Markierung erledigt. Um vom Programm aus abzufragen, welches Element gesetzt ist, gibt es die Methode *GetCurSel*, mit der der Index des markierten Elements ermittelt werden kann:

```
class CListBox:  
class CComboBox:  
    int GetCurSel();  
    int SetCurSel( int nSelect );
```

GetCurSel liefert den nullbasierten Index des markierten Elements oder LB_ERR (CB_ERR), falls kein Element markiert oder ein Fehler aufgetreten ist. Diese Methode sollte nur für Listboxen mit einfacher Selektion angewendet werden, bei Listboxen mit mehrfacher Selektion liefert sie ebenfalls LB_ERR. Wie man die Markierung von Listboxen mit mehrfacher Selektion ermittelt, wird im nächsten Abschnitt erklärt.

In HEXE wird beim Auswählen eines Elements aus dem Kombinationsfeld die Methode *OnOldResultSelected* aufgerufen, um das selektierte Ergebnis erneut in die Anzeige zu bringen. Da dieser Wert beim Drücken der Ergebnistaste im Datenteil des Elements abgelegt wurde, kann er mit *GetItemData* wiedergeholt werden:

```
void HEXEDlg::OnOldResultSelected()  
{  
    areg = m_cbListe.GetItemData(m_cbListe.GetCurSel());  
    new_number = TRUE;  
    GetDlgItem(IDD_BUTTON_EQUAL)->SetFocus();  
    UpdateCalculatorDisplay();  
}
```

Listing: Daten des aktuellen Eintrags des Kombinationsfeldes auslesen

Dazu wird zunächst ein Zeiger auf das Kombinationsfeld mit den Ergebnismwerten geholt und mit *GetCurSel* der Index des markierten Elements bestimmt. Dieser wird als Parameter von *GetItemData* dazu verwendet, den gespeicherten Zahlenwert in der Variablen *areg* abzulegen, um ihn erneut für die Berechnung zugänglich zu machen.

Mitunter ist es erforderlich, die Markierung einer Listbox vom Programm aus zu setzen, beispielsweise beim Initialisieren eines Dialogs. Hierzu kann die Methode *SetCurSel* verwendet werden, die das Element mit dem Index *nSelect* markiert. Falls das gewünschte Element nicht im gerade angezeigten Ausschnitt liegt, wird dieser so verschoben, daß das markierte Element

sichtbar wird; bei Kombinationsfeldern wird zusätzlich der Inhalt des Elements in das Eingabefeld kopiert. Soll die aktuelle Markierung rückgängig gemacht werden, so daß gar kein Element mehr markiert ist, kann -1 in `nSelect` übergeben werden.

Auch die Methode `SetCurSel` darf nur bei Listboxen mit einfacher Selektion aufgerufen werden, andernfalls liefert sie den Fehlercode `LB_ERR` zurück.

GetCount: Die Anzahl der in der Listbox befindlichen Elemente kann mit der Methode `GetCount` bestimmt werden:

```
class CListBox:
class CComboBox:
    int GetCount();
```

Der von `GetCount` zurückgegebene Wert ist um 1 größer als der Index des letzten Elements. Falls beim Aufruf der Methode ein Fehler aufgetreten ist, wird `LB_ERR` (`CB_ERR`) zurückgegeben.

Listboxen mit Mehrfachselektion

SetSel: Das Setzen von Elementen in Listboxen mit mehrfacher Selektion funktioniert etwas anders als in einfachen Listboxen. Da beim Anklicken eines Elements die Markierungen der anderen Elemente unbeeinflußt bleiben, muß es neben der Möglichkeit, Markierungen zu setzen, noch eine Möglichkeit geben, diese wieder aufzuheben. Beide Aufgaben lassen sich mit der Methode `SetSel` realisieren:

```
class CListBox:
    int SetSel( int nIndex, BOOL bSelect = TRUE);
```

`SetSel` verändert die Markierung des Elements mit dem Index `nIndex`, in Abhängigkeit vom Inhalt des Parameters `bSelect`. Ist `bSelect TRUE`, wird die Markierung gesetzt, andernfalls wird sie rückgängig gemacht. Falls `nIndex` gleich -1 ist, werden alle Elemente der Listbox gemäß `bSelect` verändert.

Diese Methode ist nur für *Listboxen* definiert, Kombinationsfelder erlauben keine Mehrfachselektionen. Sie darf außerdem nicht auf Listboxen mit einfacher Selektion angewendet werden, bei Auftreten eines Fehlers gibt sie `LB_ERR` zurück.

GetSelItems und GetSelCount: Während das Setzen der Markierung noch Schritt für Schritt erledigt werden konnte, kann es beim Abfragen der Markierung passieren, daß mehr als ein Element zurückgegeben werden muß. Die Methode `GetSelItems`, die für das Lesen der Markierungen zuständig ist, bekommt zu diesem Zweck ein (leeres) *int*-Array übergeben:

```
class CListBox:
    int GetSelItems( int nMaxItems, LPINT rgIndex );
    int GetSelCount();
```

Nach dem Aufruf enthält das Array zu jedem markierten Element der Listbox einen *int*-Wert mit dem Index des Elements. Sind also beispielsweise die Elemente 6, 7, 12 und 16 markiert, so sind nach dem Aufruf von *GetSelItems* die Elemente 0 bis 3 des Arrays mit diesen Werten belegt. Dem Rückgabewert kann man entnehmen, wie viele Werte tatsächlich in das Array geschrieben wurden.

Um ein Überlaufen des Arrays zu verhindern, übergibt der Aufrufer in *nMaxItems* die Anzahl der Elemente des übergebenen Arrays. Auch wenn mehr Elemente markiert sind, belegt *GetSelItems* höchstens *nMaxItems* Arraypositionen. Um vor dem Aufruf festzustellen, wie viele Elemente in der Listbox tatsächlich markiert sind, kann *GetSelCount* aufgerufen werden. Diese Methode liefert die Anzahl der markierten Elemente oder *LB_ERR*, falls ein Fehler aufgetreten ist.

Das Eingabefeld eines Kombinationsfeldes

SetWindowText und GetWindowText: Nachdem im vorigen Abschnitt einige Methoden erklärt wurden, die nur für Listboxen von Bedeutung sind, handelt dieser Abschnitt von Methoden, die nur für Kombinationsfelder gelten.

Das Eingabefeld eines Kombinationsfeldes kann nicht nur durch Verändern der Markierung manipuliert werden, sondern bei Listboxen, die nicht vom Typ Dropdown-Liste sind, auch durch manuelle Eingaben des Benutzers. Dabei muß die Eingabe des Benutzers nicht unbedingt mit dem Inhalt irgendeines Elements der Listbox übereinstimmen, sondern kann völlig frei gewählt werden.

Mit der Methode *SetWindowText* läßt sich der Inhalt des Eingabefeldes auch vom Programm aus verändern:

```
class CWnd:
    void SetWindowText( LPCTSTR lpszString );
    int GetWindowText( LPTSTR lpszStringBuf,
        int nMaxCount ) const;
```

SetWindowText schreibt den Inhalt des Textpuffers *lpszString* in das Eingabefeld eines Kombinationsfeldes und läßt dabei die Markierung in der Listbox unverändert. Nach dem Aufruf dieser Methode wird also wahrscheinlich der Inhalt des Eingabefeldes nicht mehr mit dem markierten

Text übereinstimmen. Um beim Benutzer keine Mißverständnisse aufkommen zu lassen, welcher der beiden Texte gültig ist, sollte vor dem Verändern des Eingabefeldes die Markierung gelöscht werden:

```
{
...
CComboBox *pCBox = (CComboBox*)GetDlgItem(IDD_COMBOBOX);
pCBox->SetCurSel(-1);
pCBox->SetWindowText("Im Eingabefeld steht...");
...
}
```

Listing: SetWindowText bei Kombinationsfeldern

Ebenso wie das Schreiben des Eingabefeldes ist auch das Lesen möglich. Dazu erwartet die Methode *GetWindowText* einerseits einen Zeiger `lpszStringBuf` auf einen Pufferbereich, und andererseits die Anzahl `nMaxCount` der maximal zu kopierenden Zeichen. Nach dem Aufruf befindet sich in `lpszStringBuf` eine nullterminierte Zeichenkette mit dem Inhalt des Eingabefeldes.

10.4 Bildlaufleiste/Schieberegler

10.4.1 Anwendungen

Als ein weiterer Steuerungstyp werden in diesem Abschnitt Bildlaufleisten oder Schieberegler behandelt. Da ihr Verhalten als Steuerung in einer Dialogbox dem Verhalten als Steuerung eines Hauptfensters sehr stark ähnelt, werden hier nur die wichtigsten Eigenschaften von Schiebereglern noch einmal kurz zusammengefaßt. Alle darüber hinausgehenden Fragen können mit Hilfe von Abschnitt 8 geklärt werden, in dem Schieberegler als Elemente des Hauptfensters bereits ausführlich erläutert wurden.

Die typischen Anwendungen für die Verwendung von Schiebereglern sind die Eingabe und Anzeige *quasi-analoger* Werte. Überall dort, wo aus einer großen Menge ein Wert ausgewählt werden soll und es nicht unbedingt auf Präzision, sondern mehr auf Übersichtlichkeit ankommt, sind Schieberegler die richtigen Eingabeinstrumente.

Schieberegler finden sich in den Dialogen vieler Windows-Programme, allerdings sind sie bei weitem nicht so verbreitet wie die anderen Dialogboxelemente. Zunächst werden sie in Listboxen oder Kombinationsfeldern verwendet, um deren Inhalt zu positionieren, wenn er größer als der Bildschirmausschnitt ist. Zudem findet man Schieberegler beispielsweise in der Systemsteuerung zur Einstellung der Mausempfindlichkeit, des Tastaturanschlags oder beim Mischen eigener Bildschirmfarben.

Schieberegler sind übrigens die einzigen vordefinierten Steuerungen, die sich nicht über die Tastatur bedienen lassen. Selbst wenn man mit der Maus auf einen Schieberegler klickt und ihm so den Eingabefokus überträgt, gibt es keine Möglichkeit, den Schieber mit Hilfe von Tastatureingaben zu bewegen. Auch das Klicken auf die Pfeile oder in der Schiebeleiste läßt sich nicht durch Tastatureingaben simulieren – obwohl es durchaus Sinn machen würde, den Cursortasten diese Bedeutung zu geben.



Auf Notebooks, die mit einem Touchpad ausgestattet sind, existiert zum Teil die Möglichkeit, einen gesonderten Bereich einzurichten (der rechte Rand des Touchpads). In diesem Bereich wird dann ein Druck der Maustaste auf die Pfeile des Schiebereglers simuliert.

Aus diesem Grund ist es wenig sinnvoll, einem Schieberegler das *Tabstop*-Attribut zu geben, denn er kann mit dem Eingabefokus nichts anfangen. Zudem ist jeder Dialog, der wenigstens einen Schieberegler enthält, nicht mehr vollständig zu bedienen, wenn die Maus fehlt. Um dieses Problem zu beheben, müßte entweder auf den Schieberegler verzichtet oder die entsprechende Tastaturschnittstelle zusätzlich eingebaut werden.

10.4.2 Attribute

Ein Schieberegler besitzt keine Attribute mit Ausnahme der *Basic Styles*, die zur Einstellung der Grundeigenschaften *Sichtbar*, *Deaktiviert*, *Gruppe* und *Tabstop* dienen. Auch bei einem Schieberegler haben diese Attribute dieselben Auswirkungen wie bei einem Button und sorgen dafür, daß er sichtbar, aktiv, mit der -Taste erreichbar oder erstes Element einer Gruppe wird.

10.4.3 Benachrichtigungscode

Ein Schieberegler sendet keine Benachrichtigungscode im eigentlichen Sinn, also keine *WM_COMMAND*-Nachricht mit dem Code in *wParam*. Vielmehr sendet ein horizontaler Schieberegler die Nachricht *WM_HSCROLL* und ein vertikaler *WM_VSCROLL*.

Um diese Nachrichten auf eigene Methoden umzuleiten, müssen die Message-Map-Einträge *ON_WM_HSCROLL* und *ON_WM_VSCROLL* verwendet werden. Sie führen dazu, daß die Methoden *OnHScroll* und *OnVScroll* der Dialogboxklasse aufgerufen werden:

```
afx_msg void OnHScroll(UINT nSBCode, UINT nPos,  
                      CScrollBar* pScrollBar);  
afx_msg void OnVScroll(UINT nSBCode, UINT nPos,  
                      CScrollBar* pScrollBar);
```

Der wichtigste Parameter beider Methoden ist `nSBCode`, denn er gibt an, welches Element eines Schiebereglers der Benutzer bedient hat. `nSBCode` kann die folgenden Werte annehmen:

nSBCode	Benutzeraktion
SB_LINEUP	nach dem Drücken des oberen (linken) Buttons
SB_LINEDOWN	nach dem Drücken des unteren (rechten) Buttons
SB_PAGEUP	nach dem Drücken der Fläche zwischen oberem (linkem) Button und Schieber
SB_PAGEDOWN	nach dem Drücken der Fläche zwischen unterem (rechtem) Button und Schieber
SB_THUMBPOSITION	nach dem Bewegen des Schiebers
SB_THUMBTRACK	während der Bewegung des Schiebers

Tabelle 10.15: Werte von `nSBCode`

Falls `nSBCode` gleich `SB_THUMBPOSITION` oder `SB_THUMBTRACK` ist, gibt der zweite Parameter `nPos` die Position des Schiebers an. Die möglichen Werte liegen dabei innerhalb der Grenzen, die mit Hilfe der Methode `SetScrollRange` festgelegt wurden.

Der letzte Parameter, `pScrollBar`, enthält einen Zeiger auf ein Objekt vom Typ `CScrollBar`, das den Schieberegler repräsentiert. Dieser dient der Unterscheidung, falls mehr als ein Schieberegler eines Typs in der Dialogbox vorhanden ist, und kann unmittelbar weiterverwendet werden, um die Methoden der Klasse `CScrollBar` darauf anzuwenden.

Im Dialog von HEXE wird der vertikale Schieberegler benötigt, um Zahlenwerte zwischen 0 und 255 in analoger Form einzugeben:

```
void CHEXEDlg::OnVScroll(UINT nSBCode,UINT nPos,CScrollBar*
pScrollBar)
{
    switch (nSBCode)
    {
        case SB_LINEDOWN :    --areg;
                               break;
        case SB_PAGEDOWN :    areg -= nbase;
                               break;
        case SB_LINEUP :      ++areg;
                               break;
        case SB_PAGEUP :      areg += nbase;
                               break;
        case SB_THUMBPOSITION :
```

```

        case SB_THUMBTRACK :    areg = 255-nPos;
                                break;
    }
    // Positionierung prüfen
    if (areg < 0)
        areg = 0;
    if (areg > 255)
        areg = 255;;
    UpdateCalculatorDisplay();
}

...
BEGIN_MESSAGE_MAP(CHEXEDlg, CDialog)

    ...
    ON_WM_VSCROLL()

    ...
END_MESSAGE_MAP()

```

Listing: Behandlung von Nachrichten einer Bildlaufleiste

Jede Benutzermanipulation des Schiebereglers führt daher zu einem Aufruf der Methode *OnVScroll* und zu einer Änderung des Registers *areg*.

Bei der Verwendung des Parameters *nPos*, der die Position des Schiebers anzeigt, ist Vorsicht geboten. Der Wertebereich des Schiebers steigt immer von seiner oberen Stellung hin zu seiner unteren. In HEXE entspricht der Wert 0 der oberen Schieberposition, während die untere den Wert 255 repräsentiert. Aus diesem Grund ist es in HEXE erforderlich, bei der Verarbeitung *nPos* von 255 zu subtrahieren, denn die untere Schieberposition soll ja der Eingabe einer 0 entsprechen.

10.4.4 Die Klasse *CScrollBar*

SetScrollRange und *GetScrollRange*

Mit der Methode *SetScrollRange* wird der Wertebereich des Schiebereglers gesetzt, mit *GetScrollRange* wird er abgefragt:

```

class CScrollBar:
    void SetScrollRange( int nMinPos,
                        int nMaxPos,
                        BOOL bRedraw = TRUE );
    void GetScrollRange( LPINT lpMinPos,
                        LPINT lpMaxPos );

```

SetScrollRange erwartet in *nMinPos* die untere und in *nMaxPos* die obere Grenze des Wertebereichs; beide Grenzen gehören dazu. Falls *bRedraw* TRUE ist, wird der Schieberegler neu gezeichnet, um die Veränderungen sichtbar zu machen, andernfalls nicht. In HEXE wird der Wertebereich des Schiebereglers beim Initialisieren des Dialogs eingestellt:

```
BOOL CHEXEDlg::OnInitDialog()
{
    ...
    m_scrZeichenWert.SetScrollRange(0,255,FALSE);
    ...
    return TRUE;
}
```

Listing: Festlegen des Wertebereichs

GetScrollRange erwartet zwei Zeiger *lpMinPos* und *lpMaxPos* auf *int*-Variablen, in denen die Grenzen des Wertebereichs zurückgegeben werden.

SetScrollPos und *GetScrollPos*

Nachdem der Wertebereich des Schiebers festgelegt wurde, kann mit Hilfe der Methode *SetScrollPos* seine Position verändert werden. Abfragen läßt sich die aktuelle Position des Schiebers mit Hilfe der Methode *GetScrollPos*:

```
class CScrollBar:
    int SetScrollPos( int nPos,
                     BOOL bRedraw = TRUE );
    int GetScrollPos() const;
```

SetScrollPos erwartet die neue Position des Schiebers in *nPos*, dessen Inhalt innerhalb des für den Schieberegler gültigen Wertebereichs liegen muß. Falls *bRedraw* TRUE ist, wird der Schieberegler neu gezeichnet, um die Veränderung sichtbar zu machen, andernfalls nicht. Der Rückgabewert der Methode ist die bisherige Position des Schiebers.

In HEXE wird die Position des Schiebers immer dann gesetzt, wenn auch die Zahlendisplays neu geschrieben werden:

```
void CHEXEDlg::UpdateCalculatorDisplay()
{
    ...
    //---Rollbalken positionieren---
    if (areg < 0 )
        m_scrZeichenWert.SetScrollPos(255, TRUE);
    else
```

```

    if (areg > 255)
        m_scrZeichenWert.SetScrollPos(0, TRUE);
    else
        m_scrZeichenWert.SetScrollPos(255 - areg, TRUE);
}

```

Listing: Programmgesteuertes Setzen des Schiebers

GetScrollPos liefert die augenblickliche Position des Schiebereglers.

10.5 Regler (Slider)

Mit Hilfe der Regler-Steuerung lassen sich Schieberegler nachbilden, die diesen auch sehr ähnlich sehen. Die im vergangenen Abschnitt behandelten Bildlaufleisten, die auch als Schieberegler bezeichnet und genutzt werden, erinnern mehr an die Bildlaufleisten eines Fensters.

Die neuen Schieberegler haben ein völlig neues Outfit und sind in einigen Bereichen konfigurierbar. Die Funktion der Eingabe quasi-analoger Werte bleibt erhalten. Sie eignen sich dazu, einen Wert in kurzer Zeit näherungsweise einzustellen. Ein ideales Einsatzgebiet ist der Mixer einer Soundkarte.

10.5.1 Attribute

Die bereits erläuterten Basic Styles *Sichtbar*, *Gruppe*, *Deaktiviert* und *Tabstop* sind auch diesen Schieberegeln eigen. Die Wirkungen der Basic Styles auf die Steuerungen sind ebenfalls gleich. Sie besitzen auch einige der erweiterten Formate.

Eine Reihe weiterer Attribute bestimmt das Aussehen und Verhalten des Schiebereglers. Sie sind in Tabelle 10.16 zusammengefaßt.

Dialogeditor	Dialogboxschablone	Bedeutung
Orientierung (Orientation) Horizontal Vertical	Ressource-Typ: CONTROL Style: <i>TBS_HORZ</i> <i>TBS_VERT</i>	Legt die Ausrichtung des Schiebereglers fest: horizontal oder vertikal.
Punkt Oben/Links Unten/Rechts Beide (Point) Top/Left Bottom/Right Both	Ressource-Typ: CONTROL Style: <i>TBS_TOP/TBS_LEFT</i> <i>TBS_BOTTOM/TBS_RIGHT</i> <i>TBS_BOTH</i>	Gibt an, wo die Skaleneinteilung des Schiebereglers angezeigt werden soll: bei horizontalem Schieberegler oben oder unten und bei vertikalem links oder rechts. Des Weiteren kann die Markierung auf beiden Seiten gesetzt werden.

Dialogeditor	Dialogboxschablone	Bedeutung
Rand (Border)	Ressource-Typ: CONTROL Style: <i>TBS_BORDER</i>	Um den Regler wird ein Rand gezeichnet.
Auswahl aktivieren (Enable Selrange)	Ressource-Typ: CONTROL Style: <i>TBS_ENABLESELRANGE</i>	Auf dem Regler wird ein Auswahlbereich dargestellt.
Teilstriche Auto-Teilstriche (Tick marks Autoticks)	Ressource-Typ: CONTROL Style: <i>TBS_NOTICKS</i> <i>TBS_AUTOTICKS</i>	Mit <i>Tick marks</i> wird festgelegt, ob eine Einteilung vorhanden ist oder nicht. <i>Autoticks</i> legt fest, daß diese automatisch entsprechend dem Inkrement platziert wird.

Tabelle 10.16: Attribute des Sliders

10.5.2 Benachrichtigungscodes

Der Regler sendet die bereits bekannte Nachricht `WM_HSCROLL`. In der Message-Map-Funktion *OnHScroll*, die überlagert werden muß, kann darauf reagiert werden. Zu beachten ist dabei, daß, wenn mehrere Regler in einem Dialog existieren, festgestellt werden muß, welche Steuerung die Nachricht gesendet hat.

```
void CControlsDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar*
pScrollBar)
{
    switch(pScrollBar->GetDlgCtrlID())
    {
        case IDC_SLIDER1: ...
        case IDC_SLIDER2: ...
    }
    CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
}
```

Listing: Nachrichtenbehandlung für mehrere Regler

Da die Regler-Steuerung auch mit der Tastatur bedienbar ist, werden zusätzliche `TB_`-Nachrichten entsprechend der gedrückten Taste verschickt.

10.5.3 Die Klasse *CSliderCtrl*

SetRange und *GetRange*

Mit der Methode *SetRange* wird der Wertebereich gesetzt, in dem sich der Schieberegler bewegen kann. Mit Hilfe von *GetRange* kann dieser ausgelesen werden.

```
class CSliderCtrl:
    void SetRange( int nMin, int nMax,
                  BOOL bRedraw = FALSE
    void GetRange( int& nMin, int& nMax ) const;
    void SetRangeMin( int nMin,
                     BOOL bRedraw = FALSE );
    void SetRangeMax( int nMax,
                     BOOL bRedraw = FALSE );
    int GetRangeMin( ) const;
    int GetRangeMax( ) const;
```

Um die Minimum- und Maximum-Werte auch einzeln lesen und ändern zu können, existieren die Methoden *SetRangeMin*, *SetRangeMax*, *GetRangeMin* und *GetRangeMax*. Der Parameter *bRedraw* gibt an, ob die Steuerung neu gezeichnet werden soll oder nicht. Ansonsten werden für den Wertebereich Integer-Werte benötigt.

GetPos und SetPos

Diese beiden Methoden erlauben es, den Knopf des Schiebereglers auf eine bestimmte Position innerhalb des Wertebereichs zu setzen bzw. die aktuelle Position auszulesen.

```
class CSliderCtrl:
    void SetPos ( int nPos );
    int GetPos ( ) const;
```

Die aktuelle Position des Schiebers wird als Integer-Wert angegeben, der innerhalb des Wertebereichs liegt.

SetTic und SetTicFreq

Die Markierungen der Skaleneinteilung des Reglers werden als *Teilstriche* oder *Ticks* bezeichnet. Diese können entweder von Hand auf bestimmte Werte oder in regelmäßigen Abständen gesetzt werden. *SetTic* setzt eine Markierung auf die übergebene Position. Um diese Methode anwenden zu können, müssen in der Ressource Teilstriche aktiviert und Auto-Teilstriche deaktiviert sein. Bei *SetTicFreq* müssen beide Styles aktiviert sein. Der Parameter *nFreq* bestimmt, nach jeweils wieviel Einheiten eine Markierung gesetzt wird. Der Wert ist standardmäßig auf eins gesetzt.

```
class CSliderCtrl:
    BOOL SetTic( int nTic );
    void SetTicFreq( int nFreq );
```

Neben diesen beiden Methoden zum Setzen von Markierungen gibt es auch noch zwei zum Lesen der Position (*GetTic*, *GetTicFreq*) sowie eine zum Löschen von Tabulatoren (*ClearTics*).

Auf der mitgelieferten CD ist das Beispielpogramm CONTROLS enthalten, welches auch die Anwendung eines Reglers zeigt.



10.6 Strukturansicht (Tree Control)

Die Strukturansicht (auch Baumansicht oder Tree Control) ist eine interessante Steuerung. Mit ihr lassen sich hierarchisch strukturierte Informationen darstellen. Diese Darstellungsweise hat in breiter Form Einzug gehalten. Die Darstellung der Datei- und Verzeichnisstruktur im Explorer ist nur ein Anwendungsfall, auch für von Hilfe-Informationen, wie zum Beispiel die Visual C++-Handbücher, ist sie geeignet.

Jedes Element des Baumes ist durch einen Bezeichner (Label) und optional durch ein Bitmap beschrieben. Strukturen, die unter einem Element liegen, können expandiert oder zusammengeklappt werden, damit läßt sich die Übersichtlichkeit beträchtlich erhöhen. Zur Darstellung der Struktur lassen sich Linien und kleine Knöpfe hinzufügen, die anzeigen, ob eine Struktur noch weitere Unterelemente enthält ☐ oder die Struktur bereits expandiert ist ☐. Ist keines der Zeichen vorhanden, so existieren darunter keine weiteren Elemente mehr.

10.6.1 Attribute

Außer den Standard- und erweiterten Eigenschaften, die auf die bereits beschriebene Weise wirken, besitzt eine Strukturansicht noch weitere Eigenschaften, die in Tabelle 10.17 zusammengefaßt sind.

Dialogeditor	Dialogboxschablone	Bedeutung
Mit Schaltflächen (Has Buttons)	Ressource-Typ: CONTROL Style: TVS_HASBUTTONS	Legt fest, daß die ☐ - und ☐ -Zeichen dargestellt werden.
Mit Linien (Has Lines)	Ressource-Typ: CONTROL Style: TVS_HASLINES	Die Darstellung der Struktur erfolgt zusätzlich mit Linien.
Rand (Border)	Ressource-Typ: CONTROL Style: WS_BORDER	Erzeugt eine Umrandung um das Tree Control.

Tabelle 10.17: Attribute der Strukturansicht

Dialogeditor	Dialogboxschablone	Bedeutung
Linien am Ursprung (Lines at Root)	Ressource-Typ: CONTROL Style: <i>TVS_LINESATROOT</i>	Gibt an, daß eine Linie zur Verbindung der Elemente der ersten Ebene dargestellt wird.
Bezeichnungen bearbeiten (Edit Labels)	Ressource-Typ: CONTROL Style: <i>TVS_EDITLABELS</i>	Erlaubt das Editieren des Textes der Elemente.
Drag & Drop deaktivieren (Disable Drag Drop)	Ressource-Typ: CONTROL Style: <i>TVS_DISABLEDRAHDROP</i>	Verhindert das Senden der TVN_BEGINDRAG-Message, die das Drag & Drop einleitet.
Auswahl immer zeigen (Show Selection always)	Ressource-Typ: CONTROL Style: <i>TVS_SHOWSELALWAYS</i>	Zeigt das ausgewählte Element auch noch nach Verlieren des Eingabefokusses des Controls an.

Tabelle 10.17: Attribute der Strukturansicht

10.6.2 Benachrichtigungscode

Baumansichten senden eine Reihe weiterer Benachrichtigungscode. Diese müssen entsprechend der Funktion des Controls angewendet werden. So hat es zum Beispiel nur Sinn, auf einen Doppelklick zu reagieren (NM_DBLCLK), wenn Elemente ausgewählt werden sollen, die keine Substruktur mehr enthalten, sonst würde zusätzlich die darunterliegende Struktur expandiert bzw. zusammengeklappt werden. Auch hier empfiehlt es sich, die Nachrichten-Funktionen mit dem Klassen-Assistenten anzulegen.

Wichtig ist die Nachricht TVN_SELCHANGED, die gesendet wird, wenn der Anwender einen anderen Punkt in der Liste gewählt hat. TVN_BEGINLABELEDIT und TVN_ENDLABELEDIT sind dafür verantwortlich, das Editieren eines Labels einzuleiten bzw. zu beenden. So kann nach dem Ändern der Text ausgelesen und in einer Datenbank o.ä. aktualisiert werden.



Weitere Benachrichtigungscode sind in den Technical Notes Nr. 60 aufgeführt.

10.6.3 Die Struktur TV_INSERTSTRUCT

Verantwortlich für den Inhalt der Steuerung ist die Datenstruktur TV_INSERTSTRUCT. In ihr werden die Texte der Labels, die Hierarchie-Ebenen und andere Parameter festgelegt.

```
typedef struct _TV_INSERTSTRUCT { tvins
    HTREEITEM hParent;
    HTREEITEM hInsertAfter;
    TV_ITEM item;
} TV_INSERTSTRUCT, FAR *LPTV_INSERTSTRUCT;
```

Der Parameter `hParent` legt fest, an welcher Stelle in der Struktur das neue Element eingefügt wird. Für die oberste Ebene muß er auf `TVI_ROOT` oder `NULL` gesetzt werden. Über `hInsertAfter` wird bestimmt, wo das Element in der Liste eingefügt wird. Möglich sind `TVI_FIRST`, `TVI_LAST` und `TVI_SORT`. Die Bedeutung läßt sich leicht aus dem Namen ableiten. Für das Element selbst muß die `TV_ITEM`-Struktur ausgefüllt werden.

```
typedef struct _TV_ITEM { tvi
    UINT mask;
    HTREEITEM hItem;
    UINT state;
    UINT stateMask;
    LPSTR pszText;
    int cchTextMax;
    int iImage;
    int iSelectedImage;
    int cChildren;
    LPARAM lParam;
} TV_ITEM;
```

Hier sind nur einige Parameter von Hand zu setzen. So muß unbedingt `pszText` gefüllt werden, denn das ist der Text des Labels. Ein benutzerdefinierter 32-Bit-Wert wird in `lParam` gespeichert. Dieser kann nach der Wahl des Benutzers ausgewertet werden. Der Parameter `mask` teilt der Struktur mit, welche der gesetzten Elemente gültig sind. Die symbolischen Parameter korrespondieren mit den entsprechenden Parametern: `TVIF_TEXT`, `TVIF_PARAM`, `TVIF_IMAGE` oder `TVIF_SELECTEDIMAGE`.

Die Strukturansicht bietet auch die Möglichkeit, Bitmaps vor den einzelnen Labels anzuzeigen. Die Bitmaps werden in einem *CImageList*-Objekt gespeichert. Diese Klasse kann eine größere Anzahl von Icons oder Bitmaps, die alle die gleiche Größe haben, über einen Index verwalten. Die Parameter `iImage` und `iSelectedImage` der `TV_ITEM`-Struktur nehmen Bezug auf diese Struktur. Für den Fall, daß das Element ausgewählt ist, wird `iSelectedImage` angezeigt, ansonsten `iImage`. Das *CImageList*-Objekt wird über eine Member-Funktion des Tree Controls diesem zugeordnet.

10.6.4 Anlegen einer Struktur

Nachdem im vorhergehenden Abschnitt beschrieben wurde, welche Datenstrukturen für ein Tree Control maßgebend sind, soll nun eine derartige Struktur angelegt werden. Hat man den vorhergehenden Abschnitt gelesen, könnte man meinen, daß dies ziemlich kompliziert sei, doch dem ist nicht so. Der richtige Ansatzpunkt zum Anlegen der Struktur ist die Methode *OnInitDialog*.

```

BOOL CTreePage::OnInitDialog()
{
    ...
    TV_INSERTSTRUCT TreeCtrlItem;
    //Wurzel
    TreeCtrlItem.hParent = TVI_ROOT;
    TreeCtrlItem.hInsertAfter = TVI_LAST;
    TreeCtrlItem.item.mask = TVIF_TEXT | TVIF_PARAM;
    TreeCtrlItem.item.pszText = "MFC-CWnd";
    TreeCtrlItem.item.lParam = 0;
    HTREEITEM hTreeItem = m_Tree.InsertItem(&TreeCtrlItem);
    //Ebene 1
    TreeCtrlItem.hParent = hTreeItem;
    TreeCtrlItem.item.pszText = "CFrameWnd";
    TreeCtrlItem.item.lParam = 1;
    HTREEITEM hTreeItem1 = m_Tree.InsertItem(&TreeCtrlItem);
    TreeCtrlItem.item.pszText = "CPropertySheet";
    TreeCtrlItem.item.lParam = 5;
    m_Tree.InsertItem(&TreeCtrlItem);
    TreeCtrlItem.item.pszText = "CSplitterWnd";
    TreeCtrlItem.item.lParam = 6;
    m_Tree.InsertItem(&TreeCtrlItem);
    TreeCtrlItem.item.pszText = "CDialog";
    TreeCtrlItem.item.lParam = 2;
    HTREEITEM hTreeItem3 = m_Tree.InsertItem(&TreeCtrlItem);
    //Ebene 2/1
    TreeCtrlItem.hParent = hTreeItem1;
    TreeCtrlItem.item.pszText = "CMDIFrameWnd";
    TreeCtrlItem.item.lParam = 11;
    m_Tree.InsertItem(&TreeCtrlItem);
    TreeCtrlItem.item.pszText = "CMDIChildWnd";
    TreeCtrlItem.item.lParam = 12;
    m_Tree.InsertItem(&TreeCtrlItem);
    TreeCtrlItem.item.pszText = "CMiniChildWnd";
    TreeCtrlItem.item.lParam = 13;
    m_Tree.InsertItem(&TreeCtrlItem);
}

```

```

TreeCtrlItem.item.pszText = "COleIPFrameWnd";
TreeCtrlItem.item.lParam = 14;
m_Tree.InsertItem(&TreeCtrlItem);
//Ebene 2/2
TreeCtrlItem.hParent = hTreeItem3;
TreeCtrlItem.item.pszText = "CCommonDialog";
TreeCtrlItem.item.lParam = 21;
HTREEITEM hTreeItem4 = m_Tree.InsertItem(&TreeCtrlItem);
TreeCtrlItem.item.pszText = "CPropertyPage";
TreeCtrlItem.item.lParam = 22;
m_Tree.InsertItem(&TreeCtrlItem);
TreeCtrlItem.item.pszText = "COlePropertyPage";
TreeCtrlItem.item.lParam = 23;
m_Tree.InsertItem(&TreeCtrlItem);
//Ebene 2/1/1
TreeCtrlItem.hParent = hTreeItem4;
TreeCtrlItem.item.pszText = "CColorDialog";
TreeCtrlItem.item.lParam = 211;
m_Tree.InsertItem(&TreeCtrlItem);
TreeCtrlItem.item.pszText = "CFileDialog";
TreeCtrlItem.item.lParam = 212;
m_Tree.InsertItem(&TreeCtrlItem);
TreeCtrlItem.item.pszText = "CFindReplaceDialog";
TreeCtrlItem.item.lParam = 213;
m_Tree.InsertItem(&TreeCtrlItem);
TreeCtrlItem.item.pszText = "CFontDialog";
TreeCtrlItem.item.lParam = 214;
m_Tree.InsertItem(&TreeCtrlItem);
TreeCtrlItem.item.pszText = "CPrintDialog";
TreeCtrlItem.item.lParam = 215;
m_Tree.InsertItem(&TreeCtrlItem);
m_Tree.Expand(hTreeItem,TVE_EXPAND);
return TRUE;
}

```

Listing: Aufbau einer Baumstruktur

Zuerst wird ein Datenelement vom Typ `TV_INSERTSTRUCT` angelegt. Diese Struktur wird anschließend mit den benötigten Werten gefüllt. In der ersten Ebene (der Wurzel) wird `hParent` auf `TVI_ROOT` gesetzt, in tieferliegenden Ebenen wird der Handle der vorhergehenden Ebene benutzt. Dieser Handle wird beim Aufruf von *InsertItem* zurückgegeben. Die Strukturelemente `hInsertAfter` und `mask` müssen nur einmal gesetzt werden, da sie sich nicht ändern, und immer die gleiche Variable benutzt wird. Hin-

gegen müssen in jeder Stufe für jedes Element `pszText` und `lParam` entsprechend belegt werden. Nachdem alle Elemente eingefügt wurden, wird noch die erste Ebene mittels einer `Expand`-Member-Funktion erweitert.

10.6.5 Die Klasse *CTreeCtrl*

GetSelectedItem

Die Methode *GetSelectedItem* liefert einen Zeiger auf das aktuell selektierte Element der Steuerung. Dieser Zeiger verweist auf eine `TV_ITEM`-Struktur. Sie kann an weitere Funktionen übergeben werden, um die Informationen zu lesen oder zu ändern.

```
class CTreeCtrl:
    HTREEITEM GetSelectedItem( );
```

GetItemData und *GetItemText*

GetItemData liefert den benutzerdefinierten 32-Bit-Wert als `DWORD` des angegebenen Elements zurück. Analog dazu liefert *GetItemText* den Text des Labels. Mittels der Gegenstücke *SetItemData* und *SetItemText* lassen sich diese auch neu setzen.

```
class CTreeCtrl:
    DWORD GetItemData( HTREEITEM hItem ) const;
    BOOL SetItemData( HTREEITEM hItem,
                     DWORD dwData );
    CString GetItemText( HTREEITEM hItem ) const;
    BOOL SetItemText( HTREEITEM hItem,
                     LPCTSTR lpszItem );
```

InsertItem, *DeleteItem* und *SelectItem*

Mittels der Member-Funktion *InsertItem* lassen sich neue Elemente in die Steuerung einfügen. Dafür existieren eine Reihe von Funktionen, die je nach Bedarf angewendet werden können. *DeleteItem* löscht ein Element aus der Steuerung, und *SelectItem* wählt eines aus.

```
class CTreeCtrl:
    HTREEITEM InsertItem( LPTV_INSERTSTRUCT
                        lpInsertStruct );
    BOOL DeleteItem( HTREEITEM hItem );
    HTREEITEM SelectItem( HTREEITEM hItem );
```

Expand

Mit Hilfe von *Expand* wird es möglich, programmgesteuert einen Zweig der Baumstruktur zu expandieren oder zusammenzufassen. Der entsprechende Parameter wird in `nCode` übergeben.


```
class CTreeCtrl:
    BOOL Expand( HTREEITEM hItem, UINT nCode );
```

Folgende Parameter sind für nCode möglich:

- ▼ TVE_COLLAPSE
klappt die Struktur zusammen.
- ▼ TVE_COLLAPSERESET
klappt die Struktur zusammen und entfernt darunterliegende Elemente.
- ▼ TVE_EXPAND
erweitert die Struktur vollständig.
- ▼ TVE_TOGGLE
schaltet um zwischen Erweitern und Zusammenklappen, je nachdem, in welchem sich der Zweig gerade befindet.

Auf der mitgelieferten CD ist das Beispielprogramm CONTROLS enthalten, welches die Anwendung einer Strukturansicht zeigt.



10.7 Drehfeld (Spin Button Control)

Das Drehfeld ist eine Steuerung, die meist in Verbindung mit einem Eingabefeld benutzt wird. Sie besteht aus zwei kleinen Pfeilspitzen als Buttons, die dazu verwendet werden, einen Wert zu inkrementieren bzw. zu dekrementieren. Der Wert ist die aktuelle Position des Controls. Das mit dem Spin Button Control verbundene Eingabefeld wird *Buddy window* genannt. Die Steuerung wirkt in Verbindung mit dem Buddy-Fenster oft wie ein einziges Control. Das wird durch einen bestimmten Style erreicht, der die automatische Positionierung des Drehfeldes übernimmt. Zu beachten ist, daß als Buddy-Fenster die Steuerung verwendet wird, die in der Tab-Ordnung unmittelbar vor dem Drehfeld liegt.

Der vorgelegte Wertebereich des Controls geht von 100 als Minimum bis 0 als Maximum. Der aktuelle Wert wird beim Drücken des rechten oberen Buttons verringert. Ohne Buddy window wird das Control zum Beispiel dazu verwendet, in einer Registerkarte die Reiter zu scrollen (im Developer Studio z. B. in PROJEKT|EINSTELLUNGEN).

10.7.1 Attribute

Außer den Standard-Eigenschaften und den erweiterten Formaten besitzt ein Drehfeld weitere Attribute, die in Tabelle 10.18 zusammengefaßt sind.

Dialogeditor	Dialogboxschablone	Bedeutung
Orientierung (Orientation) Horizontal Vertical	Ressource-Typ: CONTROL Style: <i>UDS_HORZ</i>	Legt die Ausrichtung der Steuerung fest: horizontal oder vertikal angeordnete Knöpfe (vertikal ist Standard).
Ausrichtung Links Rechts Nicht zugewiesen (Alignment) Left Right Unattached	Ressource-Typ: CONTROL Style: <i>UDS_ALIGNLEFT</i> <i>UDS_ALIGNRIGHT</i>	Gibt an, wo sich die Steuerung im Buddy-Fenster befindet: links oder rechts. Ist keiner dieser Werte gesetzt, ist das Control frei positionierbar.
Auto buddy	Ressource-Typ: CONTROL Style: <i>UDS_AUTOBUDDY</i>	Legt fest, daß die in der Tab-Ordnung davor stehende Steuerung als Buddy-Fenster benutzt wird.
Buddy Ganzzahl setzen Keine Tausender (Set buddy integer) (No thousands)	Ressource-Typ: CONTROL Style: <i>UDS_SETBUDDYINT</i> <i>UDS_NOTHOUSANDS</i>	Buddy Ganzzahl ermöglicht es, den Wert des Buddy-Fensters zu inkrementieren oder zu dekrementieren. Ist <i>UDS_NOTHOUSANDS</i> gesetzt, so entfällt das Tausender-Trennzeichen.
Umbruch (Wrap)	Ressource-Typ: CONTROL Style: <i>UDS_WRAP</i>	Dieser Style legt fest, daß einmal am Ende des Wertebereichs angekommen, zum Anfang zurückgesprungen wird und umgekehrt.
Pfeiltasten (Arrow keys)	Ressource-Typ: CONTROL Style: <i>UDS_ARROWKEYS</i>	Mit Hilfe dieses Styles läßt sich die Steuerung auch mit den Cursor-tasten bedienen.

Tabelle 10.18: Attribute des Drehfeldes

10.7.2 Benachrichtigungscode

Da ein Drehfeld meist mit seinem Buddy verbunden ist, müssen zur Veränderung von Integer-Werten keinerlei Programmierungen vorgenommen werden. Die Darstellung eines Double-Wertes erfordert aber auch nur wenig Aufwand.

Die Steuerung sendet die Nachricht *WM_VSCROLL*, die überlagert werden muß. Auch hier muß bei Vorhandensein mehrerer dieser Steuerungen unterschieden werden, von welchem Control die Nachricht versandt wurde (siehe Regler-Steuerung).

10.7.3 Die Klasse *CSpinButtonCtrl*

SetRange und GetRange

Mit Hilfe der Methode *SetRange* wird der Wertebereich der Steuerung festgelegt. Es müssen Integer-Werte angegeben werden, die den Minimum- und Maximum-Wert repräsentieren. Die Methode *GetRange* ermöglicht das Abfragen des Wertebereichs.

```
class CSpinButtonCtrl:
    void SetRange( int nLower, int nUpper );
    DWORD GetRange( ) const;
    void GetRange(int &lower, int& upper) const;
```

GetPos und SetPos

Diese beiden Methoden ermöglichen das Setzen bzw. das Abfragen der aktuellen Position der Steuerung. Der zum Setzen benutzte Wert muß im Wertebereich der Steuerung liegen.

```
class CSpinButtonCtrl:
    int SetPos ( int nPos );
    int GetPos ( ) const;
```

GetBuddy und SetBuddy

Um eine Zuordnung der Steuerung zu einem Buddy-Fenster während der Laufzeit des Programms zu ändern, wird die Methode *SetBuddy* verwendet. Ihr muß ein *CWnd*-Zeiger des zu setzenden Buddys übergeben werden. Als Rückgabewert wird ein Zeiger auf das aktuelle Buddy window geliefert. *GetBuddy* ermöglicht das Abfragen des Buddy-Fensters.

```
class CSpinButtonCtrl:
    CWnd* SetBuddy( CWnd* pWndBuddy );
    CWnd* GetBuddy( ) const;
```

Auf der mitgelieferten CD ist das Beispielprogramm **CONTROLS** enthalten, welches unter anderem die Anwendung eines Drehfeldes zeigt.



10.8 Zugriffstaste (Hotkey Control)

Die Zugriffstasten-Steuerung dient dazu, dem Benutzer die Möglichkeit zu eröffnen, für eine Funktion in einem eigenen Programm eine Zugriffstaste zu definieren. Dabei hat diese Steuerung lediglich die Aufgabe, die Eingaben des Benutzers anzuzeigen. Somit kann dieser die gewünschte Tastenkombination einfach drücken und bekommt diese im Klartext angezeigt. Damit wird verhindert, daß versehentlich eine falsche Taste gedrückt wird.

10.8.1 Attribute

Außer den allgemeinen und den erweiterten Formaten besitzt ein Hotkey Control keine weiteren Attribute.

10.8.2 Benachrichtigungscodes

Die Steuerung sendet nur dann die Nachricht `NM_OUTOFMEMORY`, wenn eine Aktion der Steuerung aus Speichermangel nicht ausgeführt werden kann.

10.8.3 Die Klasse *CHotKeyCtrl*

SetHotKey

Mit Hilfe der Methode *SetHotKey* läßt sich die Steuerung mit einem Vorgabewert belegen. Das ist zum Beispiel dann wichtig, wenn ein bestehender Hotkey angezeigt werden soll. Der Parameter `wVirtualKeyCode` enthält den Wert der Taste. In `wModifiers` werden die Zusatztasten wie *Umschalt*, *Steuerung* oder *Alt* abgelegt. Mehrere dieser Zusatztasten werden über eine ODER-Verknüpfung angegeben. Soll hingegen keine benutzt werden, wird einfach `NULL` übergeben. Die folgenden Werte sind für `wModifiers` möglich:

- ▼ `HOTKEYF_SHIFT`
- ▼ `HOTKEYF_ALT`
- ▼ `HOTKEYF_CONTROL`
- ▼ `HOTKEYF_EXT`

```
class CHotKeyCtrl:  
    void SetHotKey( WORD wVirtualKeyCode,  
                  WORD wModifiers );
```

GetHotKey

Zum Auslesen der Steuerung dient die Methode *GetHotKey*. Sie kann den Wert der Taste entweder als `DWORD` oder wie beim Setzen der Tastenkombination getrennt zurückgegeben.

```
class CHotKeyCtrl:  
    DWORD GetHotKey( ) const;  
    void GetHotKey( WORD &wVirtualKeyCode,  
                  WORD &wModifiers ) const;
```

SetRules

Mit Hilfe der Methode *SetRules* ist es möglich, nicht zulässige Tasten für die Eingabe eines Hotkeys festzulegen. Das betrifft die Kombinationen der Tasten *Umschalt*, *Steuerung* und *Alt* sowie das Drücken keiner Taste. Wird

eine verbotene Taste gedrückt, so wird an Stelle dieser die in `wModifiers` angegebene Taste angezeigt. Es kann aber auch eine Kombination aus mehreren ungültigen Tasten über eine ODER-Verknüpfung in `wInvalidComb` definiert werden.

```
class CHotKeyCtrl:
    void SetRules( WORD wInvalidComb,
                  WORD wModifiers );
```

Auf der mitgelieferten CD ist das Beispielprogramm **CONTROLS** enthalten, das exemplarisch die Anwendung einer Zugriffstaste demonstriert.



10.9 Statusanzeige (Progress Control)

Das *Progress Control* wird als Statusanzeige bezeichnet, wobei der Begriff Fortschrittsanzeige treffender wäre. Sie wird immer dann eingesetzt, wenn eine Bearbeitung oder ein Prozeß längere Zeit läuft. Somit erhält der Benutzer die Möglichkeit, den aktuellen Fortgang der Arbeiten zu verfolgen. Des weiteren wird er darüber informiert, daß die Bearbeitung noch läuft und der Rechner nicht abgestürzt ist.

Die Statusanzeige besitzt außer den Standard- und erweiterten Eigenschaften nur noch das Attribut *Rand*, um eine Umrandung der Steuerung zu erreichen. Da sie nicht auf Ereignisse reagieren muß, sendet sie auch keine Benachrichtigungs-codes. Basiswerte eines *Progress Controls* sind der Wertebereich, der standardmäßig zwischen 0 und 100 liegt, und die aktuelle Position. Der Wertebereich wird mit der Methode *SetRange* und die aktuelle Position mit *SetPos* bestimmt.

Es gibt drei verschiedene Möglichkeiten, die aktuelle Position des Controls zu ändern:

- ▼ durch schrittweises Inkrementieren um einen festen Wert
- ▼ über einen Offset, der auch negativ sein kann
- ▼ durch Setzen der aktuellen Position

Im ersten Fall wird über die Methode *SetStep* ein Inkrement festgelegt, das auch negativ sein kann. Mit Hilfe von *StepIt* wird die aktuelle Position dann bei jedem Aufruf um das Inkrement verändert.

Die Methode *OffsetPos* setzt, ausgehend von der aktuellen Position, diese um den angegebenen Offset neu. Durch negative Werte kann auch ein Countdown erreicht werden.

Die dritte Variante bedient sich der Methode *SetPos*, die die aktuelle Position auf einen festen Wert setzt. Hierbei ist es günstig, eine Variable mitzuführen, die im Wertebereich der Steuerung liegt und dem prozentualen Fortschritt entspricht. Diese Variable wird dann durch die Bearbeitungs-

funktion gesetzt und zum Beispiel von einer timergesteuerten Funktion ausgelesen. In dieser Funktion wird dann die Fortschrittsanzeige über *SetPos* aktualisiert.

```
class CProgressCtrl:
    void SetRange( int nLower, int nUpper );
    int SetStep( int nStep );
    int StepIt( );
    int OffsetPos( int nPos );
    int SetPos( int nPos );
```



Auf der mitgelieferten CD ist das Beispielprogramm CONTROLS enthalten, welches auch die Anwendung einer Fortschrittsanzeige zeigt.

10.10 Eigenschaftenfenster (Property Sheets)

Auch bei dieser Art der Steuerung kann man aus der deutschen Übersetzung wieder alles ableiten ... Sie werden auch als *Register-Dialogfelder* bezeichnet, was nicht mit der *Registerkarten*-Steuerung (siehe unten) verwechselt werden darf, die diesen sehr ähnlich ist.

Property Sheets sind mehrseitige Dialoge. Sie wurden eingeführt, um die Informationsfülle eines Dialogs zu verringern und dem Nutzer eine bessere Übersicht zu verschaffen. So wird zum Beispiel innerhalb der Option PROJEKT|EINSTELLUNGEN eine *Registerkarten*-Steuerung verwendet, um überhaupt noch gezielt eine Einstellung dieses Dialogs finden zu können. Eine *Registerkarten*-Steuerung hat die gleiche Funktion wie ein *Property Sheet*. Während ein *Property Sheet* den Dialog mit einigen Standardschaltflächen selbst bildet, ist eine *Registerkarte* Bestandteil eines Dialogs wie jede andere Steuerung auch.

Beide Steuerungen arbeiten nach dem Prinzip von Karteikarten, die über Reiter ausgewählt werden. Die Reiter sind beschriftet und ermöglichen so eine Vorauswahl. Auf jeder dieser Seiten – *Property Pages* – sind dann weitere Steuerungen enthalten. Eine abgewandelte Form von *Property Sheets* sind viele der Assistenten. Dabei werden die einzelnen Dialogseiten nicht über Reiter angewählt, sondern nacheinander in einer festgelegten Reihenfolge über die Schaltflächen WEITER bzw. ZURÜCK. Diese Schaltflächen werden automatisch in diesem Modus eingefügt.

10.10.1 Anlegen der Dialog-Ressourcen

Jede Seite eines *Property Sheets* wird als einzelne Dialog-Ressource erstellt. Dazu wird mit dem Dialogeditor ein neuer Dialog angelegt. Die Buttons OK und ABBRUCH werden entfernt, da diese global im Eigenschaftenfenster zur Verfügung stehen. Außerdem ist es erforderlich, einige Styles des Dialogs zu ändern:

- ▼ System-Menü deaktivieren
- ▼ Format: untergeordnet (Style: Child)
- ▼ Rand: dünn (Border: Thin)

Der Titel des Dialogs (Caption) wird später im Property Sheet als Beschriftung für den Reiter benutzt.

Anschließend werden die benötigten Steuerungen in den Dialog eingefügt.

10.10.2 Anlegen der Property Pages

Nachdem die Ressource erstellt worden ist, wird dafür eine Klasse angelegt. Es muß nur der Klassen-Assistent aufgerufen werden (ANSICHT|KLASSEN-ASSISTENT oder `Strg W`). Nach der Bestätigung, eine neue Klasse für die Dialog-Ressource anzulegen, wird das Dialogfenster zum Anlegen einer neuen Klasse aufgerufen (siehe Abbildung 10.3). Dort wird der *Name* der neuen Klasse festgelegt und die *Basisklasse*, von der sie abgeleitet wird. Als Typ der Basisklasse wird aus der Listbox *CPropertyPage* gewählt. Die ID der Dialog-Ressource wird automatisch vorgeschlagen. Nach dem Drücken von OK wird die Klasse in den angegebenen Dateien erzeugt. Dabei empfiehlt es sich, alle Seiten des Property Sheets in eine Quelltext- und Header-Datei zu legen. Das erspart *Include*-Anweisungen in Dateien und erleichtert die Arbeit, zumindest wenn der Umfang der einzelnen Seiten nicht allzu groß ist.

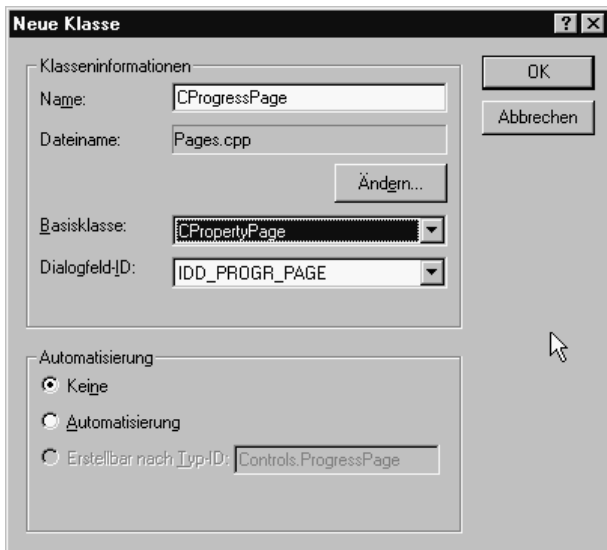


Abbildung 10.3: Anlegen einer Klasse für eine Property Page

Jetzt werden für jede Dialogseite Member-Variablen und Nachrichten-Funktionen nach Bedarf eingefügt. Jede Dialogseite kann zum Beispiel ihre eigene *OnInitDialog*-Methode aufrufen.

Die Bearbeitung von Nachrichten erfolgt analog zu einem normalen Dialog.

10.10.3 Anlegen des Property Sheets

Nachdem für alle Seiten des Dialogs die Ressourcen mit den zugehörigen Klassen angelegt wurden, muß noch eine von *CPropertySheet* abgeleitete Klasse erzeugt werden, die für die Darstellung des Eigenschaftensfensters verantwortlich ist. Dazu wird wiederum der Klassen-Assistent aufgerufen und die Schaltfläche **KLASSE HINZUFÜGEN|NEU** betätigt. Der Name der zu erzeugenden Klasse muß eingegeben und der Typ der Basisklasse – *CPropertySheet* – gewählt werden. Auch diese Klasse kann in die Quelltext-Datei der Dialogseiten eingefügt werden. Damit sind alle benötigten Ressourcen und Klassen erzeugt. Diese müssen jetzt nur noch verknüpft werden.

Das Zusammenfügen der Dialogseiten zu einem Property Sheet kann an verschiedenen Stellen erfolgen:

- ▼ im Konstruktor der Klasse
- ▼ in der Methode, die das Property Sheet anlegt und aktiviert

Die zweite Variante ermöglicht es auf einfache Weise, die Anzahl und Anordnung der Seiten während der Laufzeit des Programms festzulegen, während in der ersten Variante eine feste Zuordnung erfolgt. Das Hinzufügen von Property Pages erfolgt mit der Methode *AddPage*. Dabei muß ein Zeiger auf ein *Property-Page*-Objekt übergeben werden.

Die einzelnen Seiten werden im Deklarationsteil des *Property Sheets* als Member-Variablen eingefügt. Damit wird jeweils eine Instanz der Klasse der *Property Pages* angelegt.

```
class CPropertySheet:
    void AddPage( CPropertyPage *pPage );
    virtual int DoModal();
```



Da zwei verschiedene Konstruktoren für das Property Sheet erzeugt werden, muß das Hinzufügen der Seiten auch in beiden erfolgen.

```
CWinControls::CWinControls(UINT nIDCaption, CWnd* pParentWnd, UINT
iSelectPage)
    :CPropertySheet(nIDCaption, pParentWnd, iSelectPage)
{
    //Button Übernehmen entfernen
    m_psh.dwFlags |= PSH_NOAPPLYNOW;
```



```
//Seiten hinzufügen
AddPage(&p1);
AddPage(&p2);
AddPage(&p3);
AddPage(&p4);
}
```

Listing: Hinzufügen von Seiten zu einem Eigenschaftfenster

Zusätzlich zu den Methoden besitzt ein *Property-Sheet*-Objekt noch einen öffentlichen Daten-Member `m_psh`, der eine Struktur ist. Diese Struktur erlaubt den Zugriff auf einige weitere Parameter des Objekts und ermöglicht so das Verändern des Layouts des Property Sheets. In Tabelle 10.19 sind einige der Parameter von `dwFlags` aufgeführt.

dwFlags	Bedeutung
PSH_DEFAULT	benutzt die Standardeinstellungen des Property Sheets.
PSH_HASHELP	dient zum Einschalten des Hilfe-Buttons des Property Sheets, ist aber nur erforderlich, wenn die einzelnen Seiten die Hilfe deaktiviert haben, sie aber dennoch benötigt wird.
PSH_MULTILINE TABS	erlaubt mehrzeiligen Text in den Reitern der Seiten.
PSH_NOAPLYNOW	entfernt den Button ÜBERNEHMEN aus dem Property Sheet.
PSH_WIZARD	verändert das Property Sheet so, daß immer nur eine Seite sichtbar ist, aber Buttons WEITER und ZURÜCK eingefügt werden, die das Blättern von Seite zu Seite ermöglichen und somit einen Ablauf fest vorgeben (wie zum Beispiel beim Anlegen eines neuen Projekts mit dem AppWizard).

Tabelle 10.19: Flags eines Property Sheets

Die zweite Variante zum Erzeugen eines Property Sheets fügt die Seiten nicht im Konstruktor, sondern nach dem Anlegen der Instanz hinzu. Als Beispiel dient der Aufruf des Dialogs durch Drücken eines Buttons in der Methode `OnProp`.

```
void CControlsDlg::OnProp()
{
    CWinControls m_Sheet(IDS_TITLE);
    CSliderPage p1;
    CSpinPage p2;
    CProgressP p3;
    CTreePage p4;
    m_Sheet.AddPage(&p1);
    m_Sheet.AddPage(&p2);
```

```
m_Sheet.AddPage(&p3);  
m_Sheet.AddPage(&p4);  
m_Sheet.DoModal();  
}
```

Listing: Hinzufügen von Seiten zu einem Eigenschaftfenster (Variante 2)

Nach dem Anlegen der Instanz des Property Sheets werden Instanzen der Seiten angelegt und diese wiederum mit *AddPage* hinzugefügt. Die Methode *DoModal* startet den Property-Sheet-Dialog wie beim Aufruf eines normalen Dialogs.



Weitere Informationen sind in der Class Library Reference unter *CPropertyPage* und *CPropertySheet* zu finden. Auch sind im Lieferumfang von Visual C++ einige Beispiele dazu enthalten. Eine Übersicht über die Vielzahl der Möglichkeiten mit Property Sheets würde den Umfang dieses Buches bei weitem sprengen.



Auf der mitgelieferten CD ist das Beispielprogramm **CONTROLS** enthalten, welches auch eine Implementation eines Property Sheets enthält. Ändern Sie einfach die Flags (*PSH_WIZARD*), um zu sehen, wie ein Assistent erzeugt wird.

10.11 Registerkarte (Tab Control)

Die Registerkarten-Steuerung sieht Property Pages sehr ähnlich. Anders als diese ist sie aber eine eigenständige Steuerung und damit auch in der Werkzeugleiste **STEUERELEMENTE** enthalten. Ein typisches Anwendungsbeispiel sind die Eigenschaften von Steuerungen. Registerkarten-Steuerungen werden ebenfalls eingesetzt, um die Übersichtlichkeit in Dialogen zu erhöhen. Da der Aufwand der Programmierung von Registerkarten etwas höher ist als bei Property Pages, sollten nach Möglichkeit letztere verwendet werden. Der einzige Unterschied in der Anwendung ist der, daß Property-Pages mit fest vorgegebenen Schaltflächen bestückt sind. Wenn diese nicht gewünscht oder weitere Steuerungen außerhalb der einzelnen Seiten erforderlich sind, muß auf eine Registerkarten-Steuerung ausgewichen werden.

10.11.1 Attribute

Außer den Standard-Eigenschaften und den erweiterten Formaten besitzt ein *Tab Control* weitere Attribute, die in Tabelle 10.20 zusammengefaßt sind.

Dialogeditor	Dialogboxschablone	Bedeutung
Fokus: Standard, Bei Tastendruck, Nie (Focus: Default, OnButtonDown, Never)	Ressource-Typ: CONTROL Style: <i>TCS_FOCUSONBUTTONDOWN</i> <i>TCS_FOCUSNEVER</i>	Wenn <i>OnButtonDown</i> gewählt wird (in Zusammenhang mit <i>TCS_BUTTONS</i>), wird beim Drücken der Taste der Eingabefokus gewechselt. Bei <i>Never</i> erhält ein Tab nie den Eingabefokus.
Ausrichtung: Rechtsbündig, Feste Breite, Linksbündig (Alignment: Right Justify, Fixed Width, Ragged Right)	Ressource-Typ: CONTROL Style: <i>TCS_RIGHTJUSTIFY</i> <i>TCS_FIXEDWIDTH</i> <i>TCS_RAGGEDRIGHT</i>	Legt die Größe und Anordnung der Reiter fest: Bei fester Breite werden alle Reiter gleich groß dargestellt. Linksbündig verhindert, daß die Größe der Tabs je Zeile auf die Breite des Controls erweitert werden. Der Standard rechtsbündig besagt, daß die Reitergröße auf die Breite der Steuerung angepaßt wird.
Schaltfläche (Button)	Ressource-Typ: CONTROL Style: <i>TCS_BUTTONS</i>	Die Reiter werden als Schaltflächen dargestellt.
QuickInfo (ToolTips)	Ressource-Typ: CONTROL Style: <i>TCS_TOOLTIPS</i>	Sagt aus, daß das Tab Control <i>Tooltips</i> enthält.
Ausrichtung von Icon und Label im Reiter	Ressource-Typ: CONTROL Style: <i>TCS_FORCEICONLEFT</i> <i>TCS_FORCELABELLEFT</i>	Standardmäßig sind Symbole und Bezeichner der Reiter zentriert angeordnet. Force Icon Left ordnet das Symbol links an, läßt das Label aber zentriert. Force Label Left läßt auch den Bezeichner linksbündig erscheinen. Die Styles sind nur mit fester Breite zu verwenden.
Rand (Border)	Ressource-Typ: CONTROL Style: <i>TCS_TABS</i>	Dieser Style legt fest, daß um das Tab Control eine Umrandung gezogen wird.
Mehrzeilig (Multi-Line)	Ressource-Typ: CONTROL Style: <i>TCS_MULTILINE</i>	Mit Hilfe dieses Styles lassen sich die Reiter des Controls bei Bedarf auf mehrere Zeilen aufteilen. Die beiden kleinen Pfeile zum Scrollen der Tabs entfallen. Standard ist <i>TCS_SINGLELINE</i> .

Tabelle 10.20: Attribute der Registerkarten-Steuerung

10.11.2 Benachrichtigungscodes

Registerkarten senden einige weitere Benachrichtigungscodes. Diese müssen entsprechend der Funktion der Steuerung angewendet werden. Auch hier empfiehlt es sich, die Nachrichten-Funktionen mit dem Klassen-Assistenten anzulegen.

Die Nachricht *TCN_SELCHANGE* wird verschickt, sobald die Seite gewechselt wurde. Dagegen wird *TCN_SELCHANGING* an das Eltern-Fenster geschickt, wenn die aktuelle Registerseite verlassen werden soll. Da-

mit wird erreicht, daß mit der Nachricht TCN_SELCHANGING die aktuelle Seite verborgen wird und – sobald TCN_SELCHANGE geschickt wird – die neue Seite sichtbar wird. TCN_KEYDOWN besagt, daß eine Taste gedrückt wurde.

10.11.3 Die Struktur TC_ITEM

Ein TC_ITEM-Element enthält alle Daten der Seite einer Registerkarte. Diese Daten sind nur für den Reiter des Controls maßgebend, nicht jedoch für den Inhalt der Seite.

```
typedef struct _TC_ITEM {
    UINT mask;           // Gültigkeit
    UINT lpReserved1;
    UINT lpReserved2;
    LPSTR pszText;       // Zeiger auf den Text
    int cchTextMax;      // Größe des Puffers
    int iImage;          // Index zum Symbol
    LPARAM lParam;       // nutzerdefinierte Daten
} TC_ITEM;
```

Der Parameter `mask` legt die Gültigkeit der Einträge in der Struktur fest. TCIF_ALL bedeutet, daß alle Parameter gültig sind, sind nur einzelne definiert, so können einer oder mehrere der Bezeichner der Tabelle 10.21 entnommen werden.

Parameter	Bedeutung
TCIF_TEXT	pszText ist gültig
TCIF_IMAGE	iImage ist gültig
TCIF_PARAM	lParam wurde definiert
TCIF_RTREADING	Anzeige des Textes von rechts nach links

Tabelle 10.21: Der mask-Parameter der TC_ITEM-Struktur

In einem Dialog wird die Registerkarten-Steuerung am einfachsten in der Methode *OnInitDialog* initialisiert. Dazu wird dem Parameter `mask` der Wert TCIF_TEXT zugewiesen und dieser in `pszText` eingefügt.

```
TC_ITEM TabCtrlItem;
TabCtrlItem.mask = TCIF_TEXT;
TabCtrlItem.pszText = "Tab 1";
```

Sollen als Reiter Symbole benutzt werden, so ist ein *CImageList*-Objekt mit den entsprechenden Bitmaps anzulegen. Die TC_ITEM-Struktur muß in

`iImage` den Index des *CImageList*-Objekts enthalten sowie in `mask` den Wert `TCIF_IMAGE`. Damit wird anstelle von Text ein Bild im Reiter angezeigt.

Ändern Sie das Beispielprogramm `CONTROLS` so ab, daß für den ersten und zweiten Reiter ein Bild angezeigt wird.



Die Verwendung der Klasse *CImageList* wird in Kapitel 11.3 beschrieben.



10.11.4 Die Klasse *CTabCtrl*

InsertItem

Mit Hilfe der Methode *InsertItem* wird ein neuer Reiter einem Tab Control hinzugefügt. Als Parameter werden nur der Index des Reiters sowie die zugehörige `TC_ITEM`-Struktur angegeben. Als Rückgabewert wird der nullbasierende Index geliefert oder -1 bei einem Fehler.

```
class CTabCtrl:
    BOOL InsertItem( int nItem,
                    TC_ITEM* pTabCtrlItem );
```

Im folgenden Beispiel ist ein Ausschnitt aus der Methode *OnInitDialog* zu sehen, mit dessen Hilfe einer Registerkarte mehrere Reiter hinzugefügt werden.

```
BOOL CTab::OnInitDialog()
{
    CDialog::OnInitDialog();
    TC_ITEM TabCtrlItem;
    TabCtrlItem.mask = TCIF_TEXT;
    TabCtrlItem.pszText = "Tab 1";
    m_Tab.InsertItem( 0, &TabCtrlItem );
    TabCtrlItem.pszText = "Zweiter Tab";
    m_Tab.InsertItem( 1, &TabCtrlItem );
    TabCtrlItem.pszText = "Optionen";
    m_Tab.InsertItem( 2, &TabCtrlItem );
    TabCtrlItem.pszText = "Multiline";
    m_Tab.InsertItem( 3, &TabCtrlItem );
    TabCtrlItem.pszText = "Langer Text als TAB";
    m_Tab.InsertItem( 4, &TabCtrlItem );
    return TRUE;
}
```

Listing: Aufbauen einer Registerkarte in *OnInitDialog*

DeleteItem / DeleteAllItems

Diese beiden Methoden dienen dem Löschen eines Reiters oder aller Reiter. Soll nur ein Reiter gelöscht werden, muß der Index (nullbasierend) angegeben werden. Wird der Wert 0 zurückgegeben, war das Löschen nicht erfolgreich.

```
class CTabCtrl:
    BOOL DeleteItem( int nItem );
    BOOL DeleteAllItems( );
```

SetImageList und GetImageList

Werden für die Reiter Symbole verwendet, müssen diese in einem *CImageList*-Objekt gespeichert werden. Dieses Objekt muß mit *SetImageList* der Klasse *CTabCtrl* bekanntgemacht werden. *SetImageList* liefert den Zeiger des zuvor gültigen *CImageList*-Objekts zurück oder NULL, wenn noch kein Objekt gesetzt war. *GetImageList* gibt den Zeiger auf die Image-Liste zurück oder NULL bei nicht erfolgreicher Ausführung.

```
class CTabCtrl:
    CImageList * SetImageList( CImageList *
                               pImageList );
    HIMAGELIST GetImageList( ) const;
```

GetItem und SetItem

Mit der Methode *GetItem* kann die TC_ITEM-Struktur eines Reiters gelesen werden, mit *SetItem* können für einen Tab neue Werte, zum Beispiel ein neuer Text, eingestellt werden. Als Parameter dienen jeweils der Index des Reiters sowie ein Zeiger auf eine TC_ITEM-Struktur.

```
class CTabCtrl:
    BOOL GetItem( int nItem,
                 TC_ITEM* pTabCtrlItem ) const;
    BOOL SetItem( int nItem,
                 TC_ITEM* pTabCtrlItem );
```

GetItemCount

GetItemCount hat keine andere Funktion, als die Anzahl der Reiter im Control zurückzugeben. *GetItemCount()-1* ist der höchste Index, der zum Beispiel in der Methode *GetItem* benutzt werden kann. Um Fehler zu vermeiden, sollte der Wert des Indexes immer überprüft werden.

```
class CTabCtrl:
    int GetItemCount( ) const;
```

GetCurSel und SetCurSel

Während mit der Methode *GetCurSel* der gerade aktive Reiter ermittelt wird, kann dieser mit *SetCurSel* gesetzt werden.

```
class CTabCtrl:
    int GetCurSel( ) const;
    int SetCurSel( int nItem );
```

Wird mit *SetCurSel* ein neuer Reiter aktiviert, so werden nicht die üblichen Nachrichten `TCN_SELCHANGING` und `TCN_SELCHANGE` geschickt. Deshalb liefert *SetCurSel* als Rückgabewert den Index des gerade aktiven Reiters. Damit kann die aktuelle Seite deaktiviert werden.

Weitere Methoden sind in der Online-Hilfe zur Klasse *CTabCtrl* beschrieben.



10.11.5 Beispielprogramm

Die Funktionsweise einer Registerkarten-Steuerung ist im Beispiel-Programm `CONTROLS` enthalten. Dabei wird die Registerkarte als Steuerung im Ressource-Editor in einen Dialog eingefügt. In den Eigenschaften des Controls werden die gewünschten (mehrzeilig) eingestellt.

Testen Sie die Wirkungsweise verschiedener Eigenschaften einfach aus.

Über einen Doppelklick auf das Tab Control wird der Klassen-Assistent aufgerufen. Dieser fordert zum Anlegen einer neuen Klasse für die Steuerung auf. Wenn bereits eine Klasse angelegt wurde, springt man mit einem Doppelklick auf die erste Member-Funktion im Quelltext. Ist keine Funktion vorhanden, wird ein Dialog zum Anlegen einer neuen Funktion geöffnet. Für die Registerkarte werden zwei Funktionen angelegt, die auf die Nachrichten `TCN_SELCHANGE` und `TCN_SELCHANGING` reagieren. Die Member-Funktion *OnSelchangingTab* wird dazu benutzt, die aktuelle Dialogseite unsichtbar zu machen.



```
void CTab::OnSelchangingTab(NMHDR* pNMHDR,
                           LRESULT* pResult)
{
    switch( m_Tab.GetCurSel())
    {
        case 0: cPg1.ShowWindow(SW_HIDE); break;
        case 1: cPg2.ShowWindow(SW_HIDE); break;
        case 2: cPg3.ShowWindow(SW_HIDE); break;
```

```

        case 3: cPg4.ShowWindow(SW_HIDE); break;
        case 4: cPg5.ShowWindow(SW_HIDE); break;
    }
    *pResult = 0;
}

```

Listing: Verbergen aller Register vor dem Wechseln der Seite

Sobald dann die Funktion *OnSelchangeTab* aufgerufen wird, wird die neue Seite dargestellt.

```

void CTab::OnSelchangeTab(NMHDR* pNMHDR, LRESULT* pResult)
{
    switch( m_Tab.GetCurSel())
    {
        case 0: cPg1.ShowWindow(SW_SHOW); break;
        case 1: cPg2.ShowWindow(SW_SHOW); break;
        case 2: cPg3.ShowWindow(SW_SHOW); break;
        case 3: cPg4.ShowWindow(SW_SHOW); break;
        case 4: cPg5.ShowWindow(SW_SHOW); break;
    }
    *pResult = 0;
}

```

Listing: Anzeigen der ausgewählten Registerseite

Diese beiden Funktionen bilden die Schlüssel zur Anwendung eines Tab Controls. Jede Seite des Tab Controls stellt einen eigenen Dialog dar. Dieser Dialog hat weder eine Titelzeile mit System-Menü noch einen Rahmen, damit sieht er so aus, als sei er fest mit der Registerkarte verbunden.

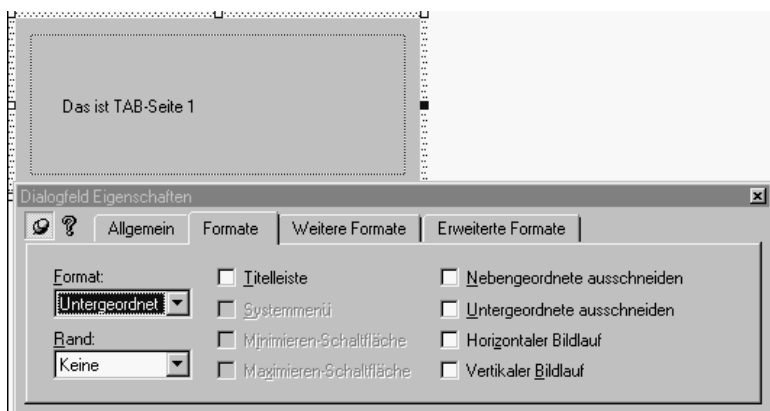


Abbildung 10.4: Die Dialogseite ohne Titelzeile und Rahmen

Die einzelnen Dialogseiten werden nicht wie üblich über die Methode *DoModal* aufgerufen, sondern mit Hilfe der Methode *Create* erzeugt und über *ShowWindow* sichtbar gemacht. Das Anlegen der Dialogseiten erfolgt in der Methode *OnShowWindow* der Tab-Control-Klasse. Dieser Trick ist notwendig, da zur Laufzeit die Registerkarten-Steuerung erst nach Durchlaufen der Methode *OnInitDialog* erzeugt wird, welche aber zum Anlegen der Dialogseiten schon vorhanden sein muß.

Ein weiterer Punkt ist beim Gestalten der Dialogseiten zu beachten. Diese werden relativ zum Tab Control positioniert. Dazu wird in der Methode *Create* die Position des Eltern-Fensters übergeben.

```
void CTab::OnShowWindow(BOOL bShow, UINT nStatus)
{
    CDialog::OnShowWindow(bShow, nStatus);
    if(bShow)
    {
        cPg5.Create(IDD_TAB5, m_Tab.GetActiveWindow());
        cPg5.ShowWindow(SW_HIDE);
        cPg4.Create(IDD_TAB4, m_Tab.GetActiveWindow());
        cPg4.ShowWindow(SW_HIDE);
        cPg3.Create(IDD_TAB3, m_Tab.GetActiveWindow());
        cPg3.ShowWindow(SW_HIDE);
        cPg2.Create(IDD_TAB2, m_Tab.GetActiveWindow());
        cPg2.ShowWindow(SW_HIDE);
        cPg1.Create(IDD_TAB1, m_Tab.GetActiveWindow());
        cPg1.ShowWindow(SW_SHOW);
    }
}
```

Listing: Erzeugen und Positionieren der Registerseiten

Damit werden die Dialogseiten relativ gesehen links oben im Tab Control positioniert. Da im oberen Teil aber noch die Reiter angeordnet sind, müssen die Seiten noch verschoben werden. Diese Verschiebung ermittelt man am einfachsten durch einige Versuche. Die Verschiebung selbst wird auf der Eigenschafts-Seite des Dialogs eingetragen.



Abbildung 10.5: Eintragen der Verschiebungsvektoren der Dialogseite

Unter *X Pos* und *Y Pos* werden numerische Werte eingetragen, die die Verschiebung des Fensters bestimmen.

Meist werden sich in den einzelnen Dialogseiten eine Vielzahl von Steuerungen befinden, die initialisiert und abgefragt werden müssen. Ein Beispiel für ein Eingabefeld ist auf der zweiten Dialogseite enthalten. Die in der Klasse *CTab* enthaltene öffentliche Variable *nWert* wird an die Dialogseite übergeben. Dies kann erst dann erfolgen, wenn die Dialogseite existiert, also nach dem Anlegen in der Methode *OnShowWindow* des Tab Controls. Dabei wird ein typisierter Zeiger auf die Steuerung mit der Funktion *GetDlgItem* geholt. Über diesen Zeiger können dann Funktionen zur Steuerung aufgerufen werden. Bei einem Kontrollkästchen kann das zum Beispiel die Funktion *SetCheck* sein. In unserem Beispiel wird mit Hilfe von *SetWindowText* der Wert in das Eingabefeld eingetragen und später mit *GetWindowText* ausgelesen.

```
CEdit * pEdCtrl;
pEdCtrl = (CEdit*)cPg2.GetDlgItem(IDC_WERT);
sprintf(buf, "%d", nWert);
pEdCtrl -> SetWindowText (buf);
```

Die Werte der Dialogseiten werden vor dem Zerstören des Controls in der Methode *OnDestroy* ausgelesen. Um zu verdeutlichen, wann welche Funktion aufgerufen wird, wurden im Quelltext an geeigneten Stellen TRACE-Meldungen implementiert. Diese werden im Debug-Modus im Ausgabefenster angezeigt.



Fügen Sie auf der dritten Seite des Tab Controls zwei Kontrollkästchen und auf der vierten Seite ein Kombinationsfeld ein. Das Kombinationsfeld soll mindestens drei Einträge erhalten (zur Laufzeit), wovon der zweite aktiviert werden soll. Das erste Kontrollkästchen soll aktiviert, das zweite deaktiviert sein. Die Ergebnisse werden in einer MessageBox beim Verlassen des Dialogs angezeigt.

10.12 Listenelement (List Control)

10.12.1 Anwendungen

Das Listenelement bietet zunächst den gleichen Funktionsumfang wie Listboxen: Es dient zur Anzeigen einer Vielzahl von Daten in einer Liste. Dazu gesellen sich eine Reihe von Erweiterungen und Verbesserungen, die dem Anwender einen einfacheren Umgang mit den Daten erlauben. Eine Implementierung des Listenelements verbirgt sich hinter dem *Arbeitsplatz*-Icon auf dem Desktop. Zusätzlich zu den Daten kann jedes Element ein Symbol aus einer Image-Liste erhalten. Des weiteren können die Daten auf vier verschiedene Weisen dargestellt werden: Große Symbole, Kleine Symbole, Liste oder Report. Diese können schon in den Eigenschaf-

ten des Controls festgelegt werden. Eine weitere Eigenschaft bietet die Möglichkeit, die Text-Labels zur Laufzeit vom Anwender ändern zu lassen. So wurde zum Beispiel das Umbenennen von Dateien auf einfachste Art ermöglicht. In der Report-Darstellung wird zusätzlich eine Kopfzeile angezeigt (auch abschaltbar); damit wird eine tabellarische Darstellung der Daten ermöglicht. Die Spaltenbreite läßt sich einfach ändern, ein Druck auf einen Spaltenkopf kann für Sortierungen verwendet werden und ähnliches.

Die Klasse *CListView* implementiert ein Listenelement in einer View nach dem Dokument-Ansicht-Prinzip der MFC. Darauf soll hier nicht weiter eingegangen werden.



10.12.2 Attribute

Außer den Standard-Attributen und den erweiterten Formaten besitzt ein Listenelement weitere Attribute, die in Tabelle 10.22 zusammengefaßt sind.

Dialogeditor	Dialogboxschablone	Bedeutung
Ansicht (View)	Ressource-Typ: CONTROL Style: <i>LVS_ICON</i> <i>LVS_SMALLICON</i> <i>LVS_LIST</i> <i>LVS_REPORT</i>	Wählt die Art der Darstellung der Elemente in der Liste aus (Große Symbole, Kleine Symbole, Liste oder Details bzw. Report).
Ausrichtung (Align)	Ressource-Typ: CONTROL Style: <i>LVS_ALIGNLEFT</i> <i>LVS_ALIGNTOP</i> <i>LVS_AUTOARRANGE</i>	Legt die Ausrichtung der Einträge in der Ansicht <i>Kleine und Große Symbole</i> fest: links oder oben ausgerichtet. Über <i>Auto arrange</i> wird festgelegt, daß die Einträge bestmöglichst im Fenster des Controls eingepaßt werden.
Automatisch anordnen (Auto arrange)		
Sortieren (Sort)	Ressource-Typ: CONTROL Style: <i>LVS_SORTASCENDING</i> <i>LVS_SORTDESCENDING</i>	Die automatische Sortierung der Einträge kann mit diesen Styles festgelegt werden. Dabei wird nur den Haupteinträgen nach sortiert. Soll im Report-Modus auch nach Untereinträgen sortiert werden, muß dies über eigene Vergleichsfunktionen gelöst werden.
Einzelauswahl (Single selection)	Ressource-Typ: CONTROL Style: <i>LVS_SINGLESEL</i>	Mit diesem Style wird erreicht, daß immer nur ein Element gewählt werden kann. Andernfalls können mehrere Elemente gleichzeitig aus der Liste ausgewählt werden.

Tabelle 10.22: Attribute für Listenelemente

Dialogeditor	Dialogboxschablone	Bedeutung
Kein Umbruch in Bezeichnung (No label wrap)	Ressource-Typ: CONTROL Style: <i>LVS_NOLABELWRAP</i>	Wenn verhindert werden soll, daß die Texte der Elemente in der Symbol-Ansicht umgebrochen werden (wenn sie zu lang sind), muß <i>LVS_NOLABELWRAP</i> aktiviert werden. Sie werden dann unabhängig von der Länge einzellig dargestellt.
Bezeichnungen bearbeiten (Edit labels)	Ressource-Typ: CONTROL Style: <i>LVS_EDITLABELS</i>	Soll dem Anwender die Möglichkeit eingeräumt werden, die Texte zu ändern, muß <i>LVS_EDITLABELS</i> gesetzt werden. Es ermöglicht zum Beispiel das Umbenennen von Dateien.
Kein Bildlauf (No scroll)	Ressource-Typ: CONTROL Style: <i>LVS_NOSCROLL</i>	Über diesen Style wird erreicht, daß keine Scroll-Leisten angezeigt werden und damit auch kein Scrollen möglich ist. Dieses Attribut ist nur sinnvoll, wenn alle Elemente im Fenster des Controls sichtbar sind.
Kein Spaltenkopf (No column header)	Ressource-Typ: CONTROL Style: <i>LVS_NOCOLUMNHEADER</i>	Standardmäßig werden in der Report-Ansicht die Spaltentitel in Form von Knöpfen angezeigt. Bei einem Mausklick auf einen Spaltenkopf wird die Nachricht <i>LVN_COLUMNCLICK</i> geschickt, mit der zum Beispiel eine Sortierung angestoßen werden kann. Dient der Spaltentitel nur zur Anzeige und soll keine Funktion aufgerufen werden, kann <i>LVS_NOSORTHEADER</i> vergeben werden. Sollen die Spaltentitel gar nicht dargestellt werden, so ist <i>LVS_NOCOLUMNHEADER</i> erforderlich.
Kein Sortier-Spaltenkopf (No sort header)	<i>LVS_NOSORTHEADER</i>	
Auswahl immer zeigen (Show selectselection always)	Ressource-Typ: CONTROL Style: <i>LVS_SHOWSELALWAYS</i>	Normalerweise werden die ausgewählten Elemente nur so lange angezeigt, wie der Eingabefokus beim <i>List Control</i> liegt. Soll die Auswahl auch nach Abgabe des Eingabefokusses noch sichtbar sein, muß dieses Attribut gesetzt werden.
Besitzerzeichnung fixiert (Owner draw fixed)	Ressource-Typ: CONTROL Style: <i>LVS_OWNERDRAWFIXED</i>	Wenn es erforderlich sein sollte, die Darstellung der Elemente in einer eigenen Funktion vorzunehmen, so ist dieser Style zu wählen. Die Methode <i>DrawItem</i> muß dabei <i>überschrieben werden</i> .
Rand (Border)	Ressource-Typ: CONTROL Style: <i>WS_BORDER</i>	Mit Hilfe dieses Styles wird erreicht, daß um die Steuerung ein Rahmen gezogen wird.
Gemeinsame Bildliste (Share image list)	Ressource-Typ: CONTROL Style: <i>LVS_SHAREIMAGELIST</i>	Durch dieses Attribut wird die Liste der Icons (Image Liste) auch anderen <i>List Controls</i> verfügbar gemacht und nicht zerstört, wenn die Steuerung zerstört wird.

Tabelle 10.22: Attribute für Listenelemente

10.12.3 Benachrichtigungscode

Das Listenelement sendet eine Reihe von Nachrichten über besondere Ereignisse oder Veränderungen. Diese Nachrichten können mit Hilfe von Einträgen in der Message-Map umgeleitet werden.

Tabelle 10.23 gibt eine Übersicht der wichtigsten Benachrichtigungscodes und der zugehörigen Message-Map-Einträge.

Benachrichtigungscode	Message-Map-Eintrag
<code>LVN_BEGINDRAG</code>	<code>ON_NOTIFY(wNotifyCode, id, memberFxn)</code>
<code>LVN_BEGINLABELEDIT</code>	<code>ON_NOTIFY(wNotifyCode, id, memberFxn)</code>
<code>LVN_BEGINRDRAG</code>	<code>ON_NOTIFY(wNotifyCode, id, memberFxn)</code>
<code>LVN_COLUMNCLICK</code>	<code>ON_NOTIFY(wNotifyCode, id, memberFxn)</code>
<code>LVN_DELETEALLITEMS</code>	<code>ON_NOTIFY(wNotifyCode, id, memberFxn)</code>
<code>LVN_DELETEITEM</code>	<code>ON_NOTIFY(wNotifyCode, id, memberFxn)</code>
<code>LVN_ENDLABELEDIT</code>	<code>ON_NOTIFY(wNotifyCode, id, memberFxn)</code>
<code>LVN_GETDISPINFO</code>	<code>ON_NOTIFY(wNotifyCode, id, memberFxn)</code>
<code>LVN_INSERTITEM</code>	<code>ON_NOTIFY(wNotifyCode, id, memberFxn)</code>
<code>LVN_ITEMCHANGED</code>	<code>ON_NOTIFY(wNotifyCode, id, memberFxn)</code>
<code>LVN_ITEMCHANGING</code>	<code>ON_NOTIFY(wNotifyCode, id, memberFxn)</code>
<code>LVN_KEYDOWN</code>	<code>ON_NOTIFY(wNotifyCode, id, memberFxn)</code>
<code>LVN_SETDISPINFO</code>	<code>ON_NOTIFY(wNotifyCode, id, memberFxn)</code>

Tabelle 10.23: Benachrichtigungscodes des List Controls

Einige der Nachrichten werden paarweise versandt. So wird `LVN_BEGINLABELEDIT` geschickt, wenn der Anwender anfangen will, ein Label zu editieren, und `LVN_ENDLABELEDIT`, wenn er damit fertig ist und den Editiermodus verläßt. Nachrichtenbehandlungsfunktionen richtet man am einfachsten über die Assistentenleiste oder den Klassen-Assistenten ein. Die Nachrichten `LVN_ITEMCHANGING` und `LVN_ITEMCHANGED` sind ebenfalls miteinander verwandt: `LVN_ITEMCHANGING` wird geschickt, wenn der Anwender ein anderes Element der Steuerung auswählt, und zwar bevor das alte Element den Fokus verliert; `LVN_ITEMCHANGED` heißt, daß das neue Element den Fokus erhalten hat. Die Nachricht `LVN_DELETEITEM` informiert über das Löschen eines Eintrages, während `LVN_DELETEALLITEMS` aussagt, daß alle Elemente gelöscht wurden.

10.12.4 Die Klasse *CListControl*

Die Klasse *CListCtrl* bietet dem Programmierer eine große Zahl von Methoden an, von denen im folgenden Abschnitt die wichtigsten erläutert werden. Grundlage der Benutzung dieser Funktionen ist das Anlegen einer Member-Variablen für die Steuerung in der *Dialog*-Klasse.

InsertItem

Um ein *List Control* mit Elementen zu füllen, die dann dem Anwender präsentiert werden, muß jeder Eintrag mit *InsertItem* in die Liste eingefügt werden. Die Methode ist mehrfach vorhanden und kann mit verschiedenen Parametern aufgerufen werden. Als Ergebnis liefert die Funktion die Nummer des angelegten Eintrags von Null beginnend oder im Fehlerfall -1 zurück.

```
class CListCtrl:
    int InsertItem(const LV_ITEM* pItem );
    int InsertItem(int nItem, LPCTSTR lpszItem);
    int InsertItem(int nItem, LPCTSTR lpszItem,
        int nImage );
    int InsertItem(UINT nMask, int nItem,
        LPCTSTR lpszItem,
        UINT nState,
        UINT nStateMask, int nImage,
        LPARAM lParam );
```

In den meisten Fällen bietet sich die Benutzung der LV_ITEM-Struktur an. Diese Struktur enthält alle Elemente, die ein Eintrag eines Listenelements haben kann. Man muß aber nur die Elemente füllen, die wirklich benötigt werden. Der Parameter *mask* gibt an, welche der Strukturelemente gültig sind (siehe Tabelle 10.24).

```
typedef struct _LV_ITEM {
    UINT    mask;
    int     iItem;
    int     iSubItem;
    UINT    state;
    UINT    stateMask;
    LPTSTR  pszText;
    int     cchTextMax;
    int     iImage;
    LPARAM  lParam;
} LV_ITEM;
```

In *iItem* wird der nullbasierende Index des Eintrags gespeichert und *iSubItem* gibt an, in welcher Spalte der Eintrag erfolgen soll. Dabei ist die Spalte Null der Haupteintrag, der in allen Ansichten zu sehen ist. Alle Spalten größer Null (*iSubItem*>0) sind nur in der Report-Darstellung zu sehen. Der Text des Eintrags wird im Parameter *pszText* gespeichert. Ein auf ein Bild hinweisender Index in der zugehörigen Image-Liste, die zuvor mit *SetImageList* zugewiesen wurde, ist in *iImage* abgelegt.

Wert	Bedeutung
LVIF_TEXT	Der Parameter <code>pszText</code> enthält einen gültigen Wert.
LVIF_PARAM	Der Parameter <code>lParam</code> ist gültig.
LVIF_STATE	Der <code>state</code> -Parameter ist gefüllt.
LVIF_IMAGE	Der Wert von <code>iImage</code> zeigt auf einen gültigen Eintrag in der Image-Liste.

Tabelle 10.24: `mask`-Parameter der `LV_ITEM`-Struktur

Die Parameter `state` und `stateMask` werden benutzt, um den Status eines Eintrags zu kennzeichnen. Sicher haben Sie schon einmal Kontrollkästchen vor den Einträgen eines List Controls gesehen (z.B. bei der *Benutzerdefinierten Installation* von Visual C++). Der Status und dessen Darstellung wird durch den `state`-Parameter gesteuert.

Schauen Sie sich einmal das Beispielprogramm ROWLIST auf der Visual C++ CD an. Dort kann man mit der Darstellung des Statusses experimentieren.



Generell wird zwischen zwei möglichen Darstellungs-Varianten unterschieden: Die Anzeige der Status-Icons kann vor einem Eintrag erfolgen oder über das Icon des Haupteintrags gelegt werden. Die Icons für den Status werden ebenfalls als Ressourcen angelegt. Es können Icon-Ressourcen sein oder – wie im ROWLIST-Beispiel – Bitmap-Ressourcen. Für die Status-Icons vor den Einträgen ist eine eigene Image-Liste zuständig. Sie wird mit dem Schalter `LVSIL_STATE` definiert. Für die Anzeige des Zustands über den Icons wird ein sogenanntes »Overlay-Bild« verwendet, das in der jeweiligen Image-Liste enthalten sein muß. Mit der Methode *SetOverlayImage* wird bekanntgegeben, welche Icons als Status-Icons verwendet werden. Aus dem eigentlichen Icon und dem Status-Icon wird dann zur Laufzeit dem Status entsprechend das Icon zusammengesetzt. Der Status wird mit der Methode *SetItemState* gesetzt oder geändert.

DeleteItem und DeleteAllItems

Diese beiden Methoden dienen zum Löschen eines oder aller Einträge des Listenelements. Soll nur ein Element gelöscht werden, muß dessen Index angegeben werden. Als Ergebnis wird ein Wert ungleich Null zurückgegeben (TRUE), wenn das Löschen erfolgreich war, andernfalls FALSE.

```
class CListCtrl:
    BOOL DeleteItem( int nItem );
    BOOL DeleteAllItems( );
```

InsertColumn

Die Benutzung mehrerer Spalten ist im Listenelement standardmäßig implementiert. Der Programmierer muß sich nicht mehr um eine tabellarische Anzeige der Daten mittels Tabulatoren kümmern. Mehrere Spalten werden nur in der Report-Ansicht dargestellt. Das Anlegen der Spalten erfolgt mit der Methode *InsertColumn*. Der erste Parameter, der angegeben werden muß, ist die Nummer der Spalte, die eingefügt wird. Spalte Null kennzeichnet dabei den Haupteintrag.

```
class CListCtrl:  
    int InsertColumn( int nCol,  
                     const LV_COLUMN* pColumn);  
    int InsertColumn( int nCol,  
                     LPCTSTR lpszColumnHeading,  
                     int nFormat = LVCFMT_LEFT,  
                     int nWidth = - 1,  
                     int nSubItem = - 1 );
```

Als zweiter Parameter wird ein Zeiger auf eine LV_COLUMN-Struktur übergeben. Die darin enthaltenen Elemente kennzeichnen je eine Spalte. Bei der zweiten Form der Methode *InsertColumn* werden einige Parameter der LV_COLUMN-Struktur direkt übergeben.

```
typedef struct _LV_COLUMN {  
    UINT mask;  
    int fmt;  
    int cx;  
    LPSTR pszText;  
    int cchTextMax;  
    int iSubItem;  
} LV_COLUMN;
```

Eine Spalte wird nur durch wenige Parameter definiert. Das sind der Titeltext der Spalte im Parameter *pszText* und die Breite der Spalte in *cx*. Des weiteren kann die Ausrichtung der Einträge (links, Mitte, rechts) in *fmt* bestimmt werden. Zudem wird im Parameter *mask* festgelegt, welche der Parameter gültig sind. In *iSubItem* wird die Zuordnung des Untereintrags zu einer Spalte festgehalten.

DeleteColumn

Mit der Methode *DeleteColumn* läßt sich eine der Spalten auch wieder löschen. Es muß nur der Index der Spalte angegeben werden. Im Falle des Erfolges wird TRUE zurückgegeben.


```
class CListCtrl:
    BOOL DeleteColumn( int nCol );
```

SetItemText und GetItemText

Um den Text eines Eintrags oder Untereintrags zu ändern, wird die Methode *SetItemText* verwendet. *GetItemText* dient zum Auslesen des Textes. Bei beiden Methoden muß die Nummer des Items und Subitems angegeben werden. Bei *SetItemText* wird noch der neue Text des Eintrags übergeben.

```
class CListCtrl:
    BOOL SetItemText( int nItem, int nSubItem,
                     LPTSTR lpszText );
    int GetItemText( int nItem, int nSubItem,
                    LPTSTR lpszText,
                    int nLen ) const;
    CString GetItemText( int nItem,
                        int nSubItem ) const;
```

SortItems

Die Sortierung der Einträge in einem List Control wird in den meisten Fällen der Programmierer selbst vornehmen müssen. Die Sortierung über die Styles LVS_SORTx läuft über den Text des Haupteintrags ab. Soll nach Untereinträgen oder nur nach einem Datum sortiert werden, muß eine eigene Sortierung implementiert werden. Die Methode *SortItems* bietet die Schnittstelle zwischen dem Listenelement und einer anwenderdefinierten Sortierung, indem eine eigene Vergleichsfunktion geschrieben wird. Der Name der Funktion sowie ein 32-Bit-Wert werden *SortItems* übergeben. Der Anwender muß die Vergleichsfunktion dann implementieren.

```
class CListCtrl:
    int CALLBACK CompareFunc(LPARAM lParam1,
                             LPARAM lParam2, LPARAM lParamSort);
```

Die Funktion liefert einen negativen Wert, wenn der erste Parameter größer ist als der zweite, oder einen Wert größer Null, wenn er kleiner ist. Bei Null sind beide Werte gleich.

Ein Beispiel für die Implementierung einer wiederverwendbaren Klasse mit Sortierung, die von *CListCtrl* abgeleitet ist, finden Sie im Microsoft System Journal 2/97.



```
class CListCtrl:
    BOOL SortItems( PFNLVCOMPARE pfnCompare,
                   DWORD dwData );
```

10.12.5 Beispielprogramm

Das Beispiel zum Listenelement ist im Programm CONTROLS enthalten. Implementiert wird darin die Anzeige aller Verzeichnisse eines Pfades.

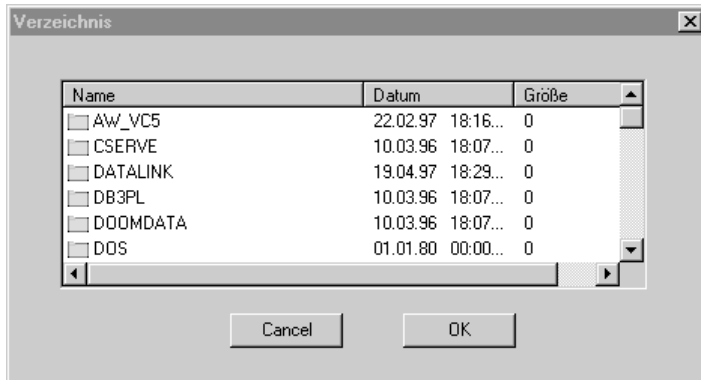


Abbildung 10.6: Der Dialog mit dem List Control

In den Eigenschaften der Steuerung sind die Attribute für die Report-Ansicht und aufsteigende Sortierung gesetzt worden. Des weiteren müssen *automatisch anordnen* und *Einzelauswahl* gesetzt werden. Nach Anlegen der Dialog-Ressource mit dem Listenelement wird neben einer neuen Klasse für den Dialog zusätzlich eine Member-Variable für das Listenelement angelegt. Anschließend kann mit der Implementierung begonnen werden. Zuerst gilt es, die Methode *OnInitDialog* zu implementieren, die aufgerufen wird, wenn der Dialog erzeugt wird. Die Methode wird mit dem Klassen-Assistenten oder der Assistentenleiste angelegt.

```

BOOL CListTest::OnInitDialog()
{
    CDialog::OnInitDialog();
    CControlsApp *pApp;
    CRect rect;
    pApp = (CControlsApp *)AfxGetApp();
    pImgList = new CImageList();
    pImgListS = new CImageList();
    ASSERT(pImgList != NULL);
    ASSERT(pImgListS != NULL);
    pImgList->Create(32, 32, TRUE, 2, 2);
    pImgListS->Create(16, 16, TRUE, 2, 2);
    pImgList->Add(pApp->LoadIcon(IDI_ICON1));
    pImgList->Add(pApp->LoadIcon(IDI_ICON2));
}

```

```

pImgListS->Add(pApp->LoadIcon(IDI_ICON1));
pImgListS->Add(pApp->LoadIcon(IDI_ICON2));
ReadDir();
return TRUE;
}

```

Listing: Erzeugen von CImageList-Objekten für das Listenelement

In dieser Methode werden zwei Variablen vom Typ *CImageList* angelegt und mit den Icons gefüllt. Anschließend wird die Funktion *ReadDir* aufgerufen, die das Einlesen des Verzeichnisses übernimmt.

```

void CListTest::ReadDir()
{
    int    i, iIcon, iItem, iSubItem, iActualItem;
    LV_ITEM    item;
    LV_COLUMN  column;
    char       buf[256];
    HANDLE     handle;
    WIN32_FIND_DATA fd;
    unsigned long    lSize;
    m_list.SetImageList(pImgList, LVSIL_NORMAL);
    m_list.SetImageList(pImgListS, LVSIL_SMALL);
    //Spaltenstruktur füllen
    column.mask = LVCF_FMT | LVCF_SUBITEM |
                  LVCF_TEXT | LVCF_WIDTH;
    column.fmt = LVCFMT_LEFT;
    column.cx = 200;
    column.pszText = "Name";
    column.iSubItem = 0;
    m_list.InsertColumn(0, &column);
    column.cx = 100;
    column.pszText = "Datum";
    column.iSubItem = 1;
    m_list.InsertColumn(1, &column);
    column.pszText = "Größe";
    column.iSubItem = 2;
    m_list.InsertColumn(2, &column);
    iItem = 0;
    item.iImage = 0;
    CString sPathName = "C:\\";
    CString sSearchSpec = sPathName + ".*.*";
    handle = ::FindFirstFile((LPCTSTR)sSearchSpec, &fd);
    do
    {

```

```

        if (fd.dwFileAttributes &
            FILE_ATTRIBUTE_DIRECTORY)
        {
            CString sFileName = (LPSTR) &fd.cFileName;
            if ((sFileName != ".") && (sFileName != ".."))
            {
                item.mask = LVIF_TEXT | LVIF_IMAGE ;
                item.iSubItem = 0;
                item.iItem = iItem++;
                item.pszText = (LPSTR)&fd.cFileName;
                iActualItem = m_list.InsertItem(&item);
                CTime cTime ( *(&fd.ftCreationTime));
                CString sTime = cTime.Format("%d.%m.%y   %H:%M:%S ");
                sprintf(buf,"%s",sTime);
                item.pszText = buf;
                item.mask = LVIF_TEXT;
                item.iItem = iActualItem;
                item.iSubItem = 1;
                m_list.SetItem(&item);
                lSize = (*(&fd.nFileSizeHigh) * MAXDWORD) +
                    *(&fd.nFileSizeLow);
                sprintf(buf,"%ld",lSize);
                item.pszText = buf;
                item.mask = LVIF_TEXT;
                item.iItem = iActualItem;
                item.iSubItem = 2;
                m_list.SetItem(&item);
            }
        }
    } while (::FindNextFile (handle, &fd));
    ::FindClose(handle);
}

```

Listing: Einlesen eines Verzeichnisses in ein Listenelement

Anfangs werden die Variablen und Strukturen angelegt. Darunter fallen auch die Strukturen für Einträge (*item*) und Spalten (*column*). Zuerst werden die Image-Listen dem Listenelement zugeordnet, eine für die großen und eine für die kleinen Icons. Danach wird die Struktur für die Spalten gefüllt, und es werden drei Spalten angelegt. Nach der Initialisierung einiger Variablen können in einer Schleife die Einträge hinzugefügt werden. Der Haupteintrag wird mit der Methode *InsertItem* dem Listenelement hinzugefügt. Die beiden weiteren Spalten werden mit Hilfe von *SetItem* an den Haupteintrag angehängt, wobei jeweils die Nummer des Unterein-

trags in `iSubItem` übergeben wird. Des weiteren wird in `iItem` die Nummer des Haupteintrags verzeichnet, die von `InsertItem` zurückgegeben wird. Da in diesem Beispiel nur Verzeichnisse berücksichtigt werden, ist die Größe immer Null. Für eine mögliche Erweiterung zur Anzeige von Dateien wird die Größe aber ermittelt. Bei Dateien kann auch das zweite Icon verwendet werden. Versuchen Sie einmal, auch oder nur die Dateien eines Verzeichnisses anzeigen zu lassen. Zumindest bei der Anzeige von Dateien und Verzeichnissen werden Sie feststellen, daß die Sortierung mißlingt.

Implementieren Sie die Umschaltung der Ansicht in alle vier möglichen Varianten mit Hilfe von Radio-Buttons wie in Abbildung 10.7 dargestellt.

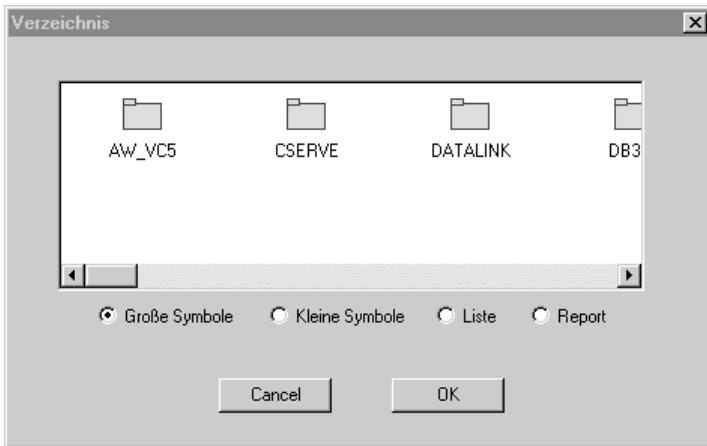


Abbildung 10.7: Der Dialog mit Radio-Buttons zum Umschalten der Ansicht

Bei einem Druck auf einen der vier Buttons wird die Funktion `OnStyle` aufgerufen, die mit dem Klassen-Assistenten für die Radio-Buttons in Verbindung mit der Nachricht `BN_CLICKED` angelegt wurde.

```
void CListTest::OnStyle()
{
    long lOldStyle;
    UpdateData(TRUE);
    //Window-Handle des Controls holen
    HWND hwnd = ((CListCtrl*)GetDlgItem(IDC_LIST))->m_hWnd;
    //Control-Styles abfragen
    lOldStyle = GetWindowLong(hwnd, GWL_STYLE);
    //alle Style-Attribute löschen
    lOldStyle &= (~LVS_ICON & ~LVS_SMALLICON & ~LVS_LIST &
~LVS_REPORT);
    //neue Attribute setzen
```

```

switch      (GetCurrentMessage() ->wParam)
{
    case IDC_RADIO1:101dStyle |= LVS_ICON;
        break;
    case IDC_RADIO2:101dStyle |= LVS_SMALLICON;
        break;
    case IDC_RADIO3:101dStyle |= LVS_LIST;
        break;
    case IDC_RADIO4:101dStyle |= LVS_REPORT;
        break;
}
//und Aktualisieren
SetWindowLong(hwnd, GWL_STYLE, 101dStyle);
UpdateData(FALSE);
}

```

Listing: Umschalten der Ansichtsarten eines Listenelements

In der Funktion werden zuerst alle vier möglichen Styles zurückgesetzt, und anschließend wird anhand des gedrückten Buttons der neue Style gesetzt. Das ist schon alles.



Die zweite Aufgabe besteht darin, bei einem Mausklick auf eines der Elemente im Listenelement den Namen des Verzeichnisses auszulesen und in einer Eingabezeile darzustellen.

Zur Lösung dieser Aufgabe ist es nötig, für die Nachricht `LVN_ITEMCHANGED` eine Behandlungsfunktion zu implementieren. Diese Nachricht wird immer dann verschickt, wenn sich der aktuell ausgewählte Eintrag des Listenelements ändert. Der Nachricht wird ein Zeiger auf eine Struktur mitgegeben, die die Nummer des Eintrags sowie des Untereintrags enthält. Mit der Methode *GetItemText* läßt sich so der Name des Verzeichnisses leicht ermitteln.

Das Beispiel kann noch erweitert werden: So ist es denkbar, eine universelle Klasse zum Ermitteln eines Verzeichnisses zu erstellen. Dazu müßte das Wechseln des Verzeichnisses auf Doppelklick noch implementiert werden. Die Anwendungsfälle für Listenelemente sind sehr vielseitig.

10.13 Animation (Animate Control)

10.13.1 Anwendung

Das Aminate Control ist jedem Anwender seit Windows 95 in Form kleiner Animationen bekannt, zum Beispiel beim Kopieren von Dateien. Da diese Steuerung standardmäßig in den Common Controls enthalten ist,

kann sie auch auf einfache Weise in eigene Programme eingebunden werden. Sie muß wie alle anderen Steuerungen auch nur in einen Dialog mit dem Ressource-Editor eingefügt werden.

10.13.2 Attribute

Außer den Standard- und den erweiterten Eigenschaften besitzt eine Animation weitere Attribute, die in Tabelle 10.25 zusammengefaßt sind.

Dialogeditor	Dialogboxschablone	Bedeutung
Zentriert (Center)	Ressource-Typ: CONTROL Style: ACS_CENTER	Mit diesem Attribut wird festgelegt, daß die Animation innerhalb des Rahmens der Steuerung zentriert wird.
Transparent	Ressource-Typ: CONTROL Style: ACS_TRANSPARENT	Durch diesen Style wird der Hintergrund der Animation transparent.
Auto-Wiedergabe (Autoplay)	Ressource-Typ: CONTROL Style: ACS_AUTOPLAY	Ist Autoplay aktiviert, wird die Animation automatisch und in einer Endlosschleife abgespielt.
Rand (Border)	Ressource-Typ: CONTROL Style: WS_BORDER	Um die Steuerung mit einer Umrandung zu versehen, muß dieses Attribut aktiviert werden.

Tabelle 10.25: Attribute des Animate Controls

10.13.3 Die Klasse *CAnimateCtrl*

Die Zahl der Methoden der Klasse *CAnimateCtrl* ist recht begrenzt. Man benötigt aber auch nicht viel, um die Animation zu steuern. Voraussetzung ist nur eine vorhandene Animation im AVI-Dateiformat.

Open und Close

Mit der Methode *Open* wird eine AVI-Datei geöffnet und das erste Bild der Animation angezeigt. Falls das Attribut *Auto-Wiedergabe* gesetzt ist, läuft die Animation sofort los. Als Parameter wird entweder der Dateiname der AVI-Datei angegeben oder die ID der Ressource. Wurde die Ressource-ID verwendet und anschließend NULL übergeben, wird das AVI-Video geschlossen.

```
class CAnimateCtrl:
    BOOL Open( LPCTSTR lpszFileName );
    BOOL Open( UINT nID );
    BOOL Close( );
```

Das Schließen einer Animationsdatei erfolgt mit der Methode *Close*. Eine Besonderheit ist beim Umgang mit den Animationen zu beachten: Wird das Attribut *Auto-Wiedergabe* verwendet, so muß es vor dem Öffnen der Animationsdatei gesetzt sein. Ansonsten muß diese zuerst mit *Close* geschlossen und dann wieder geöffnet werden.

Play und Stop

Um eine mit *Open* geöffnete Animation zu starten, wird die Methode *Play* aufgerufen. Ihr wird übergeben, von welchem bis zu welchem Frame der Animation das Video abgespielt werden soll. Zudem kann noch die Anzahl der Wiederholungen angegeben werden. Wird für *nFrom* der Wert Null angegeben, beginnt das Abspielen mit dem ersten Frame. Soll das Video bis zum letzten Bild gespielt werden, gibt man einfach -1 für *nTo* an. Enthält *nRep* ebenfalls -1, läuft das AVI endlos. Das Anhalten einer Animations-Steuerung erfolgt mit der Methode *Stop*.

```
class CAnimateCtrl:
    BOOL Play( UINT nFrom, UINT nTo,UINT nRep );
    BOOL Stop( );
```

10.13.4 Beispielprogramm

Das Programm CONTROLS wurde um einen Animations-Dialog ergänzt. Zwei Schaltflächen (PLAY und STOP) ermöglichen das Starten und Anhalten der Animation. Wird die Checkbox »Autoplay« aktiviert, so wird die AVI-Datei zuerst entladen, dann der Style gesetzt und schließlich das AVI neu geladen.



Das Problem der meisten Entwickler dürfte das Erstellen eigener AVI-Videos sein. Hier hilft ein kleines Shareware-Programm auf der CD.

10.14 Datum/Uhrzeit-Steuerung

10.14.1 Anwendung

Diese Steuerung erlaubt es, komfortabel und standardisiert Datum oder Uhrzeit vom Benutzer eingeben zu lassen.

In Abbildung 10.8 sind verschiedene Formate der Steuerung dargestellt: Links oben wurden das Langformat und das Drehfeld als Optionen eingestellt; rechts daneben befindet sich die Steuerung im Kurzformat mit geöffnetem Kalender zur Auswahl des Datums. In der Abbildung links unten ist das Zeitformat eingestellt, das nur mit dem Drehfeld dargestellt werden kann.



Abbildung 10.8: Datum/Uhrzeit-Steuerung in verschiedenen Varianten

Über spezielle Benachrichtigungs-codes ist es auch möglich, eigene Formate zu spezifizieren. Weitere Informationen dazu finden Sie in der Online-Hilfe unter »Verwenden von Rückruffeldern in einem Steuerelement für die Datums- und Zeitauswahl«.



10.14.2 Attribute

Außer den Standard- und den erweiterten Eigenschaften besitzt eine Datum/Uhrzeit-Steuerung weitere Attribute, die in Tabelle 10.26 zusammengefaßt sind.

Dialogeditor	Dialogboxschablone	Bedeutung
Rechtsbündig (Right Align)	Ressource-Typ: CONTROL Style: <i>DTS_RIGHTALIGN</i>	Mit diesem Attribut wird festgelegt, daß der Kalender rechtsbündig zur Steuerung aufgeklappt wird, wenn die Schaltfläche betätigt wird. Wird die Option <i>Drehfeld</i> gewählt, ist rechtsbündig unwirksam.
Drehfeld-Steuerelement verwenden (Use Spin Control)	Ressource-Typ: CONTROL Style: <i>DTS_UPDOWN</i>	Mit Hilfe der Option <i>Drehfeld</i> ändert sich die Schaltfläche zum Anzeigen des Kalenders in ein Drehfeld. Die Änderung des Datums bzw. der Uhrzeit erfolgt dann darüber oder über die Cursortasten. Jeder Teil des Datums bzw. der Uhrzeit kann eigenständig geändert werden.
Format	Ressource-Typ: CONTROL Style: <i>DTS_LONGDATEFORMAT</i> <i>DTS_SHORTDATEFORMAT</i> <i>DTS_TIMEFORMAT</i>	Zur Wahl des Formats von Datum bzw. Uhrzeit existieren drei Varianten: <i>Kurzes Datum</i> (dd.mm.jj), <i>Langes Datum</i> (ttt, dd. mmm jjjj), <i>Zeit</i> (hh:mm:ss)

Tabelle 10.26: Attribute der Datum/Uhrzeit-Steuerung

Dialogeditor	Dialogboxschablone	Bedeutung
Bearbeitung zulassen (Allow Edit)	Ressource-Typ: CONTROL Style: <i>DTS_APPCANPARSE</i>	Ist diese Option aktiviert, kann der Benutzer mit der Taste F2 den Inhalt der Steuerung markieren und überschreiben.
Nichts anzeigen (Show None)	Ressource-Typ: CONTROL Style: <i>DTS_SHOWNONE</i>	Ist diese Option gewählt, wird ein Kontrollkästchen vor Datum/Uhrzeit angezeigt. Damit wird es möglich, der Steuerung mitzuteilen, daß kein Datum bzw. keine Uhrzeit gewählt wurde, wenn das Häkchen entfernt wird.

Tabelle 10.26: Attribute der Datum/Uhrzeit-Steuerung

10.14.3 Benachrichtigungscodes

Einige spezielle Benachrichtigungscodes der Datum/Uhrzeit-Steuerung werden in diesem Abschnitt beschrieben.

DTN_CLOSEUP wird gesendet, wenn der geöffnete Kalender geschlossen werden soll. Diese Nachricht kann nicht gesendet werden, wenn der Style »Drehfeld« aktiviert ist. Die Nachricht DTN_DROPDOWN ist das Gegenstück dazu; sie wird geschickt, wenn der Kalender angezeigt werden soll. Mit Hilfe dieser Benachrichtigungen ist es möglich, die in die Datum/Uhrzeit-Steuerung eingebettete Kalender-Steuerung zu modifizieren (z.B. Farbe oder Font).

Die Nachricht DTN_DATETIMECHANGE wird an das Eltern-Fenster geschickt, wenn Datum oder Uhrzeit geändert wurden. In diesem Fall kann hier eine eigene Verarbeitung, wie zum Beispiel eine Plausibilitätsprüfung, erfolgen.

DTN_FORMAT und DTN_FORMATQUERY dienen zur Behandlung von benutzerdefinierten Formaten der Datum/Uhrzeit-Steuerung. DTN_WMKEYDOWN wird gesendet, wenn der Benutzer Eingaben in Callback-Feldern (Rückruffelder) vorgenommen hat. Weitere Informationen zum Einfügen und Behandeln von Callback-Feldern sind in der Online-Hilfe zu finden.

Wurde der Style »Bearbeitung zulassen« aktiviert, wird die Nachricht DTN_USERSTRING geschickt, sobald der Benutzer das Editieren der Werte beendet hat.

10.14.4 Die Klasse *CDateTimeCtrl*

Die Klasse *CDateTimeCtrl* ermöglicht dem Programmierer, die Steuerung in gewissen Grenzen an eigene Bedürfnisse anzupassen; dazu dienen die nachfolgend beschriebenen Member-Funktionen.

SetMonthCalColor und GetMonthCalColor

Diese beiden Funktionen ermöglichen die Farbanpassung des eingebetteten Monatskalenders. Der Monatskalender wird über ein Drop-Down-Feld wie bei einem Kombinationsfeld aufgeblendet, jedoch nur, wenn die Option »Drehfeld-Steuer-element benutzen« nicht aktiviert ist. Für einige Bereiche des Monatskalenders läßt sich die Farbe anpassen. Um die Bereiche anzusprechen, sind die in Tabelle 10.27 aufgeführten IDs zu benutzen.

Wert	Bedeutung
MCSC_BACKGROUND	Hintergrundfarbe zwischen den Monaten
MCSC_MONTHBK	Hintergrundfarbe innerhalb eines Monats
MCSC_TEXT	Textfarbe innerhalb eines Monats
MCSC_TITLEBK	Hintergrundfarbe des Titels
MCSC_TITLETEXT	Textfarbe des Titels
MCSC_TRAILINGTEXT	Textfarbe der Tage des Vor- bzw. Folgemonats, die im aktuellen Monat mitangezeigt werden

Tabelle 10.27: Bereichs-Bezeichner zur Farbänderung des Monatskalenders

Die Funktion *GetMontCalColor* liefert die Farbe des in *iColor* angegebenen Bereiches zurück. Mit Hilfe von *SetMonthCalColor* wird die Farbe eines Bereiches auf den Wert *ref* gesetzt. Gleichzeitig liefert die Funktion den vorhergehenden Wert der Farbe zurück. Bei beiden Funktionen bedeutet ein Rückgabewert von -1, daß das Ausführen der Funktion mißlungen ist.

```
class CDateTimeCtrl:
    COLORREF SetMonthCalColor( int iColor, COLORREF ref );
    COLORREF GetMonthCalColor( int iColor ) const;
```

Den Farbwert kann man sehr einfach über das Makro *RGB* bilden (siehe Kapitel 6.2.1).

Um die Hintergrundfarbe des Titels auf Rot zu setzen, kann folgende Befehlszeile verwendet werden:

```
m_DT2.SetMonthCalColor(MCSC_TITLEBK, RGB(255,0,0));
```

Bei *m_DT2* handelt es sich um die Member-Variable, die für die Steuerung mit dem Klassen-Assistenten angelegt wurde. Die RGB-Werte bewegen sich im Bereich von 0 bis 255. Der Rot-Anteil von 255 entspricht dabei 100%.



GetMonthCalFont und SetMonthCalFont

Auch der Zeichensatz des Monatskalenders läßt sich ermitteln bzw. ändern. Dies wird mit Hilfe der Member-Funktionen *GetMonthCalFont* und *SetMonthCalFont* erledigt. *GetMonthCalFont* liefert einen Zeiger auf ein temporäres Cfont-Objekt.

```
class CDateTimeCtrl:
    CFont* GetMonthCalFont( ) const;
    void SetMonthCalFont( HFONT hFont, BOOL bRedraw = TRUE );
```

Mit Hilfe von *SetMonthCalFont* wird ein Handle vom Typ `HFONT` übergeben, das den neu anzuwendenden Zeichensatz enthält. Der Parameter `bRedraw` teilt der Steuerung mit, daß der neue Font unmittelbar angewendet werden soll (Wert `TRUE`).



Die `HFONT`-Struktur im GDI wird der Klasse *CFont* der MFC zugeordnet. Die Klasse besitzt einen Operator *HFONT*, der das Handle auf die Struktur liefert. Deshalb ist der folgende Konstrukt möglich:

```
CFont mFont = CreateFont( ... );
HFONT hF = (HFONT) mFont;
```

SetFormat

Um nicht auf eines der drei Formate der Steuerung angewiesen zu sein, kann mit der Funktion *SetFormat* ein eigenes Format für die Datum/Uhrzeit-Steuerung festgelegt werden. Das Format wird in Form einer nullterminierten Zeichenkette übergeben.

```
class CDateTimeCtrl:
    BOOL SetFormat( LPCTSTR pstrFormat );
```

Die Zeichenkette enthält Formatspezifikationen entsprechend Tabelle 10.28.

Zeichenkette	Ausgabe	Beispiel
„d«	Tag ohne Vornull	9, 19
„dd«	Tag immer zweistellig, ggf. mit Vornull	09, 19
„ddd«	gekürzter Name des Wochentages (drei Zeichen)	Mon, Son
„dddd«	voller Name des Wochentages	Montag, Sonntag
„h«	Stunde im 12-Stunden-Format ohne Vornull	1, 10
„hh«	Stunde im 12-Stunden-Format, ggf. mit Vornull	01, 10

Zeichenkette	Ausgabe	Beispiel
„H«	Stunde im 24-Stunden-Format ohne Vornull	1, 14
„HH«	Stunde im 24-Stunden-Format, ggf. mit Vornull	01, 14
„m«	Minuten ohne Vornull	1, 48
„mm«	Minuten immer zweistellig, ggf. mit Vornull	01, 48
„M«	Monat ohne Vornull	1, 12
„MM«	Monat immer zweistellig, ggf. mit Vornull	01, 12
„MMM«	gekürzter Name des Monats (drei Zeichen)	Jan, Dez
„MMMM«	voller Name des Monats	Januar, Dezember
„t«	Abkürzung für AM bzw. PM beim 12-Stunden-Format	A, P
„tt«	Anzeige von AM bzw. PM beim 12-Stunden-Format	AM, PM
„X«	Festlegung von Callback-Feldern (Rückruffelder), um eigene Felder einzufügen Der Programmierer muß die Nachrichten DTN_WMKEYDOWN, DTN_FORMAT, und DTN_FORMATQUERY behandeln. Mehrere Callback-Felder können durch unterschiedliche Anzahl der Zeichen erzeugt werden.	X, XX, XXX ...
„y«	Jahreszahl einstellig (letzte Ziffer)	8 für 1998
„yy«	die letzten beiden Stellen der Jahreszahl	98 für 1998
„yyy«	Jahreszahl vierstellig	1998

Tabelle 10.28: Formatfelder der Datum/Uhrzeit-Steuerung

Die Code-Zeile

```
m_DT4.SetFormat(„'+++ ',dddd', ,dd'. ,MMM yy' '+++'“);
```

zum Setzen des benutzerdefinierten Formats bewirkt folgende Ausgabe:

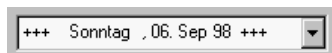


Abbildung 10.9: Datum im benutzerdefinierten Format

GetRange und SetRange

Für die Datum/Uhrzeit Steuerung kann ein Gültigkeitsbereich festgelegt werden, der den Bereich der einzugebenden Daten einschränkt. Mit Hilfe von *GetRange* kann dieser Bereich abgefragt und mit *SetRange* neu definiert werden.

```

Class CdateTimeCtrl:
    DWORD GetRange( ColeDateTime* pMinRange,
                    ColeDateTime* pMaxRange ) const;
    DWORD GetRange( Ctime* pMinRange, Ctime* pMaxRange ) const;
    BOOL SetRange(  const ColeDateTime* pMinRange,
                    const ColeDateTime* pMaxRange );
    BOOL SetRange(  const Ctime* pMinRange, const Ctime* pMaxRange
    );

```

```

ColeDateTime pDT5, pDT6;
...
m_DT4.SetRange(&pDT5, &pDT6);

```

Die beiden Code-Zeilen zeigen die Definition von Variablen sowie das Setzen eines Gültigkeitsbereiches der Steuerung, die als `m_DT4` deklariert wurde.

GetMonthCalCtrl

Diese Member-Funktion liefert einen Zeiger auf das Monatskalender-Steuererelement, das als Kindfenster erzeugt wird, wenn der Benutzer die Auswahl Schaltfläche betätigt. Zu beachten ist dabei, daß der Zeiger nur gültig ist, wenn der Monatskalender angezeigt wird, d.h. zwischen den Nachrichten `DTN_DROPDOWN` und `DTN_CLOSEUP`.

```

class CDateTimeCtrl:
    CMonthCalCtrl* GetMonthCalCtrl( ) const;

```

Soll eine Veränderung an den Eigenschaften des Monatskalenders erfolgen, so läßt sich das am besten in der Nachrichtenbehandlungsroutine von `DTN_DROPDOWN` erledigen.

GetTime und SetTime

Die eigentlich wichtigsten Member-Funktionen sind *GetTime* und *SetTime* zum Lesen und Setzen von Datum bzw. Uhrzeit. Die Ergebnisse der Benutzereingaben werden mit *GetTime* aus der Steuerung ausgelesen.

```

class CDateTimeCtrl
    BOOL GetTime( COleDateTime& timeDest ) const;
    DWORD GetTime( CTime& timeDest ) const;
    DWORD GetTime( LPSYSTEMTIME pTimeDest ) const;
    BOOL SetTime( const COleDateTime& timeNew );
    BOOL SetTime( const CTime* pTimeNew );
    BOOL SetTime( LPSYSTEMTIME pTimeNew = NULL );

```

Als Format des Rückgabewertes in `timeDest` bzw. `pTimeDest` stehen drei Möglichkeiten zur Verfügung: `ColeDateTime`, `Ctime` oder die `LPSYSTEM-TIME`-Struktur. Ebenso verhält es sich beim Setzen der Zeit mit `SetTime`.

10.14.5 Beispielprogramm

Das Programm `CONTROLS` wurde um einen weiteren Dialog zur Datum-/Uhrzeit-Auswahl ergänzt. Dieser zeigt verschiedene Möglichkeiten der Steuerung auf.

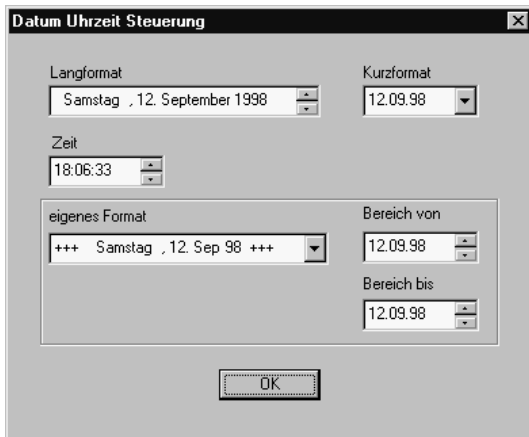


Abbildung 10.10: Dialog mit Datum/Uhrzeit-Steuerelementen

Im oberen Teil sind die Steuerungen in den drei Standard-Formaten dargestellt. Die zweite Steuerung *Kurzformat* zeigt, wie das eingebettete Monatskalender-Steuerelement beeinflusst werden kann. Zunächst werden im Klassen-Assistenten Member-Variablen für die Steuerungen angelegt.

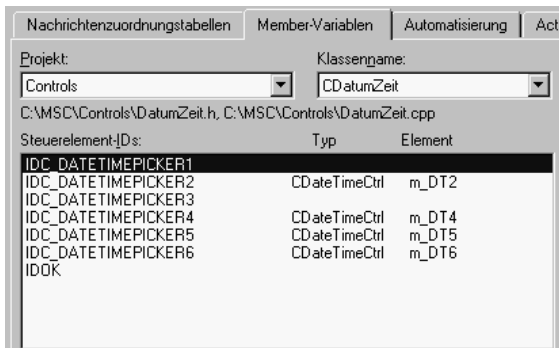


Abbildung 10.11: Anlegen von Member-Variablen im Klassen-Assistenten

```

BOOL CDatumZeit::OnInitDialog()
{
    CDialog::OnInitDialog();
    m_pMonthFont = new CFont;
    BOOL bOk = m_pMonthFont->CreateFont(-24,0,0,0,700,1,0,0,
                                        ANSI_CHARSET,
                                        OUT_DEFAULT_PRECIS,
                                        CLIP_DEFAULT_PRECIS,
                                        DEFAULT_QUALITY,
                                        DEFAULT_PITCH | FF_SWISS,
                                        NULL);

    if (bOk)
        m_DT2.SetMonthCalFont((HFONT)*m_pMonthFont);
    else
        m_pMonthFont = NULL;
    m_DT2.SetMonthCalColor(MCSC_TEXT, RGB(0,0,255));
    m_DT2.SetMonthCalColor(MCSC_TITLEBK, RGB(255,0,0));
    m_DT2.SetMonthCalColor(MCSC_TITLETEXT, RGB(0,0,0));
    m_DT4.SetFormat(„'+++  ,ddd', ,dd'. ,MMM yy'  +++'“);
    return TRUE; // return TRUE unless you set the focus to a
control
    // EXCEPTION: OCX-Eigenschaftenseiten sollten FALSE
zurückgeben
}

```

Listing: Verschiedene Initialisierungen des Monatskalenders

Die Änderung der Eigenschaften erfolgt in der Methode *OnInitDialog*, die beim Initialisieren des Dialogs vor der Darstellung auf dem Bildschirm abgearbeitet wird. Zunächst wird ein neuer Zeichensatz erzeugt (siehe Kapitel 7.3.3) und anschließend mit *SetMonthCalFont* aktiviert. Des weiteren werden für einige Elemente des Kalenders neue Farben gesetzt.



In der Header-Datei der Klasse wurde eine Variable *m_pMonthFont* definiert, die im Konstruktor initialisiert und im Destruktor wieder zerstört wird.

Im unteren Teil von Abbildung 10.10 wird zunächst die Methode *SetFormat* benutzt, um ein eigenes Format zur Darstellung in der Steuerung zu nutzen. Zwei weitere Standard-Steuerelemente dienen zur Festlegung eines Gültigkeitsbereiches der Datum/Uhrzeit-Steuerung mit dem eigenen Format. Der Bereich wird mit der Methode *SetRange* erstmals gesetzt, wenn eines der beiden Steuerelemente zur Festlegung des Bereiches geändert wurde. Fortan sind nur noch Daten wählbar, die innerhalb des Bereiches liegen. Änderungen werden anhand der Nachricht *DTN_DATETIMECHANGE* erkannt. Die zugehörige Nachrichtenbehandlungsroutine *OnDateTimeChange* wird mit dem Klassen-Assistenten für beide Steuerungen angelegt.


```

void CDatumZeit::OnDatetimechange(NMHDR* pNMHDR, LRESULT* pResult)
{
    COleDateTime pDT4, pDT5, pDT6;
    //Werte der Steuerungen auslesen
    m_DT4.GetTime(pDT4);
    m_DT5.GetTime(pDT5);
    m_DT6.GetTime(pDT6);
    //Bereich von muß kleiner/gleich sein als Bereich bis
    if (pDT5<=pDT6)
    {
        //wenn DT4 nicht im Bereich -> auf Wert im Bereich setzen
        if (pDT5>pDT4)
            m_DT4.SetTime(pDT5);
        if (pDT6<pDT4)
            m_DT4.SetTime(pDT6);
        //Bereich setzen
        m_DT4.SetRange(&pDT5, &pDT6);
    }
    *pResult = 0;
}

```

Listing: Nachrichtenbehandlung bei Änderung des Datums

Als erstes werden mit der Methode *GetTime* die aktuellen Werte aller beteiligten Steuerungen ausgelesen. (pDT4 beinhaltet das zu begrenzende Element, pDT5 die untere Bereichsgrenze und pDT6 die obere Bereichsgrenze) Nach diversen Prüfungen wird schließlich mit *SetRange* der Gültigkeitsbereich festgelegt. Es wird gewährleistet, daß durch die Änderung des Wertebereichs der Wert der Steuerung nicht außerhalb des Bereichs liegt.

Im Beispielprogramm wird auch der Zugriff auf das eingebettete Monatskalender-Steurelement demonstriert. Wie bereits erwähnt, ist der Zeiger darauf, der mit *GetMonthCalCtrl* ermittelt wird, nur solange gültig, wie der Kalender angezeigt wird. Deshalb ist der günstigste Zeitpunkt zum Zugriff darauf der Moment, wenn der Kalender angezeigt werden soll, der Benutzer also die Auswahl Schaltfläche betätigt hat. In diesem Moment wird die Nachricht DTN_DROPDOWN geschickt, die mit der Nachrichtenbehandlungsfunktion *OnDropdownDatetimestpicker2* verbunden wird.

```

void CDatumZeit::OnDropdownDatetimestpicker2(NMHDR* pNMHDR,
                                              LRESULT* pResult)
{
    (m_DT2.GetMonthCalCtrl()->SetFirstDayOfWeek(6);
    *pResult = 0;
}

```

Listing: Setzen des ersten Wochentages des eingebetteten Monatskalenders

Hier wird schon ein wenig auf den nächsten Abschnitt vorgegriffen: mit der Methode *SetFirstDayOfWeek* der Monatskalender-Steuerung wird der erste Tag der Woche auf Sonntag gesetzt.

10.15 Die Monatskalender-Steuerung

10.15.1 Anwendung

Die Monatskalender-Steuerung ermöglicht dem Benutzer die Anzeige eines Kalenderblatts und die Auswahl eines oder mehrerer Tage darin. Standardmäßig wird der aktuelle Tag angezeigt und besonders hervorgehoben.

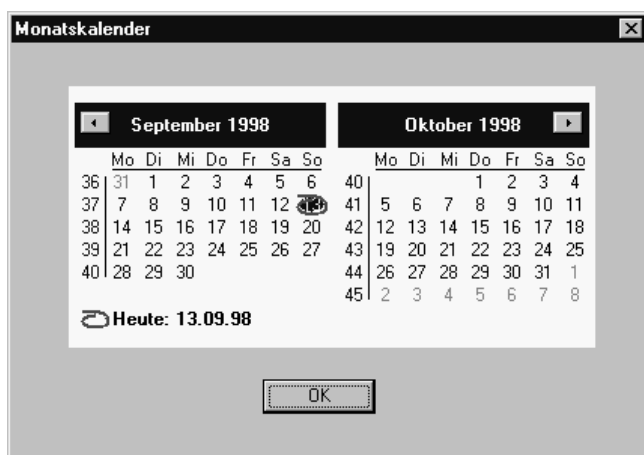


Abbildung 10.12: Ansicht einer Monatskalender-Steuerung im Dialog

Die Größe der Steuerung bestimmt, wie viele Monate im Kalender dargestellt werden. In Abbildung 10.12 wurde die Größe auf zwei Monate angepasst. Beim Blättern über die Pfeiltasten oben links und rechts werden dann immer zwei Monate vor- bzw. zurückgeblättert. Ein Mausklick auf *Heute: ...* veranlaßt die Steuerung, das Kalenderblatt mit dem heutigen Datum darzustellen. Des weiteren ist es möglich, mehrere Tage auszuwählen oder bestimmte Tage fett hervorzuheben.

10.15.2 Attribute

Außer den Standard- und den erweiterten Eigenschaften besitzt eine Monatskalender-Steuerung weitere Attribute, die in Tabelle 10.29 zusammengefaßt sind.

Dialogeditor	Dialogboxschablone	Bedeutung
Tagesstatus (Day States)	Ressource-Typ: CONTROL Style: <i>MCS_DAYSTATE</i>	Mit diesem Attribut wird festgelegt, daß der Kalender bestimmte Tage fett hervorgehoben darstellen kann.
Mehrfachauswahl (Multi Select)	Ressource-Typ: CONTROL Style: <i>MCS_MULTISELECT</i>	Soll der Benutzer mehr als einen Tag gleichzeitig wählen können, so ist dieses Attribut zu setzen. Die Anzahl der maximal wählbaren Tage kann über eine Member-Funktion festgelegt werden.
Kein Heute (No Today)	Ressource-Typ: CONTROL Style: <i>MCS_NOTODAY</i>	Dieses Attribut bestimmt, daß am unteren Rand der Steuerung die Zeile <i>Heute: ...</i> nicht angezeigt wird. Es bleibt jedoch im Kalenderblatt die Markierung des heutigen Tages erhalten.
Keine Markierung für Heute (No Today Circle)	Ressource-Typ: CONTROL Style: <i>MCS_NOTODAYCIRCLE</i>	Dieses Attribut bestimmt, daß das heutige Datum nicht gesondert hervorgehoben wird.
Kalenderwochen (Week Numbers)	Ressource-Typ: CONTROL Style: <i>MCS_WEEKNUMBERS</i>	Ist diese Option gewählt, wird zusätzlich im Kalender die Nummer der Woche angezeigt.

Tabelle 10.29: Attribute der Monatskalender-Steuerung

10.15.3 Benachrichtigungscodes

Das Monatskalender-Steuerelement sendet einige Nachrichten auf bestimmte Eingaben des Benutzers hin. Diese Nachrichten können dann in eigenen Behandlungsroutinen verarbeitet werden.

Die MCN_GETDAYSTATE-Nachricht wird verwendet, um die Hervorhebung bestimmter Tage zu ermöglichen. Sie fordert das Programm auf, die Tage zu markieren. Die Anwendung dieser Nachricht wird im Beispielprogramm demonstriert. Sie wird immer dann geschickt, wenn das Kalenderblatt neu gezeichnet werden muß, zum Beispiel nach dem Weiterblättern der Monate.

Wird die Nachricht MCN_SELCHANGE gesendet, bedeutet dies, daß der Benutzer ein neues Datum oder einen anderen Datumsbereich gewählt hat.

Während die Nachricht MCN_SELCHANGE bei jeder Änderung der Auswahl geschickt wird, also beispielsweise beim Blättern durch die Monate, wird MCN_SELECT nur bei einer expliziten Auswahl eines oder mehrerer Tage gesendet. Auch ein Klick auf *Heute:...* veranlaßt zum Senden von MCN_SELECT. Oftmals treten beide Nachrichten auch in Kombination auf.

10.15.4 Die Klasse *CMonthCalCtrl*

GetMonthDelta und SetMonthDelta

Die Funktion *GetMonthDelta* ermittelt die Anzahl der zu scrollenden Monate, wenn der Benutzer die Schaltflächen zum Blättern der Monate betätigt. Mittels *SetMonthDelta* kann der Wert neu gesetzt werden. Die Funktion kann sehr sinnvoll eingesetzt werden, wenn der Kalender mehrere Monate anzeigt und nicht um die gesamte Anzahl der angezeigten Monate gescrollt werden soll.

```
class CMonthCalCtrl:
    int GetMonthDelta( ) const;
    int SetMonthDelta( int iDelta );
```



Wird der Wert 0 als Scrollrate zurückgegeben, heißt das, daß diese nicht gesetzt wurde und die Standardeinstellung von der Anzahl der dargestellten Monate benutzt wird.

GetFirstDayOfWeek und SetFirstDayOfWeek

Mittels der Funktion *GetFirstDayOfWeek* wird der erste Tag der Woche (in der linken Spalte des Kalenders) ermittelt. Er wird als Integer-Wert zurückgegeben, dessen Bedeutung in Tabelle 10.30 zu sehen ist. Des weiteren wird ein Zeiger auf einen BOOL-Wert *pbLocal* übergeben. Dieser Wert enthält Null, wenn der erste Tag der Woche mit den Ländereinstellungen der Systemsteuerung übereinstimmt.

```
class CMonthCalCtrl:
    int GetFirstDayOfWeek( BOOL* pbLocal = NULL ) const;
    BOOL SetFirstDayOfWeek( int iDay, int* lpnOld = NULL );
```

SetFirstDayOfWeek kann einen neuen Tag als ersten Wochentag einstellen. In *iDay* wird dazu der Wert des Tages übergeben. Über einen Zeiger auf den Integer-Wert *lpnOld* kann der vorher eingestellte Tag ermittelt werden. Der Rückgabewert der Funktion entspricht *pbLocal* der Methode *GetFirstDayOfWeek*.

int-Wert	Wochentag
0	Montag
1	Dienstag
2	Mittwoch
3	Donnerstag

int-Wert	Wochentag
4	Freitag
5	Samstag
6	Sonntag

Tabelle 10.30: Zuordnung der Werte zu Wochentagen

GetColor und SetColor

Die Funktionen *GetColor* und *SetColor* entsprechen den schon bei der Datum/Uhrzeit-Steuerung beschriebenen Funktionen *Set/GetMonthCalColor*. Auch hier können die Farben der einzelnen Bereiche angepaßt werden. Die zugehörigen Parameter zu *nRegion* sind in Tabelle 10.27 zu finden.

```
class CMonthCalCtrl:
    COLORREF SetColor( int nRegion, COLORREF ref );
    COLORREF GetColor( int nRegion ) const;
```

GetToday und SetToday

GetToday liefert den Tag als Datum in drei verschiedenen Varianten, der als heutiger Tag gilt. Mit Hilfe von *SetToday* läßt sich dieser Tag auf ein individuelles Datum setzen. Das Datum von *GetToday* ist nur gültig, wenn die Funktion TRUE zurückgibt.

```
class CMonthCalCtrl:
    BOOL CMonthCalCtrl::GetToday( COleDateTime& refDateTime ) const;
    BOOL CMonthCalCtrl::GetToday( CTime& refDateTime ) const;
    BOOL GetToday( LPSYSTEMTIME pDateTime ) const;
    void SetToday( const COleDateTime& refDateTime );
    void SetToday( const CTime* pDateTime );
    void SetToday( const LPSYSTEMTIME pDateTime );
```

Die folgenden beiden Zeilen setzen den heutigen Tag auf den 12.9.1998, 18 Uhr:

```
CTime cToday(1998,9,12,18,0,0);
m_MK.SetToday(&cToday);
```

Die Konstruktoren von *CTime* bzw. von *COleDateTime* bieten eine Reihe von Möglichkeiten zur Initialisierung. Für Konstanten bietet sich die oben dargestellte Lösung an.

GetCurSel und SetCurSel

Unabhängig vom Setzen des heutigen Tages bleibt der ausgewählte Tag in der Monatskalender-Steuerung derjenige mit dem aktuellen Datum, wenn nicht zuvor mit Hilfe von *SetCurSel* ein anderer festgelegt wurde. Auch

hier bestehen die gleichen Möglichkeiten zur Übergabe des Datums in den bekannten drei Formen. Die Funktion *GetCurSel* ermittelt das ausgewählte Datum. Sie kann dazu verwendet werden, den vom Benutzer gewählten Tag beim Verlassen des Dialogs auszulesen, wenn keine Mehrfachauswahl zugelassen wurde.

```
class CMonthCalCtrl:
    BOOL SetCurSel( const COleDateTime& refDateTime );
    BOOL SetCurSel( const CTime& refDateTime );
    BOOL SetCurSel( const LPSYSTEMTIME pDateTime );
    BOOL GetCurSel( COleDateTime& refDateTime ) const;
    BOOL GetCurSel( CTime& refDateTime ) const;
    BOOL GetCurSel( LPSYSTEMTIME pDateTime ) const;
```

Die Funktionen wurden fehlerfrei ausgeführt, wenn der Rückgabewert ungleich Null ist.

GetRange und SetRange

Der Bereich, den der Benutzer zur Datumswahl nutzen kann, wird mit der Funktion *SetRange* festgelegt und kann mit *GetRange* ausgelesen werden. Das Datum kann wieder in den drei bekannten Formen angegeben werden. *SetRange* liefert einen Wert ungleich Null zurück, wenn das Setzen eines Gültigkeitsbereiches erfolgreich war.

```
class CMonthCalCtrl:
    DWORD GetRange( COleDateTime* pMinRange,
                    COleDateTime* pMaxRange ) const;
    DWORD GetRange( CTime* pMinRange, CTime* pMaxRange ) const;
    DWORD GetRange( LPSYSTEMTIME pMinRange,
                    LPSYSTEMTIME pMaxRange ) const;
    BOOL SetRange( const COleDateTime* pMinRange,
                  const COleDateTime* pMaxRange );
    BOOL SetRange( const CTime* pMinRange, const CTime* pMaxRange );
    BOOL SetRange( const LPSYSTEMTIME pMinRange,
                  const LPSYSTEMTIME pMaxRange );
```

Der Rückgabewert von *GetRange* muß dagegen detailliert ausgewertet werden. Die möglichen Werte sind in Tabelle 10.31 zusammengefaßt.

Wert	Bedeutung
0	es wurden keine Grenzen gesetzt.
GDTR_MIN	eine minimale Begrenzung wurde gesetzt und ist gültig.
GDTR_MAX	eine maximale Begrenzung wurde gesetzt und ist gültig.

Tabelle 10.31: Rückgabewerte von *GetRange*

GetMonthRange

Die Funktion *GetMonthRange* liefert als Rückgabewert die Anzahl der Monate, die im Kalender angezeigt werden. Mit Hilfe des Parameters *dwFlags* geschieht dies durch unterschiedliche Betrachtungsweisen: Wird der Wert *GMR_DAYSTATE* benutzt, werden auch die vorangehenden und die folgenden Monate mitgezählt; wird dagegen *GMR_VISIBLE* verwendet, werden nur die vollständig sichtbaren Monate ermittelt.

```
class CMonthCalCtrl:
    int GetMonthRange(COleDateTime& refMinRange,
                     COleDateTime& refMaxRange, DWORD dwFlags )
const;
    int GetMonthRange(CTime& refMinRange,
                     CTime& refMaxRange, DWORD dwFlags ) const;
    int GetMonthRange(LPSYSTEMTIME pMinRange, LPSYSTEMTIME
pMaxRange,
                     DWORD dwFlags ) const;
```

GetMaxSelCount und SetMaxSelCount

Bei Verwendung der Mehrfachauswahl von Tagen kann die Anzahl der maximal wählbaren Tage begrenzt werden. Mit der Funktion *GetMaxSelCount* kann der aktuell gesetzte Wert ermittelt und mit *SetMaxSelCount* neu definiert werden. Der Rückgabewert von *SetMaxSelCount* ist Null, wenn die Funktion nicht erfolgreich ausgeführt werden konnte.

```
class CMonthCalCtrl:
    int GetMaxSelCount( ) const;
    BOOL SetMaxSelCount( int nMax );
```

GetSelRange und SetSelRange

Um bei Mehrfachauswahl den vom Benutzer gewählten Bereich zu ermitteln bzw. eine Auswahl vorzugeben, werden die Funktionen *GetSelRange* und *SetSelRange* verwendet. Die Mehrfachauswahl erlaubt allerdings nur, einen zusammenhängenden Bereich von Tagen auszuwählen. Die Funktionen liefern einen Wert ungleich Null, wenn die Ausführung erfolgreich war.

```
class CMonthCalCtrl:
    BOOL GetSelRange( COleDateTime& refMinRange,
                     COleDateTime& refMaxRange ) const;
    BOOL GetSelRange( CTime& refMinRange, CTime& refMaxRange )
```

```
const;
    BOOL GetSelRange( LPSYSTEMTIME pMinRange,
                      LPSYSTEMTIME pMaxRange ) const;
    BOOL SetSelRange( const COleDateTime& pMinRange,
                      const COleDateTime& pMaxRange );
    BOOL SetSelRange( const CTime& pMinRange, const CTime& pMaxRange );
    BOOL SetSelRange( const LPSYSTEMTIME pMinRange,
                      const LPSYSTEMTIME pMaxRange );
```

SetDayState

Der Tagesstatus zum Hervorheben bestimmter Tage eines Monats wird mit der Methode *SetDayState* gesetzt. Dabei zeigt die Variable *pStates* auf ein Array vom Typ *MONTHDAYSTATE*, das *nMonth* Monate enthält. *MONTHDAYSTATE* implementiert einen 32-Bit-Datentyp, bei dem die Bits zur Repräsentation der Tage eines Monats dienen. Ein gesetztes Bit bedeutet ein hervorgehobener Tag im Monat. Im Beispielprogramm wird die Anwendung dieser Methode näher erläutert.

```
class CMonthCalCtrl:
    BOOL SetDayState( int nMonths, LPMONTHDAYSTATE pStates );
```

10.15.5 Beispielprogramm

Die Beispiele zum Monatskalender wurden als Dialog in das Programm *CONTROLS* eingebunden. Das Layout des Dialogs ist in Abbildung 10.13 zu sehen.

Im oberen Teil ist die Monatskalender-Steuerung mit zwei sichtbaren Monaten enthalten. Der Benutzer kann darin seine Auswahl treffen. Das Stichwort *GetMonthRange* zeigt das Ergebnis der Anwendung dieser Methode mit den zwei verschiedenen Parametern.

Der Parameter *MaxSelCount* ermöglicht das Eingeben eines Wertes, der die maximale Zahl der markierbaren Tage angibt. Neben *GetSelRange* wird schließlich der vom Benutzer gewählte Datumsbereich ausgegeben.

Weitere Einstellungen betreffen die Hintergrundfarbe, den heutigen Tag sowie das Hervorheben bestimmter Tage.

```
BOOL CMonatsKalender::OnInitDialog()
{
    CDialog::OnInitDialog();
    //immer nur um einen Monat blättern
    int nMon = m_MK.SetMonthDelta(1);
    TRACE("Scrollrate stand auf %i Monate\n",nMon);
```



```

//Hintergrund dunkelgrau
m_MK.SetColor(MCSC_BACKGROUND, RGB(100, 100, 100));
//Heute: 12.9.98, 18 Uhr
CTime cToday(1998,9,12,18,0,0);
m_MK.SetToday(&cToday);
//Tests mit FirstDayOfWeek
BOOL b;
m_MK.SetFirstDayOfWeek(6);
m_MK.GetFirstDayOfWeek(&b);
TRACE("FDOW: %i\n",b);
//max. Anzahl wählbarer Daten ermitteln und anzeigen
m_SelCount = m_MK.GetMaxSelCount();
//aktuell gewählten Bereich ermitteln und anzeigen
CTime minTime, maxTime;
if(m_MK.GetSelRange(minTime, maxTime))
{
    m_DT1 = minTime;
    m_DT2 = maxTime;
    UpdateData(FALSE);
}
return TRUE; // return TRUE unless you set the focus to a control
// EXCEPTION: OCX-Eigenschaftenseiten sollten FALSE
zurückgeben
}

```

Listing: OnInitDialog des Beispielprogramms

In der Methode *OnInitDialog* werden alle nötigen Initialisierungen vorgenommen. Zuvor wurden mit dem Klassen-Assistenten Member-Variablen angelegt. *OnInitDialog* selbst kann ebenfalls mit dem Klassen-Assistenten oder mit der Assistentenleiste angelegt werden.

Als erste Aktion erfolgt das Setzen der Scrollrate der Steuerung auf nur einen Monat. Standardmäßig wird um die Anzahl der dargestellten Monate weitergeblättert. Danach wird die Hintergrundfarbe zwischen den Monaten des Kalenders neu gesetzt. Allen weiteren Bereichen können hier ebenfalls neue Farben zugewiesen werden. Auch der Zeichensatz könnte geändert werden (siehe Kapitel 10.14.4).

Anschließend wird mit Hilfe der Methode *SetToday* der heutige Tag über ein *CTime*-Objekt neu definiert. Die folgenden Zeilen beschäftigen sich mit dem ersten Tag der Woche, der hier auf Sonntag gesetzt wird. Die anschließende Ausgabe einer TRACE-Meldung im Debug-Modus zeigt, ob dieser Tag mit den Einstellungen der Systemsteuerung übereinstimmt oder nicht.

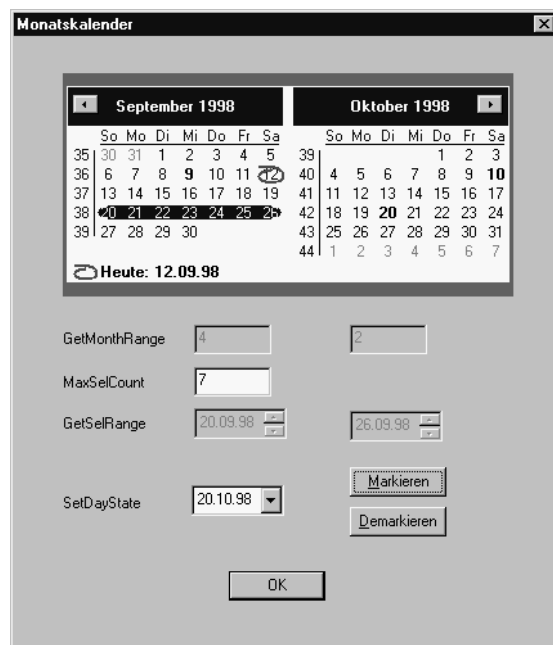


Abbildung 10.13: Der Monatskalender-Dialog

Die maximale Anzahl der markierbaren Tage bei Mehrfachselektion wird im Dialog über ein Eingabefeld vom Anwender festgelegt. Bei der Initialisierung ist der Standardwert aktiv. Dieser wird hier ausgelesen und in das Eingabefeld geschrieben. Hinzu kommt noch das Lesen des aktuell selektierten Bereiches und das Eintragen in die beiden Eingabefelder. Bei der Initialisierung ist es der aktuelle Tag, da nichts anderes festgelegt wurde.

Nun, da die nötigen Initialisierungen abgeschlossen sind, geht es darum, auf die Eingaben des Benutzers zu reagieren. Dazu ist es notwendig, für einige der gesendeten Nachrichten entsprechende Behandlungsroutinen anzulegen. Worauf soll reagiert werden? Auf die Wahl eines Bereiches des Nutzers, auf das Blättern innerhalb des Kalenders, um Tage zu markieren, und auf das Festlegen der maximalen Anzahl von wählbaren Tagen.

```
void CMonatsKalender::OnChangeSelcount()
{
    //Wert auslesen und MaxSelCount setzen
    UpdateData(TRUE);
    m_MK.SetMaxSelCount(m_SelCount);
}
```

Listing: Ändern der maximal wählbaren Tage durch den Benutzer

Das Ändern der maximal wählbaren Tage erfolgt in einer Eingabefeld-Steuerung vom Benutzer. Eine Änderung des Wertes erkennt man an der Nachricht `EN_CHANGE`. Diese wird in der Funktion `OnChangeSelCount` behandelt. Da das Eingabefeld eine Member-Variable vom Typ `Integer` besitzt, muß nur der Dialoginhalt mittels `UpdateData` in die Variable übertragen werden. Anschließend kann mit der Methode `SetMaxSelCount` sofort der neue Wert gesetzt werden.

```
void CMonatsKalender::OnSelchangeMK(NMHDR* pNMHDR, LRESULT* pResult)
{
    CTime minTime, maxTime;
    char sMinTime[256], sMaxTime[256];
    //Anzahl der angezeigten Monate ermitteln
    sprintf(sMinTime,"%i",m_MK.GetMonthRange(minTime,maxTime,
GMR_DAYSTATE));
    sprintf(sMaxTime,"%i",m_MK.GetMonthRange(minTime,maxTime,
GMR_VISIBLE));
    m_MR1.SetWindowText(sMinTime);
    m_MR2.SetWindowText(sMaxTime);
    //gewählten Bereich ermitteln
    if(m_MK.GetSelRange(minTime, maxTime))
    {
        m_DT1 = minTime;
        m_DT2 = maxTime;
        UpdateData(FALSE);
    }
    *pResult = 0;
}
```

Listing: Reaktion auf das Blättern im Kalender

`OnSelchange` wird unter anderem dann aufgerufen, wenn der Benutzer die Monate des Kalenders durchblättert. In der Funktion wird mit der Methode `GetMonthRange` die Anzahl der angezeigten Monate in zwei Varianten ermittelt. Der zurückgegebene Integer-Wert wird mittels `sprintf` sofort in eine Zeichenkette konvertiert. Anschließend werden diese Zeichenketten mit Hilfe der Methode `SetWindowText` der Eingabefeld-Steuerung in das Eingabefeld geschrieben.

Des weiteren wird in der Nachrichtenbehandlung der gewählte Bereich ermittelt. Die Datum/Uhrzeit-Felder, in die die Werte geschrieben werden, wurden als Wert-Felder im Klassen-Assistenten angelegt. Somit ist es möglich, die in der Methode `GetSelRange` ermittelten `CTime`-Objekte sofort den Steuerungen zuzuweisen und mittels `UpdateData` anzuzeigen.

In dieser Funktion sind zwei verschiedene Möglichkeiten dargestellt, wie Daten in einer Steuerung sichtbar gemacht werden können. Die Anwendung einer Methode hängt davon ab, wie die Member-Variablen im Klassen-Assistenten angelegt wurden – als Wert oder Control. Geht es nur darum, Werte anzuzeigen oder auszulesen, können sie als Wert angelegt werden.

```
void CMonatsKalender::OnGetdaystateMK(NMHDR* pNMHDR, LRESULT*
pResult)
{
    NMDAYSTATE* pDayState= (NMDAYSTATE *)pNMHDR;
    TRACE("GetDayState\n");
    int nDispMon = pDayState->stStart.wMonth;
    if(nDispMon==12) nDispMon=0;
    pDayState->prgDayState = mdState+nDispMon-1;
    *pResult = 0;
}
```

Listing: Behandlung des Tagesstatus

Kommen wir nun zum Hervorheben bestimmter Tage. In einem Terminkalender sind zum Beispiel alle Tage fett dargestellt, bei denen bereits Termine vorhanden sind. Genau dieses Verhalten kann hier abgebildet werden. Voraussetzung ist das Einfügen einer Nachrichtenbehandlungs-routine für die Nachricht MCN_GETDAYSTATE. Sie wird von der Steuerung aufgerufen, um die zu markierenden Daten zu ermitteln.

Der Status aller Tage eines Monats wird in einer Variablen vom Typ MONTHDAYSTATE festgehalten. Die 32 Bit werden als Bitfeld betrachtet. Dabei erfolgt eine Zuordnung der einzelnen Bits zu je einem Tag. Ist das Bit gesetzt, wird der Tag fett dargestellt und umgekehrt. Da immer mehrere Monate gleichzeitig angezeigt werden, wird ein Array benötigt. Um etwas mehr Funktionalität als im Beispiel der Online-Hilfe zu erreichen, wurde in der Header-Datei ein Array für 12 Monate angelegt, um die Stati für 12 Monate zu speichern. Auch dieses reicht normalerweise nicht aus. An den Übergängen von Januar und Dezember kommt es zu Problemen bei der Darstellung der Stati.

```
MONTHDAYSTATE mdState[12];
```

Im Konstruktor wird dieses Array mit Null initialisiert, d.h. kein Tag wird hervorgehoben.

```
for(int i=0;i<12;i++)
    mdState[i] = (MONTHDAYSTATE)0;
```

In der Nachrichtenbehandlung `OnGetdaystateMK` wird die in der Online-Dokumentation beschriebene Variable `pDayState` vom Typ `NMDAYSTATE` angelegt. In dieser Struktur werden die Informationen für die Steuerung abgelegt. Zunächst wird der erste angezeigte Monat ermittelt. Dieser wird als Index für die Struktur `mdState` benutzt, die dann an `pDayState->prgDayState` übergeben wird. In dieser Nachrichtenbehandlungsroutine ist es nicht möglich, auf die Steuerung zuzugreifen oder Methoden wie `SetDayState` aufzurufen, da die Routine erstmals aufgerufen wird, bevor die Steuerung erzeugt wurde.

Das eigentliche Markieren erfolgt in der Methode `OnMark`, die durch das Klicken auf die Schaltfläche MARKIEREN ausgelöst wird.

```
void CMonatsKalender::OnMark()
{
    UpdateData(TRUE);
    CTime minTime, maxTime;
    int nMonate = m_MK.GetMonthRange(minTime,maxTime,GMR_DAYSTATE);
    int nDispMon = minTime.GetMonth();
    if(nDispMon==12) nDispMon=0;
    BOLDDAY(mdState[m_DT_Mark.GetMonth()-1],m_DT_Mark.GetDay());
    m_MK.SetDayState(nMonate, mdState + nDispMon-1);
}
```

Listing: Markieren einzelner Tage im Kalender

Im Beispielprogramm wird der hervorzuhebende Tag in einer Datum/Uhrzeit-Steuerung festgelegt. Wenn der Benutzer auf MARKIEREN klickt, wird dieser Wert ausgelesen. Mit Hilfe der Methode `GetMonthRange` werden die Anzahl der angezeigten Monate `nMonate` sowie der erste Monat `nDispMon` ermittelt. Das Makro `BOLDDAY` setzt das Bit für den Tag und Monat, der hervorgehoben werden soll, in der Variablen `mdState`. Schließlich wird mit der Methode `SetDayState` der Tagesstatus gesetzt und angezeigt, wenn es sich um einen Tag der angezeigten Monate handelt. Das Makro `BOLDDAY` (wie auch `NORMDAY`) vereinfacht das Setzen/Löschen der Bits und macht das Programm übersichtlicher. Die Makros werden mit dem `#define`-Statement definiert.

```
#define BOLDDAY(ds,iDay) if(iDay>0 && iDay<32)
(ds)|=(0x00000001<<(iDay-1))
#define NORMDAY(ds,iDay) if(iDay>0 && iDay<32)
(ds)&=(0xFFFFFFF&&<<(iDay-1))
```

Listing: Die Makros zum Markieren und Demarkieren

Die Nachrichtenbehandlung `OnDemark` zur Demarkierung eines Tages läuft genauso ab wie das Markieren, mit der Ausnahme, daß anstelle von `BOLD-DAY` das Makro `NORMDAY` aufgerufen wird.

```
void CMonatsKalender::OnDemark()
{
    UpdateData(TRUE);
    CTime minTime, maxTime;
    int nMonate = m_MK.GetMonthRange(minTime,maxTime,GMR_DAYSTATE);
    int nDispMon = minTime.GetMonth();
    if(nDispMon==12) nDispMon=0;
    NORMDAY(mdState[m_DT_Mark.GetMonth()-1],m_DT_Mark.GetDay());
    m_MK.SetDayState(nMonate, mdState + nDispMon-1);
}
```

Listing: Funktion zum Demarkieren eines Tages

In einem konkreten Anwendungsprogramm würde das Markieren der Tage wahrscheinlich in einer anderen Funktion erfolgen, die hier dargestellte Vorgehensweise kann aber übernommen werden.

10.16 Die IP-Adressensteuerung

10.16.1 Anwendung

Die Steuerung dient zur komfortablen, formatierten Eingabe von IP-Adressen. Es besteht aus einer Art Eingabefeld, unterteilt in vier Felder, die durch Punkte getrennt sind. Jedes Feld repräsentiert einen 8-Bit-Wert, ist also zwischen 0 und 255 gültig.

172	.	18	.	45	.	2

Abbildung 10.14: Die IP-Adressen-Steuerung

10.16.2 Attribute

Außer den Standard- und den erweiterten Eigenschaften besitzt eine IP-Adressen-Steuerungsfelder keine weiteren Attribute.

10.16.3 Benachrichtigungscodes

Die Nachricht `IPN_FIELDCHANGED` wird geschickt, wenn der Benutzer eines der Steuerungsfelder ändert oder den Cursor innerhalb der Felder bewegt. Weitere Nachrichten werden nicht verschickt.

10.16.4 Die Klasse *CIPAddressCtrl*

IsBlank

Die Methode *IsBlank* ermittelt, ob alle Felder der Steuerung leer sind, und gibt einen Wert ungleich Null zurück, wenn das der Fall ist.

```
class CIPAddressCtrl:
    BOOL IsBlank( ) const;
```

ClearAddress

Durch die Anwendung der Methode *ClearAddress* werden alle Felder der Steuerung gelöscht.

```
class CIPAddressCtrl:
    void ClearAddress( );
```

GetAddress und SetAddress

Zu den wichtigsten Methoden der IP-Adressen-Steuerung zählen *GetAddress* und *SetAddress* zum Lesen und Setzen der IP-Adresse. Die IP-Adresse kann in zwei verschiedenen Arten angegeben werden: die gesamte Adresse als DWORD oder jedes einzelne Feld als BYTE. Die Funktion gibt einen Integer-Wert zurück, der die Anzahl der Felder enthält, die nicht leer sind. Wird als IP-Adresse ein DWORD benutzt, so repräsentieren die Bits in Tabelle 10.32 die einzelnen Felder.

Feld	Bit-Repräsentation
0	24 – 31
1	16 – 23
2	8 – 15
3	0 – 7

Tabelle 10.32: Zuordnung der DWORD-Bits zu den Feldern

```
class CIPAddressCtrl:
    int GetAddress( BYTE& nField0, BYTE& nField1, BYTE& nField2,
        BYTE& nField3 );
    int GetAddress( DWORD& dwAddress );
    void SetAddress( BYTE nField0, BYTE nField1, BYTE nField2,
        BYTE nField3 );
    void SetAddress( DWORD dwAddress );
```

SetFieldFocus

Mit der Methode *SetFieldFocus* lässt sich auf komfortable Weise der Cursor in ein bestimmtes Feld der Steuerung setzen. Somit kann der Anwender gezielt auf Fehleingaben hingewiesen werden. Der Index der Variablen *nField* beginnt mit Null. Wird ein Wert größer drei angegeben, springt der Cursor auf das erste freie Feld; sind alle Felder belegt, springt er auf das erste Feld.

```
class CIPAddressCtrl:
    void SetFieldFocus( WORD nField );
```

SetFieldRange

Soll der Benutzer nur IP-Adressen in einem bestimmten Bereich eingeben können, ist es möglich, mit *SetFieldRange* diesen Bereich für jedes einzelne Feld zu setzen. In *nField* wird das Feld angegeben, in *nLower* und *nUpper* der Bereich von/bis.

```
class CIPAddressCtrl:
    void SetFieldRange( int nField, BYTE nLower, BYTE nUpper );
```

10.16.5 Beispielprogramm

Das Beispiel zur IP-Adressen-Steuerung wurde als Dialogseite in das Programm CONTROLS integriert. Die Seite sieht auf den ersten Blick sehr einfach aus (siehe Abbildung 10.15), zeigt aber die Anwendung einiger Methoden der Klasse *CIPAddressCtrl*, um eine abgesicherte Eingabe vom Benutzer zu erhalten.

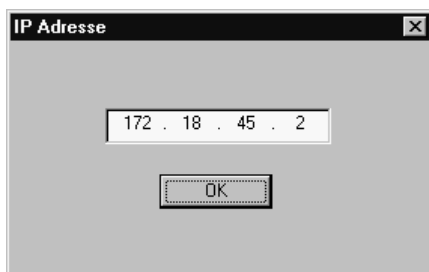


Abbildung 10.15: IP-Dialogseite

```
BOOL CIPAdresse::OnInitDialog()
{
    CDialog::OnInitDialog();
    m_IP.SetAddress(172,18,45,0);
    m_IP.SetFieldRange(0,172,172);
```



```

m_IP.SetFieldRange(1,18,18);
m_IP.SetFieldRange(2,40,48);
return TRUE; // return TRUE unless you set the focus to a
control
// EXCEPTION: OCX-Eigenschaftenseiten sollten FALSE
zurückgeben
}

```

Listing: Die Initialisierung einer IP-Adressensteuerung

Das Setzen der nötigen Optionen erfolgt in der Methode *OnInitDialog*, in der die Voreinstellungen der Steuerung vorgenommen werden. Neben dem Setzen einer Standard-IP-Adresse als Vorgabewert, werden auch die Wertebereiche der ersten drei Felder festgelegt. Dieses erfolgt mit der Methode *SetFieldRange*. Damit sind alle Initialisierungen vorgenommen – der Benutzer kann mit seinen Eingaben beginnen. Zur Beendigung der Adressen-Eingabe betätigt der Benutzer die OK-Schaltfläche. Dadurch wird die Methode *OnOK* aufgerufen, in der die Kontrolle der Benutzereingaben erfolgt. Erst wenn die Eingaben korrekt sind, kann der Benutzer den Dialog verlassen. Dies ist eine Möglichkeit, den Benutzer zu korrekten Eingaben zu zwingen.

```

void CIPAdresse::OnOK()
{
    BYTE nF0, nF1, nF2, nF3;
    char buf[256];
    if(m_IP.GetAddress(nF0,nF1,nF2,nF3)<4)
    {
        MessageBeep(MB_ICONHAND);
        m_IP.SetFieldFocus(5);
    }
    else
    {
        sprintf(buf,"gewählte Adresse: %i.%i.%i.%i",nF0,nF1,nF2,nF3);
        AfxMessageBox(buf);
        CDialog::OnOK();
    }
}

```

Listing: Auslesen der IP-Adresse aus der Steuerung

Die Methode *GetAddress* in *OnOK* hat zwei Funktionen: zum einen ermittelt sie die vom Benutzer eingegebene Adresse und legt sie in lokalen Variablen ab, zum anderen überprüft sie, ob Felder freigelassen wurden. Ist das der Fall, wird ein Fehlerton ausgegeben, und anschließend wird mit

der Methode *SetFieldFocus* der Cursor an die erste fehlerhafte Stelle gesetzt. Sind alle Eingaben richtig, wird die IP-Adresse als Zeichenkette in einer Message-Box ausgegeben.

10.17 Das erweiterte Kombinationsfeld-Steuerelement

10.17.1 Anwendung

Das erweiterte Kombinationsfeld (siehe Abbildung 10.16) entspricht dem Kombinationsfeld aus Kapitel 10.3, erweitert um die Benutzung von Image-Listen zur Einblendung von kleinen Icons vor dem Eintrag. Bisher war dies nur über den Style CBS_OWNERDRAWFIXED möglich, wobei der Programmierer dann für die gesamte Ausgabe verantwortlich ist. In das erweiterte Kombinationsfeld ist das »normale« Kombinationsfeld als Kindfenster eingebettet.



Weitere Informationen zur Verwendung von Image-Listen finden Sie in Kapitel 11.3.

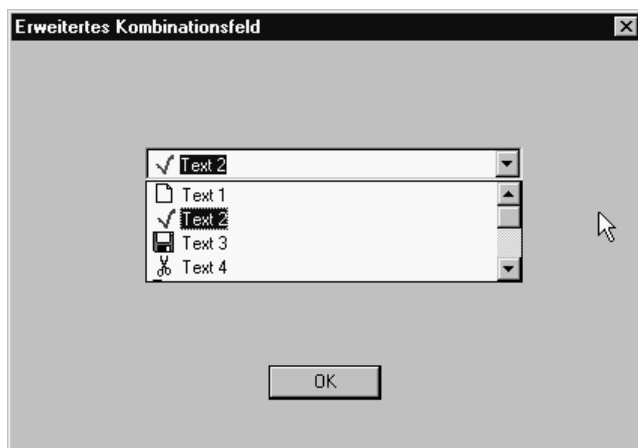


Abbildung 10.16: Erweitertes Kombinationsfeld mit geöffneter Dropdown-Liste

10.17.2 Attribute

Außer den Standard- und den erweiterten Eigenschaften besitzt eine IP-Adressen-Steuerung drei weitere Attribute. Diese können direkt im Eigenschaften-Fenster des Ressource-Editors geändert werden. Sie bestimmen das Aussehen des Kombinationsfeldes.

Dialogeditor	Dialogboxschablone	Bedeutung
Einfach, Dropdown, Dropdown-Listenfeld (Simple, Dropdown, Drop List)	Ressource-Typ: CONTROL Style: <i>CBS_SIMPLE</i> <i>CBS_DROPDOWN</i> <i>CBS_DROPDOWNLIST</i>	Mit diesem Attribut wird festgelegt, daß der Kalender bestimmte Tage fett hervorgehoben darstellen kann.
	Erweiterter Style: <i>CBES_EX_CASESENSITIVE</i>	Die Suche von Zeichenketten im Kombinationsfeld wird auf die exakte Suche umgestellt. Es wird zwischen Groß- und Kleinschreibung unterschieden.
	Erweiterter Style: <i>CBES_EX_NOEDITIMAGE</i>	In der Eingabezeile wird kein Bild (Icon) angezeigt.
	Erweiterter Style: <i>CBES_EX_NOSIZELIMIT</i>	Das erweiterte Kombinationsfeld kann in der horizontalen Ausdehnung kleiner sein als das eingebettete Kombinationsfeld, das dann abgeschnitten wird.
	Erweiterter Style: <i>CBES_EX_NOEDITIMAGEINDENT</i>	Das Einrücken des Textes wird verhindert. Der Text wird über die Bilder gezeichnet.
	Erweiterter Style: <i>CBES_EX_PATHWORDBREAKPROC</i>	Dieses Attribut ist nur bei Windows NT wirksam. Es ermöglicht dem Anwender den Cursor wortweise zu bewegen. Als Worttrennzeichen dienen Schrägstrich, Backslash und Punkt. In Datei- und Pfadangaben ist dies sehr nützlich.

Tabelle 10.33: Attribute des erweiterten Kombinationsfeldes

Alle erweiterten Styles in Tabelle 10.33 können nicht direkt im Ressource-Editor gesetzt werden. Sie dürfen erst dann benutzt werden, wenn das Fenster schon existiert, also zum Beispiel in der Methode *OnInitDialog*. Eine Member-Funktion der Klasse *CComboBoxEx* ist dafür zuständig.

10.17.3 Benachrichtigungscodes

Neben den Benachrichtigungscodes der Standard-Kombinationsschaltfläche besitzt die erweiterte noch die folgenden: *CBEN_BEGINEDIT* zeigt an, daß der Anwender selbst Text eingibt oder ein Element aus der Liste auswählen will. Das Senden von *CBEN_ENDEDIT* heißt, daß der Benutzer die Eingabe beendet oder ein Element aus der Liste gewählt hat.

CBEN_DELETEITEM wird gesendet, wenn ein Element gelöscht wurde und *CBEN_INSERTITEM*, wenn es eingefügt wurde.

Die Nachricht *CBEN_DRAGBEGIN* zeigt an, daß der Benutzer das Bild des Eintrags mit der Maus angeklickt hat, um es zu verschieben.

10.17.4 Die Klasse *CComboBoxEx*

InsertItem und DeletelItem

Mit Hilfe von *InsertItem* werden Elemente zum Kombinationsfeld hinzugefügt. Dies ist immer dann erforderlich, wenn die Elemente zur Kompilierung noch nicht feststehen, also nicht statisch zur Ressource eingegeben worden sind. Der Parameter *pCBItem* enthält einer Struktur vom Typ *COMBOBOXEXITEM*. Sie ist ähnlich aufgebaut wie die Struktur *TV_ITEM* der Strukturansicht.

```
typedef struct {
    UINT mask;
    int iItem;
    LPTSTR pszText;
    int cchTextMax;
    int iImage;
    int iSelectedImage;
    int iOverlay;
    int iIndent;
    LPARAM lParam;
} COMBOBOXEXITEM, *PCOMBOBOXEXITEM;
```

Der Parameter *mask* legt die Gültigkeit der Einträge der Struktur fest. Die möglichen Werte sind in Tabelle 10.34 zu finden. In *iItem* wird der nullbasierende Index des Eintrags gespeichert. Der Text des Eintrags wird über einen Zeiger in *pszText* abgelegt. Wenn der Benutzer Werte eingeben darf, muß die Länge des Textpuffers in *cchTextMax* angegeben werden. In *iImage*, *iSelectedImage* und *iOverlay* wird jeweils der nullbasierende Index auf ein Bild der Image-Liste gespeichert. Sie gelten für das normale Bild, das Bild des ausgewählten Eintrags und ein Bild zur Überlagerung. Im Parameter *lParam* kann ein 32-Bit-Wert abgelegt werden, der den Eintrag näher kennzeichnet.

Wert	Bedeutung
CBEIF_DI_SETITEM	Die Einträge werden in der Steuerung gespeichert und nicht noch einmal abgefragt. Der Parameter wird in Zusammenhang mit der <i>CBEN_GETDISPINFO</i> -Nachricht verwendet.
CBEIF_INDENT	Der Parameter <i>iIndent</i> ist gültig und enthält den Wert zum Einrücken des Textes.
CBEIF_LPARAM	Der Parameter <i>lParam</i> enthält einen gültigen Wert.
CBEIF_IMAGE	In <i>iImage</i> wurde der Wert für das Bild des Eintrags angegeben.

Wert	Bedeutung
CBEIF_OVERLAY	iOverlay enthält einen gültigen Wert.
CBEIF_SELECTEDIMAGE	Der Wert des ausgewählten Bildes ist gültig.
CBEIF_TEXT	Der Text-Parameter der Struktur enthält gültige Werte.

Tabelle 10.34: Werte des Parameters mask der COMBOBOXEXITEM-Struktur

Die Methode *InsertItem* gibt einen Integer-Wert zurück, der den Index enthält, an dem der neue Eintrag eingefügt wurde. Falls das Einfügen fehlschlägt, wird »- 1« zurückgegeben.

Mit Hilfe von *DeleteItem* kann der Eintrag iIndex aus der Liste der Einträge gelöscht werden. In diesem Fall wird die Anzahl der verbleibenden Einträge zurückgegeben. Ist iIndex ungültig, wird CB_ERR zurückgeliefert.

```
class CComboBoxEx:
    int InsertItem( const COMBOBOXEXITEM* pCBItem );
    int DeleteItem( int iIndex );
```

GetItem und SetItem

Mit der Methode *GetItem* kann der Inhalt eines Eintrags des Kombinationsfeldes ermittelt werden. In pCBItem wird dazu die Adresse auf eine COMBOBOXEXITEM-Struktur angegeben, die die Informationen aufnimmt. War die Operation erfolgreich, wird ein Wert ungleich Null zurückgegeben. *SetItem* ist das Gegenstück zu *GetItem* und setzt die Parameter eines Eintrags neu. Auch hier wird Null im Fehlerfall zurückgegeben.

```
class CComboBoxEx:
    BOOL GetItem( COMBOBOXEXITEM* pCBItem );
    BOOL SetItem( const COMBOBOXEXITEM* pCBItem );
```

SetImageList und GetImageList

Um Bilder vor den Einträgen zu verwenden, muß eine Image-Liste angelegt werden. Diese Liste wird dann mit der Methode *SetImageList* der Steuerung bekanntgegeben. *GetImageList* liefert als Ergebnis einen Zeiger auf ein CImageList-Objekt, das die Steuerung enthält.

```
class CComboBoxEx:
    CImageList* SetImageList( CImageList* pImageList );
    CImageList* GetImageList( ) const;
```

GetComboBoxCtrl und GetEditCtrl

Wie einleitend erläutert, enthält die Steuerung ein normales Kombinationsfeld. Einen Zeiger auf die Klasse *CComboBox* der eingebetteten Steuerung erhält man über die Methode *GetComboBoxCtrl*. Damit können dann alle Methoden der Klasse *CComboBox* aufgerufen werden.

Über *GetEditCtrl* wird der Zeiger der im Kombinationsfeld enthaltenen Eingabefeld-Steuerung zurückgegeben. Der Zeiger sollte aber nicht längere Zeit gespeichert werden, da er vom System regelmäßig zerstört wird.

```
class CComboBoxEx:  
    CComboBox* GetComboBoxCtrl( );  
    CEdit* GetEditCtrl( );
```

GetExtendedStyle und SetExtendedStyle

Die im Kapitel »Attribute« beschriebenen erweiterten Styles des Kombinationsfeldes werden mit der Methode *SetExtendedStyle* gesetzt. Im Parameter *dwExMask* kann eine Anzahl von erweiterten Attributen angegeben werden, die geändert werden dürfen. Wird Null angegeben, können alle erweiterten Styles geändert werden. In *dwExStyles* werden die zu ändernden Attribute angegeben. Zurückgegeben werden die Attribute, die vor der Änderung gesetzt waren. Die Methode *GetExtendedStyles* gibt die erweiterten Attribute des Kombinationsfeldes zurück.

```
class CComboBoxEx:  
    DWORD GetExtendedStyle( ) const;  
    DWORD SetExtendedStyle( DWORD dwExMask, DWORD dwExStyles );
```

HasEditChanged

Mit Hilfe der Methode *HasEditChanged* wird ermittelt, ob der Benutzer Text im Eingabefeld des Kombinationsfeldes eingegeben bzw. geändert hat. Ist das der Fall, wird ein Wert ungleich Null zurückgegeben.

```
class CComboBoxEx:  
    BOOL HasEditChanged( );
```

10.17.5 Beispielprogramm

Das Beispiel zum erweiterten Kombinationsfeld wurde als Dialogseite realisiert; es ist sehr einfach gehalten. Die Steuerung enthält Einträge, die in der Methode *OnInitDialog* gesetzt werden. Zur Darstellung der Bilder wurde eine Image-Liste eingefügt. Die Bilder stammen von der Toolbar eines anderen Projektes. Es wurde nur ein Bild für das ausgewählte Element hinzugefügt.

```

BOOL CComboBox::OnInitDialog()
{
    CDialog::OnInitDialog();
    CImageList *m_List = new CImageList;
    COMBOBOXEXITEM m_Item;
    m_List->Create(IDB_BMPLISTE,16,1,RGB(192,192,192));
    m_CBEX.SetImageList(m_List);
    m_Item.mask = CBEIF_IMAGE | CBEIF_TEXT | CBEIF_SELECTEDIMAGE;
    m_Item.pszText = "Text 1";
    m_Item.iImage = 0;
    m_Item.iItem = 0;
    m_Item.iSelectedImage = 8;
    m_CBEX.InsertItem(&m_Item);
    m_Item.pszText = "Text 2";
    m_Item.iImage = 1;
    m_Item.iItem = 1;
    m_CBEX.InsertItem(&m_Item);
    ...
    m_Item.pszText = "Text 8";
    m_Item.iImage = 7;
    m_Item.iItem = 7;
    m_CBEX.InsertItem(&m_Item);
    m_CBEX.SetCurSel(1);
    return TRUE; // return TRUE unless you set the focus to a
    control
    // EXCEPTION: OCX-Eigenschaftenseiten sollten FALSE
    zurückgeben
}

```

Listing: Einfügen der Daten in das Erweiterte Kombinationsfeld

Das Erzeugen der Image-Liste ist in diesem Fall recht einfach. Anhand der Breite ermittelt die Methode *Create* von *CImageList* automatisch die Anzahl der Bilder und baut die Liste auf. Mit Hilfe von *SetImageList* wird sie der Steuerung bekanntgegeben. Anschließend werden die Einträge aufgebaut, wobei die Parameter *pszText*, *iImage* und *iSelectedImage* gültig sind. Mit der Methode *InsertItem* werden die Einträge hinzugefügt. Zum Schluß wird noch der Eintrag »Eins« vorausgewählt. Da die Klasse *CComboBoxEx* von *CComboBox* abgeleitet ist, kann die Methode *SetCurSel* aufgerufen werden. Damit steht dem Anwender die Auswahlsteuerung zur Verfügung.

10.18 ActiveX-Controls

10.18.1 Anwendung

Über ActiveX-Controls wurde viel geschrieben, besonders im Zusammenhang mit dem Internet. Sie sind aber nicht nur in HTML-Seiten verwendbar, sondern sind universell einsetzbare Steuerelemente. Entwickelt haben sie sich aus den sehr populären VBX-Controls von Visual Basic über die OLE-Controls in der Version 4 von Visual C++. ActiveX-Controls sind eigentlich auch nichts anderes als OCX-Controls, was auch an ihrer Dateinamenserweiterung zu sehen ist.

Im Zusammenhang mit ActiveX Controls ist auch das Schlagwort »Komponentensoftware« wieder hervorgeholt worden. Ziel von Komponentensoftware ist es, daß der Anwender sich seine Programme selbst zusammenstellt: eine Komponente hiervon, eine davon – und fertig ist eine Textverarbeitung. Damit sollen kleinere und vor allem überschaubarere Programme entstehen. Doch bis dahin ist es noch ein weiter Weg.

Microsoft liefert mit Visual C++ 6 und auch schon mit dem Betriebssystem einige (und nicht wenige!) ActiveX-Controls mit. Leider ist die Dokumentation dieser Komponenten meist nicht besonders gut. So ist die bei der Anwendung fertiger Komponenten gewonnene Zeit wieder weg. Wir legen Ihnen hier nur beispielhaft die Vorgehensweise bei der Benutzung von ActiveX-Controls dar. Als Anwendungsbeispiel wurde das *Web-Browser-Control* gewählt, mit dem es möglich wird, einen Internet Explorer in eigene Programme einzubinden.

Voraussetzung für den Einsatz von ActiveX-Controls ist, daß die jeweiligen Controls auf dem Zielsystem vorhanden sein müssen. Das bedeutet für den Entwickler, daß die entsprechende Datei auf dem Zielsystem installiert werden muß. Hier gilt es auch, die lizenzrechtlichen Bestimmungen zu beachten. Meist reicht das einfache Kopieren der Datei aber nicht aus, sondern das Control muß registriert werden. Auch dafür bietet Visual C++ Unterstützung. Im EXTRAS-Menü befindet sich der Eintrag TESTCONTAINER FÜR ACTIVEX-STEUERELEMENTE. Mit Hilfe dieses Programms kann man nicht nur neue Controls registrieren, sondern vorhandene Controls auch testen. Eine weitere Möglichkeit zum Registrieren von ActiveX-Controls besteht mit dem Programm REGSVR32. Dieses arbeitet kommandozeilenorientiert und bekommt als Parameter den Namen der Datei übergeben.

10.18.2 Ablauf

Als erstes ist es notwendig, die ActiveX-Komponente in das Projekt einzufügen. Das erfolgt über das MENÜ PROJEKT|DEM PROJEKT HINZUFÜGEN|KOMponenten und Steuerelemente (siehe Abbildung 10.17). Danach wird eine erweiterte Dateiauswahlbox geöffnet, die zur Wahl zwischen Developer

Studio-Komponenten oder registrierten ActiveX-Komponenten auffordert. Nach der Auswahl des Ordners für registrierte ActiveX-Komponenten werden alle im System vorhandenen ActiveX-Controls aufgelistet. Zu jedem Element wird eine kurze (meist nichtssagende) Beschreibung angezeigt, sobald es angewählt wird. Über den Button WEITERE INFORMATIONEN kann zu einigen Steuerelementen eine Hilfedatei aufgerufen werden.



Abbildung 10.17: Einfügen von ActiveX-Controls

Mit Hilfe des Buttons EINFÜGEN wird die Komponente in das aktuelle Projekt übernommen. Dabei wird noch eine Dialogbox geöffnet, die darüber informiert, welche Klassen und Dateien dem Projekt hinzugefügt werden (Abbildung 10.18). Einzelne Klassen können dabei auch deaktiviert werden.

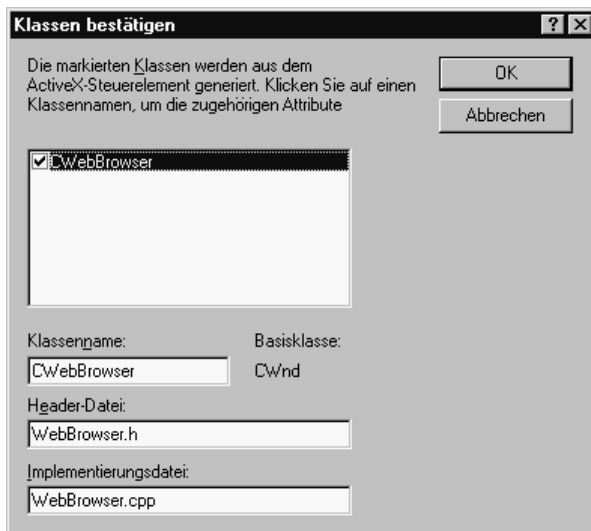


Abbildung 10.18: Einfügen des Web Browser Controls in das Projekt

Nach Betätigen der OK-Taste wird die Klasse dem Projekt hinzugefügt, und in der Steuerelemente-Palette im Ressource-Editor steht die neue Komponente zur Verfügung. Der restliche Teil der Arbeiten läuft wie gewohnt ab: Im Ressource-Editor wird der Dialog geladen oder angelegt, in dem das Control eingefügt werden soll; anschließend kann es wie jede andere Steuerung im Dialog angelegt werden.

10.18.3 Attribute

Die Standardeigenschaften besitzen auch alle ActiveX-Controls. Alle weiteren Attribute sind vom jeweiligen Control abhängig. Einige besitzen nur wenige weitere Attribute, zu denen auch das Web Browser Control zählt, andere wiederum haben eine fast unüberschaubare Zahl von Attributen im Gepäck.

10.18.4 Beispielprogramm

Das Programm MEINBROWSER wurde als dialogbasierende Anwendung angelegt. Dieser wird das ActiveX Control *Web Browser* hinzugefügt.

Das Web Browser Control wurde wie oben beschrieben aus der Komponenten-Sammlung in das aktuelle Projekt eingefügt. Mit dem Ressource-Editor wird ein neuer Dialog angelegt, und darin wird das *Web Browser Control* platziert. Nun gilt es zu überlegen, welche Funktionen der Browser mindestens ausführen muß. Denkt man dabei an den Internet Explorer, so fällt einem zuerst die Eingabe einer Adresse ein, von welcher der Browser die Seiten laden soll. In unserem Beispiel wird die Eingabe der Adresse in einem Eingabefeld ermöglicht. Die Schaltfläche Go, die als Standard-Schaltfläche definiert ist und damit auf Druck der Enter-Taste reagiert, weist den Browser an, im Internet danach zu suchen. Wurde die Web-Seite gefunden, wird sie im ActiveX Control angezeigt.



Sie können auch über Querverweise andere Seiten aufrufen.

Die Schaltfläche Home bringt Sie zurück zur Startseite. Hier wird die Seite angesprungen, die in den Eigenschaften des Internet Explorers als Startseite eingetragen ist. Versäumen Sie es nicht, auch die anderen beiden Buttons auszuprobieren ...

Um diese Funktionalität zu erreichen, sind nur zwei Methoden der Klasse *CWebBrowser* erforderlich: *GoHome* und *Navigate*. Da dieses Control eines von denjenigen ist, die nicht dokumentiert sind, muß man sich in der Header-Datei die entsprechenden Funktionen selbst suchen.

```
void CMeinBowserDlg::OnGo()  
{  
    CString sURL;  
    m_URL.GetWindowText(sURL);
```

```
m_WWW.Navigate(sURL,NULL,NULL,NULL,NULL);
}
void CMeinBrowserDlg::OnHome()
{
    m_WWW.GoHome();
}
void CMeinBrowserDlg::OnAutor()
{
    m_WWW.Navigate("visualc.kessler-gruppe.de",NULL,NULL,NULL,NULL);
}
void CMeinBrowserDlg::OnAddison()
{
    m_WWW.Navigate("www.addison-wesley.de", NULL,NULL,NULL,NULL);
}
```

Listing: Nachrichten-Funktion der Schaltflächen des Browsers

Fügen Sie weitere Buttons in den Dialog ein, die Sie mit den Methoden *GoBack*, *GoForward* und *Stop* verbinden, und testen Sie sie aus.

Das ActiveX Control sendet auch Nachrichten auf eine Reihe von Ereignissen. So könnte zum Beispiel über die Ereignisse *BeforeNavigate* und *NavigateComplete* eine Anzeige realisiert werden, daß der Browser noch mit dem Laden einer Seite beschäftigt ist.





Weitere MFC-Klassen

11

Kapitelübersicht

11.1 Die Klasse CString	406
11.1.1 Allgemeines	406
11.1.2 Anlegen von CString-Objekten	406
11.1.3 Zugriff auf Zeichen	408
11.1.4 Zuweisung und Vergleich	408
11.1.5 Extrahieren von Teilzeichenketten und weitere Konvertierungen	409
11.1.6 Suchen von Teilzeichenketten	410
11.2 Die Klasse CObArray	410
11.2.1 Die Familie von Feldern	410
11.2.2 Anlegen und Zugriff	410
11.3 Die Klasse CImageList	412
11.3.1 Allgemeines	412
11.3.2 Anlegen einer Image-Liste	412
11.3.3 Hinzufügen und Löschen von Bildern	414
11.4 Die Symbolleiste	415
11.4.1 Allgemeines	415
11.4.2 Standard-Code vom Anwendungs-Assistenten	415
11.4.3 Laden und Speichern der Einstellungen	417
11.4.4 Mehrere Symbolleisten	419
11.4.5 Text-Label	419
11.4.6 Drop-down-Listen	420
11.5 Die Klasse CFile	422
11.5.1 Grundlagen	422
11.5.2 Hierarchie und abgeleitete Klassen	423
11.5.3 Die wichtigsten Methoden	424
11.5.4 Ausnahmebehandlung	428
11.5.5 Ein Beispiel	430

Die MFC besteht zum größten Teil aus Klassen, die alle eine gemeinsame Basisklasse haben – *CObject*. Es existieren aber auch einige, die nicht von *CObject* abgeleitet sind und quasi parallel dazu existieren (siehe Anhang E). Zu diesen Klassen gehören unter anderem *CArchive*, *CRuntimeClass* oder *CRect*. In diesem Abschnitt kommt eine weitere aus dieser Kategorie hinzu, die bereits benutzte Klasse *CString*. Sie ist für die Behandlung und Speicherung von Zeichenketten zuständig.

Weitere Klassen, die oft benötigt werden, gehören zu den Arrays oder Feldern. Diese sind wiederum direkt von *CObject* abgeleitet und dienen der Aufnahme einer Anzahl von Elementen eines Datentyps.

Auch die für teilweise Steuerungen wichtige Klasse *CImageList* soll in diesem Abschnitt kurz behandelt werden.

Eine Klasse, die uns indirekt schon begegnet ist, ist *CToolBar*. Wie mit ihrer Hilfe einige Änderungen am Layout möglich sind, wird in einem weiteren Abschnitt erläutert. Dazu kommen eine Menge Tips und Tricks im Umgang mit dieser Klasse und der Symbolleiste.

Für eine große Anzahl von Programmen ist es wichtig, unstrukturierte Daten auf einen Datenträger zu schreiben oder von ihm zu lesen. Die MFC kapselt auch den Dateizugriff in eine eigene Klasse in *CFile*. Die dazugehörigen Member-Funktionen bieten einen komfortablen Zugriff auf die Daten auf einem Datenträger.

11.1 Die Klasse *CString*

11.1.1 Allgemeines

Die Klasse *CString* dient der Speicherung einer variablen Anzahl von Zeichen. Zusätzlich existieren in der Klasse eine Vielzahl von Methoden zum Bearbeiten oder Abfragen der Zeichenketten. Zudem sind Operatoren für Vergleichsoperationen oder Zuweisungen definiert. *CString*-Objekte können ganz einfach wie Zeichenketten behandelt werden. Sie können auch überall da eingesetzt werden, wo LPCSTR-(oder *const char**-)Parameter gefordert sind. Wichtig für zukünftige Aufgaben ist die Portierbarkeit von Programmen. Auch hier bietet die Klasse *CString* die volle Unterstützung an, wenn es darum geht, UNICODE-Programme zu schreiben. Ein Zeichen ist dann 16 Bit lang anstelle der jetzigen 8 Bit.

11.1.2 Anlegen von *CString*-Objekten

Das Anlegen einer neuen Zeichenkette mit der Klasse *CString* ist auf verschiedene Arten möglich. Die Instanz der Klasse kann sowohl auf dem Heap als auch auf dem Stack angelegt werden. Dazu ist nur eine Member-Variable vom Typ *CString* anzulegen, die dann den Konstruktor aufruft. Die Constructoren der Klasse *CString* lassen vielfältige Möglichkeiten zu:

```

class CString:
    CString( );
    CString( const CString& stringSrc );
        throw( CMemoryException );
    CString( TCHAR ch, int nRepeat = 1 );
        throw( CMemoryException );
    CString( LPCTSTR lpch, int nLength );
        throw( CMemoryException );
    CString( const unsigned char* psz );
        throw( CMemoryException );
    CString( LPCWSTR lpsz );
        throw( CMemoryException );
    CString( LPCSTR lpsz );
        throw( CMemoryException );

```

Da diese Konstruktoren Speicher allokieren, kann es zu Memory-Exceptions kommen. Die folgenden Beispiele zeigen einige Möglichkeiten, wie man *CString*-Instanzen anlegen kann.

```

CString s1;
// eine leere Zeichenkette
CString s2( "Test" );
// durch Zuweisen einer literalen Zeichenkette
CString s3 = s2;
// durch Kopieren aus einer anderen Zeichenkette
CString s4( s2 + " " + s3 );
// durch Zusammenfügen mehrerer Zeichenketten
CString s5( 'x' );
// s5 = "x"
CString s6( 'x', 6 );
// s6 = "xxxxxx" Generieren einer Zeichenkette
CString *s7;
// Anlegen eines Zeigers auf ein CString-Objekt
s7 = new CString("Test");
// Instantiieren des Objekts auf dem Heap
delete( s7 );
// Entfernen der Instanz !

```



Auf dem Heap mit dem Operator *new* angelegte Objekte müssen mit *delete* am Ende wieder entfernt werden, da sonst Memory Leaks (siehe Kapitel »Fehlersuche«) entstehen!

Wie Sie sehen, ist das Zuweisen von Zeichenketten an eine Klasse *CString* genauso einfach wie das Zuweisen an eine Variable. Verantwortlich dafür ist die Definition eines Operators = in der Klasse.

11.1.3 Zugriff auf Zeichen

Zum Zugriff auf einzelne Zeichen der Zeichenkette existieren einige Methoden. Der Zugriff geht ähnlich vor sich wie bei einem Charakter-Array. Mittels *GetLength* kann die Länge der Zeichenkette ermittelt werden, die als Integer-Zahl zurückgegeben wird. Die Methode *GetAt* liefert das Zeichen an der angegebenen Position, wobei diese zwischen 0 und *GetLength* liegen muß! Das Gegenstück dazu – *SetAt* – ändert das Zeichen an der Position *nIndex* in *ch*. Die Klasse definiert noch den Operator *[]*, der es ermöglicht, auf ein Zeichen zuzugreifen wie in einem *char*-Array der C-Programmierung. Mit dem Operator können Zeichen nur gelesen, nicht aber geändert werden.

```
class CString:  
    int GetLength( ) const;  
    TCHAR GetAt( int nIndex ) const;  
    void SetAt( int nIndex, TCHAR ch );  
    TCHAR operator [ ]( int nIndex ) const;  
    BOOL IsEmpty( ) const;  
    void Empty( );
```

Die Methode *IsEmpty* ermittelt, ob eine Zeichenkette leer ist, während *Empty* alle Zeichen der Zeichenkette entfernt.

11.1.4 Zuweisung und Vergleich

Das Zuweisen einer Zeichenkette an ein *CString*-Objekt erfolgt auf einfache Weise mit dem Operator *=*. Der Operator ist polymorph ausgeführt, um möglichst viele Varianten der Zuweisung abzudecken. Zum Verketteten zweier Zeichenketten wird der Operator *+* benutzt, zum Anhängen an die aktuelle Zeichenkette *+=*.

Des weiteren sind die bekannten Vergleichsoperatoren *==*, *!=*, *<*, *>*, *>=* und *<=* definiert. Diese ermöglichen das Vergleichen von zwei Zeichenketten und ersetzen die Member-Funktion *Compare*. Allerdings liefern die Operatoren einen logischen Wert zurück, während *Compare* einen Integer-Wert liefert. Ist dieser kleiner als 0, so ist die Zeichenkette in *CString* kleiner als das Vergleichsobjekt *lpsz*, bei größer als 0 ist sie größer. Nur wenn 0 zurückgegeben wird (*FALSE*), sind sie identisch. Die Methode *CompareNoCase* macht im Gegensatz zu *Compare* keinen Unterschied zwischen Groß- und Kleinschreibung.

```
class CString:  
    const CString& operator =(const CString& stringSrc);  
        throw(CMemoryException);
```



```

const CString& operator =(TCHAR ch);
    throw(CMemoryException);
friend CString operator +(const CString& string1,
                        const CString& string2);
    throw(CMemoryException);
const CString& operator +=(const CString& string);
    throw(CMemoryException);
BOOL operator ==(const CString& s1, const CString& s2);
BOOL operator !=(const CString& s1, const CString& s2);
BOOL operator !=(LPCTSTR s1, const CString& s2);
BOOL operator < (const CString& s1, const CString& s2);
BOOL operator > (const CString& s1, const CString& s2);
BOOL operator <=(const CString& s1, const CString& s2);
BOOL operator >=(const CString& s1, const CString& s2);
int Compare(LPCTSTR lpsz) const;
int CompareNoCase(LPCTSTR lpsz) const;

```

11.1.5 Extrahieren von Teilzeichenketten und weitere Konvertierungen

Um Teilzeichenketten aus einem *CString*-Objekt zu extrahieren, gibt es die Member-Funktionen *Left*, *Right* und *Mid*. Sie arbeiten wie die gleichnamigen Basic-Funktionen: *Left* holt *nCount* Zeichen von links beginnend, *Right* von rechts. *Mid* extrahiert *nCount* Zeichen ab der Position *nFirst*.

```

class CString:
    CString Left(int nCount) const;
        throw(CMemoryException);
    CString Right(int nCount) const;
        throw(CMemoryException);
    CString Mid(int nFirst, int nCount) const;
        throw(CMemoryException);
    void MakeUpper();
    void MakeLower();
    void TrimLeft();
    void TrimRight();

```

Auch kann die gesamte Zeichenkette in Groß- oder in Kleinbuchstaben konvertiert werden. Dazu dienen die Methoden *MakeUpper* und *MakeLower*.

Zum Abschneiden führender oder folgender Leerzeichen dienen die beiden Methoden *TrimLeft* und *TrimRight*.

11.1.6 Suchen von Teilzeichenketten

Die Member-Funktionen *Find* und *ReverseFind* ermöglichen das Suchen von Teilzeichenketten oder einzelnen Zeichen in einem größeren String. *ReverseFind* beginnt die Suche dabei am Ende der Zeichenkette und kann nur einzelne Zeichen suchen.

```
class CString:  
    int Find( TCHAR ch ) const;  
    int Find( LPCTSTR lpszSub ) const;  
    int ReverseFind( TCHAR ch ) const;
```

Neben diesen beschriebenen Methoden der Klasse *CString* existieren noch weitere, die in der Online-Dokumentation nachgelesen werden können. Die Handhabung dieser Klasse sollte kein Problem darstellen.

11.2 Die Klasse CObArray

11.2.1 Die Familie von Feldern

Die Klasse *CObArray* gehört wie auch *CStringArray*, *CWordArray* oder *CUIntArray* zu den Klassen von Feldern. Während *CObArray* als Elemente von *CObjekt* abgeleitete Klassen enthält, sind es bei *CStringArray* *CString*-Objekte oder bei *CUIntArray* Integer-Werte. Die Größe eines Arrays wird nur durch den verfügbaren Arbeitsspeicher begrenzt. Alle Felder besitzen die gleichen Member-Funktionen, nur die Datentypen sind unterschiedlich. Deshalb soll hier exemplarisch die Klasse *CObArray* vorgestellt werden.

11.2.2 Anlegen und Zugriff

Das Anlegen neuer Felder kann entweder auf dem Heap oder dem Stack erfolgen:

```
CObArray array;  
CObArray* pArray = new CObArray;
```

Das Hinzufügen neuer Elemente erfolgt mit der Methode *Add*. Optional kann *InsertAt* verwendet werden, um ein Element an einer bestimmten Position einzufügen. *InsertAt* bietet zudem die Möglichkeit, ein Element mehrfach ein oder ein komplettes Array hinzuzufügen. Da in *CObArray* nur von *CObjekt* abgeleitete Klassen eingefügt werden können, soll zunächst eine eigene Klasse angelegt werden:

```
class Kunde: public CObject  
{ public:  
    int KdNr;  
    CString Name;  
    void SetValue(int Nr, CString name);
```

```

};
Kunde::SetValue(int Nr, CString name)
{
    KdNr = Nr; Name = name;
}
// Anlegen eines neuen Objekts
Kunde *k = new (Kunde);
COBArray feld;
int index;
k->SetValue(1, "Test");
index=feld.Add(k);
// Ändern des Objektes an der Position Index
k->SetValue(1, "Test 2");
feld.SetAt(index,k);
// Index bestimmen und Wert aus Array anzeigen
int nInd = m_Select.GetPos();
int nKDNR = (*(Kunde*)feld[nInd]).KdNr;
CString sName = (*(Kunde*)feld[nInd]).Name;

```

Listing: Eine eigene Klasse, die von CObject abgeleitet wird

Wenn das Array und dessen Elemente auf dem Heap angelegt werden, ist darauf zu achten, daß alle Elemente wieder korrekt freigegeben werden. Ansonsten kommt es schnell zu Memory Leaks. Auch kann die Verwendung eines falschen Indexes zu einem Laufzeitfehler führen.



Die Methode *SetValue* macht nichts anderes, als die Member-Variablen des Objekts *Kunde* zu belegen. Anschließend werden mit *Add* die Objekte dem Array hinzugefügt. Mit der Funktion *SetAt* kann ein Element an der Position *nIndex* durch ein neues Objekt ersetzt werden. Die Member-Funktion *RemoveAt* entfernt *nCount* Elemente ab der Position *nIndex*. Optional kann anstelle von *SetAt* und *GetAt* der Operator `[ ]` verwendet werden, der die Anwendung meist vereinfacht und übersichtlicher macht. Mit der Member-Funktion *GetSize* ist es möglich, die Anzahl der Elemente des Arrays zu ermitteln.

Wichtig zu wissen ist, daß vor dem Entfernen des Arrays vom Heap, sofern es dort angelegt wurde, alle Elemente des Arrays gelöscht werden müssen. Liegen auch diese auf dem Heap, so müssen sie zuerst einzeln mit *delete* entfernt werden. Danach kann mit *RemoveAll* das Array geleert werden.

Zur Demonstration der Arbeitsweise mit Feldern ist das Beispielpogramm *Arrays* auf der CD enthalten.



```
class CObArray:
    int Add(CObject* newElement);
        throw(CMemoryException);
    void InsertAt(int nIndex, CObject* newElement, int nCount = 1);
        throw(CMemoryException);
    void InsertAt(int nStartIndex, CObArray* pNewArray);
        throw(CMemoryException);
    void SetAt(int nIndex, CObject* newElement);
    CObject* GetAt(int nIndex) const;
    void RemoveAt(int nIndex, int nCount = 1);
    CObject*& operator [ ](int nIndex);
    CObject* operator [ ](int nIndex) const;
    int GetSize() const;
    void RemoveAll();
```

Eine wesentliche Eigenschaft von Arrays ist die Fähigkeit zur Serilisation. Damit besteht die Möglichkeit, Daten eines Programms zur Laufzeit einfach im Speicher zu verwalten und diese Daten in ihrer Struktur anschließend auf den Datenträger zu sichern (siehe Kapitel 14.2.3). Dazu ist nicht etwa die Programmierung einer aufwendigen Schleife erforderlich. Es wird lediglich der Name der Variablen angegeben, den Rest erledigen die MFC-Funktionen.

11.3 Die Klasse CImageList

11.3.1 Allgemeines

Die Klasse *CImageList* ist eine von *CObject* abgeleitete Klasse zur Speicherung von Bildern (Bitmaps oder Symbole) gleicher Größe. Die Speicherung erfolgt ähnlich wie in einem Array, so daß über einen Index darauf zugegriffen werden kann. Die Anwendung dieser Klasse erfolgt hauptsächlich im Zusammenhang mit den Steuerelementen für Listen, Baumansicht und Registerkarten (siehe Kapitel »Steuerungen«). Es gibt zwei Arten von Bilderlisten: maskierte und nicht maskierte. Bei maskierten Listen werden zu jedem Bild zwei Teile gespeichert: das eigentliche Bild und eine – zumeist monochrome – Maske. Bei der Darstellung des eigentlichen Bildes wird dieses dann in geeigneter Weise mit der Maske überlagert.

11.3.2 Anlegen einer Image-Liste

Die allgemeine Arbeitsweise mit Bilderlisten sieht so aus, daß zuerst eine Instanz der Klasse *CImageList* mit der Methode *Create* angelegt wird. Danach werden auf verschiedene Arten die einzelnen Bilder hinzugefügt. Ist das erledigt, wird die Liste mit der Steuerung verknüpft und kann genutzt werden.

```
class CImageList:
    BOOL Create(int cx, int cy, UINT nFlags,
               int nInitial, int nGrow);
    BOOL Create(UINT nBitmapID, int cx,
               int nGrow, COLORREF crMask);
    BOOL Create(LPCTSTR lpszBitmapID, int cx,
               int nGrow, COLORREF crMask);
    BOOL Create(CImageList& imagelist1,
               int nImage1, CImageList&
               imagelist2, int nImage2,
               int dx, int dy);
```

Anhand der Methode *Create* erkennt man schon die vielfältigen Möglichkeiten zum Anlegen einer Image-Liste. Falls *Create* nicht korrekt ausgeführt werden kann, wird null zurückgegeben, ansonsten TRUE.

Zuerst muß in jedem Fall der Konstruktor von *CImageList* aufgerufen werden. *Create* legt die Größe der Bilder fest und füllt gegebenenfalls die Liste. Die Parameter *cx* und *cy* geben die Größe des Bildes an. In *nInitial* wird die Anzahl der Bilder festgelegt, für die Speicher reserviert werden soll, und über *nGrow* wird die Anzahl der neu zu reservierenden Speicherplätze festgelegt, wenn *nInitial* überschritten wird. Damit soll das Fragmentieren des Speichers vermindert werden.

Die erste Form von *Create* reserviert nur Speicher für die Bilder, die anschließend noch geladen werden müssen. Die zweite und dritte Form hingegen arbeiten mit den Ressourcen zusammen. Alle Bilder befinden sich aneinander gereiht in einem Bitmap mit der ID *nBitmapID* oder dem Namen *lpszBitmapID*.

Über *cx* wird wiederum die Breite eines Bildes angegeben, so daß die Klasse selbst die Anzahl der Bilder bestimmt und diese auch sofort lädt. Der Parameter *crMask* enthält einen RGB-Wert einer Maske, bei der jedes Pixel des Bildes in der Farbe der Maske schwarz gewandelt wird. Der Parameter *nFlags* bestimmt den Inhalt der Liste genauer, da dieser ja bei der ersten Form der Klasse noch nicht bekannt ist. Über das Flag *ILC_COLORx* wird festgelegt, wieviel Farben das Bitmap enthält. Zusätzlich kann noch *ILC_MASK* angegeben werden, um zu spezifizieren, daß zusätzlich ein Maskenbild für jedes Bitmap existiert. Zur Verdeutlichung hier noch einmal einige Beispiele für das Anlegen von Bilderlisten:

```
pImgList = new CImageList();
ASSERT(pImgList != NULL);
pImgList->Create(32, 32, TRUE, 2, 2);
pImgList->Add(pApp->LoadIcon(IDI_ICON1));
pImgList->Add(pApp->LoadIcon(IDI_ICON2));
```



```

...
CImageList *cIL = new CImageList;
CBitmap cBM;
// Image-Liste mit 32x32 Pixeln erzeugen
// ILC_MASK Maske für den Hintergrund(transparent)
cIL->Create(32,32,ILC_MASK,2,2);
// Bitmap aus Ressource laden und anhängen
cBM.LoadBitmap(IDB_BITMAP1);
cIL->Add(&cBM, (COLORREF)0xFFFFFFFF);
cBM.DeleteObject();
cBM.LoadBitmap(IDB_BITMAP2);
cIL->Add(&cBM, (COLORREF)0xFFFFFFFF);
cBM.DeleteObject();

```

Die nächsten Beispiele laden die Bilder direkt aus einer Bitmap-Ressource (Abbildung 11.1).



```

m_LargeImageList.Create(IDB_LARGEICONS, 32, 1,
                        RGB(255, 255, 255));
m_StateImageList.Create(IDB_STATEICONS, 16, 1,
                        RGB(255, 0, 0));

```



Abbildung 11.1: Das Bitmap IDB_LARGEICONS

11.3.3 Hinzufügen und Löschen von Bildern

Die Klasse *CImageList* stellt einige Methoden zum Laden, Ändern und Löschen der einzelnen Elemente der Bilderliste bereit:

```

class CImageList:
    int Add(CBitmap* pbmImage, CBitmap* pbmMask);
    int Add(CBitmap* pbmImage, COLORREF crMask);
    int Add(HICON hIcon);
    BOOL Remove(int nIndex);
    BOOL Replace(int nIndex, CBitmap* pbmImage, CBitmap* pbmMask);
    int Replace(int nIndex, HICON hIcon);
    HICON ExtractIcon(int nIndex);

```

Mit Hilfe der verschiedenen Varianten der Methode *Add* wird ein neues Bild zur Liste hinzugefügt. *Remove* entfernt ein Bild an der Position *nImage*, und *Replace* ersetzt eines der Bilder durch ein anderes. Man kann ein Bild auch wieder aus der Liste herausholen. Dazu dient die Methode *ExtractIcon*.

Des weiteren können Bilder der Liste als sogenannte Overlays definiert werden. Diese werden dann dazu benutzt, um das Overlay-Bild auf das Bild quasi transparent zu legen. Ein Beispiel für die Anwendung dieser Technik ist das Programm ROWLIST, welches als Beispielprogramm im Visual C++-Programmpaket enthalten ist.

Weitere Hinweise zur Verwendung von Image-Listen sind im *Kapitel »Steuerungen«* sowie in der Online-Dokumentation unter *Verwenden von CImageList* zu finden.



11.4 Die Symbolleiste

11.4.1 Allgemeines

Die Symbolleiste ist fester Bestandteil eines jeden Windows-Programms. Keiner möchte mehr den Schnellzugriff auf oft benötigte Funktionen vermissen. Da eine Symbolleiste meist mangels Platz gar nicht alle Symbole aufnehmen kann (siehe Winword), ist es in vielen Programmen möglich, zwischen verschiedenen Symbolleisten zu wechseln und diese teilweise auch selbst anzupassen. Mit dem Internet Explorer und Microsoft Office 97 hielt dann auch noch ein neues flaches Aussehen der Symbolleiste Einzug, das heute jedes Programm schmückt, was etwas auf sich hält.

In einigen der vorangegangenen Beispiele haben wir schon gelernt, wie die Standard-Symbolleiste, die vom Anwendungs-Assistenten erzeugt wird, angepaßt wird. Hierbei wurden aber ausschließlich die Schaltflächen verändert und angepaßt. In diesem Abschnitt soll auch gezeigt werden, wie man zum Beispiel eine Drop-down-Liste einfügt.

11.4.2 Standard-Code vom Anwendungs-Assistenten

Wenn der Anwendungs-Assistent verwendet wird, haben Sie in Schritt 4 die Möglichkeit, eine Standard-Symbolleiste mit anlegen zu lassen. In den weiteren Optionen kann man noch das Aussehen dieser Symbolleiste als *Normal* und *Internet Explorer-Infoleisten* wählen.

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;
    if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT,
        WS_CHILD | WS_VISIBLE | CBRS_TOP | CBRS_GRIPPER |
        CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC) ||
        !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
    {
        TRACE0("Symbolleiste konnte nicht erstellt werden\n");
        return -1;        // Fehler bei Erstellung
    }
}
```

```

...
m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
EnableDocking(CBRS_ALIGN_ANY);
DockControlBar(&m_wndToolBar);
return 0;
}

```

Listing: Anlegen einer »normalen« Symbolleiste



Abbildung 11.2: Die »normale« Symbolleiste in der Anwendung

```

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;
    if (!m_wndToolBar.CreateEx(this) ||
        !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
    {
        TRACE0("Symbolleiste konnte nicht erstellt werden\n");
        return -1;        // Fehler bei Erstellung
    }
    if (!m_wndDlgBar.Create(this, IDR_MAINFRAME,
                           CBRS_ALIGN_TOP, AFX_IDW_DIALOGBAR))
    {
        TRACE0("Dialogleiste konnte nicht erstellt werden\n");
        return -1;        // Fehler bei Erstellung
    }
    if (!m_wndReBar.Create(this) || !m_wndReBar.AddBar(&m_wndToolBar)
    ||
        !m_wndReBar.AddBar(&m_wndDlgBar))
    {
        TRACE0("Infoleiste konnte nicht erstellt werden\n");
        return -1;        // Fehler bei Erstellung
    }
}

```



```

m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
    CBRS_TOOLTIPS | CBRS_FLYBY);
return 0;
}

```

Listing: Anlegen einer Symbolleiste ähnlich dem Internet Explorer

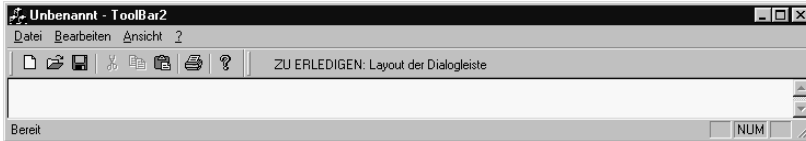


Abbildung 11.3: Die Symbolleiste mit Internet Explorer-Infoleisten

Die erste Variante legt, wie schon bekannt, eine ganz normale Symbolleiste an. Das Attribut `TBSTYLE_FLAT` verleiht ihr das flache Aussehen. Dieses Attribut ist nur verfügbar, wenn die Symbolleiste mit *CreateEx* angelegt wird. `CBRS_GRIPPER` erzeugt den kleinen hervorgehobenen Balken am linken Rand der Symbolleiste, um diese zum Verschieben mit der Maus anzufassen.

Das zweite Beispiel verwendet die Infoleisten (ReBars). Die Infoleiste ist ein Container-Fenster für sogenannte Bänder. Ein Band kann aus einem Schiebegriff, Text, Bitmap und untergeordnetem Fenster bestehen. Die untergeordneten Fenster können zum Beispiel eine Symbolleiste oder eine Dialogleiste sein. Mit der Methode *AddBar* wird ein Band hinzugefügt.

```

class CReBar:
    BOOL AddBar( CWnd* pBar, LPCTSTR lpszText = NULL,
                CBitmap* pbmp = NULL,
                DWORD dwStyle = RBBS_GRIPPERALWAYS | RBBS_FIXEDBMP
    );
    BOOL AddBar( CWnd* pBar, COLORREF clrFore, COLORREF clrBack,
                LPCTSTR pszText = NULL,
                DWORD dwStyle = RBBS_GRIPPERALWAYS );

```

11.4.3 Laden und Speichern der Einstellungen

Wenn der Benutzer im Programm die Position von Symbolleisten verändert hat, ist es sinnvoll, diese für den nächsten Start der Applikation zu speichern. So muß der Benutzer nicht jedesmal seine Symbolleisten selbst zurechtschieben. Die MFC unterstützt das Sichern und Wiederherstellen des Toolbar-Status in der Klasse *CFrameWnd* mit den beiden Member-Funktionen *SaveBarState* und *LoadBarState*.

```
class CFrameWnd:
    void LoadBarState( LPCTSTR lpszProfileName );
    void SaveBarState( LPCTSTR lpszProfileName ) const;
```

Der Parameter *lpszProfileName* ist eine Zeichenkette für den Schlüssel, unter dem die Einstellungen gespeichert oder geladen werden. Je nach Einstellung in *InitInstance* werden die Informationen in einer INI-Datei oder der Registrierung abgelegt. Gespeichert werden die Sichtbarkeit, die Position, der Docking-Status und die Orientierung der Symbolleiste. Das Speichern der Informationen sollte vor dem Beenden des Programms in der Methode *OnClose* erfolgen. Das Wiederherstellen der Symbolleisten erfolgt am günstigsten in der Methode *OnCreate* nach dem Anlegen der Symbolleisten.

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;
    if (!m_wndToolBar.Create(this) ||
        !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
    {
        TRACE0("Failed to create toolbar\n");
        return -1;      // fail to create
    }
    m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
        CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);
    m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
    EnableDocking(CBRS_ALIGN_ANY);
    DockControlBar(&m_wndToolBar);
    LoadBarState("Toolbar1");
    ...
}

void CMainFrame::OnClose()
{
    SaveBarState("Toolbar1");
    CFrameWnd::OnClose();
}
```

Listing: Laden und Speichern von Symbolleisten

Wenn Sie mehrere Symbolleisten in Ihrer Applikation verwenden, darf *LoadBarState* erst nach dem Anlegen aller Symbolleisten aufgerufen werden.

11.4.4 Mehrere Symbolleisten

Wenn Sie mehrere Symbolleisten verwenden wollen, sollten alle in `CMainFrame::OnCreate` initialisiert werden. Erst danach kann auch `LoadBarState` aufgerufen werden. Zu beachten ist, daß jede Symbolleiste ihre eigene ID erhält. Diese ID wird unter anderem beim Speichern der Toolbar-Informationen verwendet. Wird keine ID angegeben, bedeutet das, daß die Standard-ID der Methode `CToolBar::Create` benutzt wird. Das kann dazu führen, daß die Anwendung nach einigen Starts mit einer Fehlermeldung abbricht. In `AFXRES.H` ist ein Bereich für die IDs der Symbolleisten vorgesehen. Dieser liegt zwischen `AFX_IDW_CONTROLBAR_FIRST` und `AFX_IDW_CONTROLBAR_LAST`.

```
...
if(!m_wndToolBar2.Create(this,WS_CHILD | WS_VISIBLE | CBRS_LEFT,
                        AFX_IDW_CONTROLBAR_LAST)
    || !m_wndToolBar2.LoadToolBar(IDR_TOOLBAR2))
{
    TRACE0("Statusleiste 2 konnte nicht erstellt werden\n");
    return -1;
}
if(!m_wndToolBar3.Create(this,WS_CHILD | WS_VISIBLE | CBRS_TOP,
                        AFX_IDW_CONTROLBAR_LAST-1)
    || !m_wndToolBar3.LoadToolBar(IDR_TOOLBAR3))
{
    TRACE0("Statusleiste 3 konnte nicht erstellt werden\n");
    return -1;
}
m_wndToolBar3.SetWindowText("Toolbar 3");
m_wndToolBar3.EnableDocking(0);
CPoint pt(200,200);
FloatControlBar(&m_wndToolBar3,pt);
LoadBarState("ToolBar1");
```

Listing: Das Anlegen weiterer Symbolleisten in `CMainFrame::OnCreate`

Die im Beispiel angegebene Symbolleiste `m_wndToolBar2` wird fest am linken Rand andockt. Bei der nächsten Symbolleiste wird das Andocken verhindert. Diese ist eine freie Tool-Palette, die beim ersten Start der Anwendung mit `FloatControlBar` auf eine feste Position gesetzt wird. Zu beachten ist der dritte Parameter der Funktion `Create`. Diese beinhaltet die ID der Symbolleiste. Der Parameter kann auch per `#define`-Statement im Header definiert werden.

11.4.5 Text-Label

Die Klasse `CToolBar` enthält die Methode `SetButtonText`, die es ermöglicht, außer dem Bild noch einen Text anzuzeigen.

```
class CToolBar:
    void SetSizes( SIZE sizeButton, SIZE sizeImage );
    BOOL SetButtonText( int nIndex, LPCTSTR lpszText );
    UINT GetItemID( int nIndex ) const;
```

Einige wenige Zeilen Quelltext genügen, um den Text, der als Tooltip angezeigt wird, für den Text der Symbole der Toolbar zu nutzen. In einer Schleife wird für jedes Element der Symbolleiste die ID ermittelt und der zugehörige Text aus den Ressourcen geladen. Diese Zeichenkette ist so aufgebaut, daß nach einem Zeilenwechsel (\n) der Text für den Tooltip folgt. Dieser wird, falls vorhanden, extrahiert und dem Button zugewiesen.

```
for(i = 0; i < m_wndToolBar.GetCount(); i++)
{
    sText.LoadString(m_wndToolBar.GetItemID(i));
    if((j=sText.Find("\n")) < 0) continue;
    sText = sText.Mid(j+1);
    m_wndToolBar.SetButtonText(i,sText);
}
// Anpassen der Größe
CRect rect;
m_wndToolBar.GetItemRect(0,&rect);
m_wndToolBar.SetSizes(rect.Size(),CSize(16,16));
```

Listing: Text der Symbole der Toolbar aus den Tooltips generieren

Im Anschluß daran ist es noch erforderlich, die Größe der Symbole der Toolbar neu festzulegen. Dazu wird als erstes die Gesamtgröße eines Elements ermittelt. Da die Größe der Bilder bekannt ist, kann sie mit *SetSizes* neu definiert werden.

11.4.6 Drop-down-Listen

Die neuen erweiterten Attribute *TBSTYLE_DROPDOWN* und *TBSTYLE_EX_DRAWDDARROWS* ermöglichen, aus einer einfachen Schaltfläche eine Drop-down-Liste zu erzeugen. Zuerst muß das Attribut *TBSTYLE_EX_DRAWDDARROWS* der Symbolleiste zugewiesen werden, um die Möglichkeit der Drop-down-Listen zu aktivieren. Dann erhält jede Schaltfläche, die eine Drop-down-Liste sein soll, das Attribut *TBSTYLE_DROPDOWN*.

```
UINT id,style;
int i,iBild;
m_wndToolBar.GetToolBarCtrl().SetExtendedStyle(TBSTYLE_EX_DRAWDDARRO
WS);
```

```
i = m_wndToolBar.CommandToIndex(ID_FILE_OPEN);
m_wndToolBar.GetButtonInfo(i,id,style,iBild);
m_wndToolBar.SetButtonInfo(i,ID_FILE_OPEN,TBSTYLE_BUTTON |
                           TBSTYLE_DROPDOWN,iBild);
```

Listing: Öffnen-Button mit Drop-down-Liste



Abbildung 11.4: Öffnen-Schaltfläche mit Drop-down-Liste

Bei einem Mausklick auf die Schaltfläche wird die normale Funktion aufgerufen. Wird die Pfeiltaste neben der Schaltfläche betätigt, wird die Nachricht `TBN_DROPDOWN` an das Elternfenster geschickt. Diese wird in eine Notify-Nachricht verpackt. Hier muß von Hand in die Message-Map eingegriffen und eine Nachrichtenbehandlung hinzugefügt werden.

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    //{AFX_MSG_MAP(CMainFrame)
    ON_WM_CREATE()
    //}}AFX_MSG_MAP
    ON_NOTIFY(TBN_DROPDOWN,AFX_IDW_TOOLBAR,OnDropDown)
END_MESSAGE_MAP()

void CMainFrame::OnDropDown(NMHDR* pNotifyStruct, LRESULT* pResult)
{
    //Zeiger umwandeln
    NMTOOLBAR* pNMToolBar = (NMTOOLBAR*)pNotifyStruct;
    CRect rect;
    //Position für Popup-Menü festlegen
    m_wndToolBar.GetToolBarCtrl().GetRect(pNMToolBar->iItem, &rect);
    rect.top = rect.bottom;
    ::ClientToScreen(pNMToolBar->hdr.hwndFrom, &rect.TopLeft());
    //bei Öffnen-Drop-Down
    if(pNMToolBar->iItem == ID_FILE_OPEN)
    {
        CMenu menu;
        CMenu* pPopup;
        menu.LoadMenu(IDR_F_OPEN);
```

```

        pPopup = menu.GetSubMenu(0);
        pPopup->TrackPopupMenu(TPM_LEFTALIGN | TPM_LEFTBUTTON,
rect.left,
                                rect.top + 1, AfxGetMainWnd());
    }
    *pResult = TBDDRET_DEFAULT;
}

```

Listing: Nachrichtenbehandlung einer Drop-down-Liste

Die Notify-Struktur wird zuerst typisiert. Danach wird die Position für das zu öffnende Popup-Menü bestimmt. Dieses Menü kann als ganz normale Ressource angelegt werden. Schließlich wird noch überprüft, welche Schaltfläche die Funktion ausgelöst hat und das Menü anzeigt. Um auf die Menüpunkte zu reagieren, werden Nachrichtenbehandlungs-Routinen mit dem Klassenassistenten eingefügt. Zuvor muß das Menü mit der Anichts-Klasse verbunden werden. Die Nachrichten-Funktionen führen dann die gewünschten Funktionen aus.

```

void CToolBar2View::OnOpenAutoexecbat()
{
    AfxGetApp()->OpenDocumentFile("C:\\AUTOEXEC.BAT");
}
void CToolBar2View::OnOpenConfigsys()
{
    AfxGetApp()->OpenDocumentFile("C:\\CONFIG.SYS");
}

```



Sehen Sie sich das mitgelieferte Beispielprogramm MFCIE einmal etwas genauer an. Dort sind einige Möglichkeiten der Anwendung der neuen Symbolleisten demonstriert.

11.5 Die Klasse CFile

11.5.1 Grundlagen

Die Klasse *CFile* ist in der MFC für die Operationen mit Dateien zuständig. Im Unterschied zu der objektorientierten Serialisation von Daten (siehe Kapitel 13.2.3) repräsentiert diese Klasse eher den konventionellen Umgang mit Dateien. Sie stellt eine Schnittstelle zu Binärdateien zur Verfügung. Der interne Zugriff anderer Klassen in der MFC auf Dateien wird meist über die Klasse *CFile* oder eine davon abgeleitete realisiert.

CFile kapselt die Dateioperationen der Laufzeitbibliothek in eine komfortable Klasse. Wenn Sie bereits mit C-Funktionen auf Dateien zugegriffen haben, fällt Ihnen die Nutzung dieser Klasse leicht. Es steht neben den

Methoden zum Öffnen, Schließen, Lesen und Schreiben auch eine integrierte Ausnahmebehandlung zur Verfügung.

Neben dem Standardkonstruktor besitzt die Klasse noch zwei weitere Konstruktoren. Der erste erwartet als Parameter `hFile` ein Handle auf eine bereits geöffnete Datei. Diesen Konstruktor können Sie z.B. verwenden, wenn Sie in Ihrem Programm eine C-Schnittstelle benutzen, die ein File-Handle zur Verfügung stellt, und Sie wollen damit objektorientiert weiterarbeiten. Der zweite zusätzliche Konstruktor erwartet in `lpszFileName` den Dateinamen mit einer relativen oder absoluten Pfadangabe und in `nOpenFlags` den Modus, mit dem die Datei geöffnet werden soll.

```
CFile::CFile  
    CFile( );  
    CFile(int hFile);  
    CFile(LPCTSTR lpszFileName, UINT nOpenFlags);  
        throw(CFileException);
```

Der Standardkonstruktor kann die Datei nicht öffnen. Die Member-Variable `m_hFile` wird auf `NULL` gesetzt. Wird der Konstruktor mit dem Handle als Parameter benutzt, wird die Datei im Destruktor nicht geschlossen.



11.5.2 Hierarchie und abgeleitete Klassen

Die Klasse *CFile* kann nicht nur direkt als Instanz eingesetzt werden, sie ist auch gleichzeitig die Basisklasse für eine Reihe von Klassen. Dabei lassen sich vier Hauptgruppen unterscheiden.

CMemFile ist eine Ableitung, die eine Datei im Speicher darstellt. Diese Datei ist für temporäre Daten mit einem schnellen Zugriff gedacht. Außerdem können so auch Daten zwischen verschiedenen Prozessen ausgetauscht werden.

COleStreamFile repräsentiert einen Datenstrom in Form von einem COM *IStream* in eine Verbunddatei. Das *IStream* Objekt muß bereits existieren bevor Sie das *COleStreamFile* Objekt anlegen.

CSocketFile stellt eine Datei dar, die über Windows Sockets im Netzwerk kommunizieren kann. Diese Klasse kann z.B. mit *CArchive* zusammen zum Senden und Empfangen von Daten in einem Netzwerk verwendet werden.

CStdioFile ist eine interessante Erweiterung von *CFile*, wenn es um Textdateien geht. Diese Klasse stellt Funktionen zur Verfügung, mit denen Sie zeilenweise lesen und schreiben können.

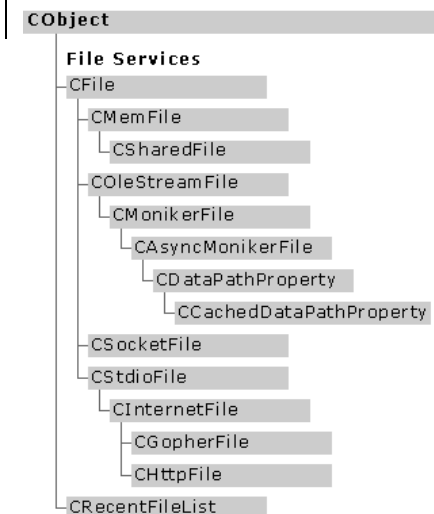


Abbildung 11.5: Klassenhierarchie von CFile

Einige dieser abgeleiteten Klassen sind selbst wieder Basisklassen für weitere Klassen. Außerdem müssen Sie darauf achten, daß nicht alle Methoden von *CFile* in den abgeleiteten Klassen unterstützt werden.

11.5.3 Die wichtigsten Methoden

Die Klasse *CFile* besitzt eine Vielzahl von Member-Funktionen für die Benutzung einer Datei. Hier wird nur eine Auswahl vorgestellt, die Sie aber in die Lage versetzt, mit einer Datei die wichtigsten Operationen durchzuführen.

Bevor Sie mit den Daten einer Datei arbeiten können, muß diese geöffnet werden. Dazu gibt es zwei Möglichkeiten. Die erste wurde bereits erwähnt, der Konstruktor. Wenn Sie den Konstruktor mit den beiden Parametern *lpszFileName* und *nOpenFlags* benutzen, wird versucht, die angegebene Datei im gewählten Modus zu öffnen. Die zweite Möglichkeit zum Öffnen einer Datei ist die Methode *Open*. Sie besitzt die gleichen Parameter wie der Konstruktor zum Öffnen.

```

CFile::Open
virtual BOOL Open(LPCTSTR lpszFileName, UINT nOpenFlags,
                  CFileException* pError = NULL);
  
```

Diese Methode liefert 0 zurück, wenn ein Fehler beim Öffnen aufgetreten ist. Dann steht in der Ausnahmebehandlung *pError* die Fehlerursache. Die Auswertung ist im nächsten Abschnitt beschrieben.

Wie auch beim Konstruktor enthält der Parameter `lpszFileName` den Dateinamen mit einer relativen oder absoluten Pfadangabe. Der zweite Parameter `nOpenFlags` gibt an, in welchem Modus bzw. mit welchen Berechtigungsattributen Sie die Datei öffnen wollen. Die möglichen Modi sind in Tabelle 11.1 beschrieben. Die einzelnen Flags können mit dem Bit-Operator `|` verknüpft werden.

Flag	Bedeutung
<code>CFile::modeCreate</code>	Die Datei wird neu angelegt. Eine vorhandene wird überschrieben.
<code>CFile::modeNoTruncate</code>	Nur zusammen mit <code>modeCreate</code> . Wenn die Datei bereits vorhanden ist, wird sie nicht überschrieben.
<code>CFile::modeRead</code>	Nur zum Lesen öffnen.
<code>CFile::modeReadWrite</code>	Öffnen zum Lesen und Schreiben.
<code>CFile::modeWrite</code>	Nur zum Schreiben öffnen.
<code>CFile::modeNoInherit</code>	Verhindert das Vererben der Datei an einen Kind-Prozeß.
<code>CFile::shareDenyNone</code>	Die Datei wird geöffnet, und andere Prozesse können die Datei lesen und schreiben.
<code>CFile::shareDenyRead</code>	Die Datei wird geöffnet, und andere Prozesse können die Datei lesen.
<code>CFile::shareDenyWrite</code>	Die Datei wird geöffnet, und andere Prozesse können die Datei schreiben.
<code>CFile::shareExclusive</code>	Öffnen der Datei im Exklusiv-Mode. Kein anderer Prozeß kann lesen oder schreiben.
<code>CFile::shareCompat</code>	Wie <code>shareExclusive</code> (kompatibel zu 16 Bit MFC).
<code>CFile::typeText</code>	Setzt den Text-Mode für die Datei. Die Steuerzeichen für »neue Zeile« werden entsprechend berücksichtigt (nicht für alle Klassen).
<code>CFile::typeBinary</code>	Setzt den Binär-Mode für die Datei (Gegenteil zu <code>typeText</code>).

Tabelle 11.1: Modi zum Öffnen einer Datei

Bei den Flags ist zu beachten, daß einige wie `typeText` nur in abgeleiteten Klassen und andere nur in Kombination verwendet werden können.

```
CFile myFile;
CFileException fileException;
if (!myFile.Open("myDat.dat", CFile::modeCreate |
                    CFile::modeReadWrite,
                    &fileException)
{
    TRACE("Fehler %u beim Öffnen der Datei.\n",
        fileException.m_cause);
}
```

Listing: Öffnen einer Datei zum Lesen und Schreiben

In diesem Beispiel wird eine neue Datei erstellt und zum Lesen und Schreiben geöffnet. Schlägt dieses fehl, wird mit dem Makro `TRACE` eine Meldung ausgegeben.

Eine Datei muß auch wieder geschlossen werden. Auch hier gibt es zwei Möglichkeiten. Die erste besteht darin, daß Sie das Schließen vom Destruktor der Klasse *CFile* ausführen lassen. Wenn beim Auflösen des Objektes die Datei noch geöffnet ist, wird die Methode *Abort* aufgerufen.

```
CFile::Abort  
virtual void Abort();
```

Dies ist eine etwas radikale Art, eine Datei zu schließen, denn sie ignoriert alle Fehler und Warnungen. Sie haben keine Kontrolle, ob Ihre Daten auch ordnungsgemäß zurückgeschrieben werden konnten. Sehen Sie diese Methode nur als Sicherheitsmechanismus von *CFile* an, wenn Sie einmal »vergessen«, die Datei ordnungsgemäß zu schließen oder die Datei nur zum Lesen geöffnet haben.

Die zweite und bessere Möglichkeit, eine offene Datei zu schließen, ist die Methode *Close*. Diese schließt nicht nur die Datei, sondern leert vorher gegebenenfalls noch die Puffer.

```
CFile::Close  
virtual void Close();  
throw(CFileException);
```

Außerdem bietet diese Methode auch eine Ausnahmebehandlung, mit der Sie auf Fehlerzustände reagieren können.

Um aus einer Datei zu lesen, stellt die Klasse *CFile* zwei weitgehend identische Methoden *Read* und *ReadHuge* zur Verfügung. Sie unterscheiden sich nur durch die mögliche Anzahl zu lesender Zeichen. Die ältere Methode *Read* kann »nur« maximal 64kByte -1 Byte auf einmal einlesen, die Methode *ReadHuge* dagegen kann 4 GByte einlesen.

```
CFile::Read  
virtual UINT Read(void* lpBuf, UINT nCount);  
throw(CFileException);  
CFile::ReadHuge  
DWORD ReadHuge(void* lpBuffer, DWORD dwCount);  
throw(CFileException);
```

Die Parameter haben dabei folgende Bedeutung: Der erste *lpBuf* ist ein Zeiger auf einen Puffer im Speicher, der die Daten der Datei aufnehmen soll. Der zweite Parameter *nCount* gibt die Anzahl der zu lesenden Bytes an.

Sie müssen dafür sorgen, daß der Puffer auch die erforderliche Größe hat, um alle Bytes aufzunehmen. Der Rückgabewert der Methode ist die Anzahl der tatsächlich gelesenen Bytes. Er entspricht `nCount` außer dann, wenn das Ende der Datei (EOF) erreicht wurde und nicht mehr die angeforderte Anzahl Bytes in der Datei vorhanden ist. Die Methode *Read* unterstützt auch die Ausnahmebehandlung.

```
// ...
// Datei ist bereits zum Lesen geöffnet!
char pbuf[100];
UINT nBytesRead;
nBytesRead = myFile.Read(pbuf, 100);
```

Listing: Lesen in einer Datei

Bei jedem Lesen in der Datei wird der Datenzeiger um die Anzahl der gelesenen Bytes erhöht. Um aber ohne einen Lesevorgang an eine bestimmte Stelle in der Datei zu kommen, steht die Methode *Seek* zur Verfügung.

```
CFile::Seek
virtual LONG Seek(LONG lOff, UINT nFrom);
throw(CFileException);
```

Dabei bedeutet `lOff` die Anzahl Bytes, um die der Zeiger versetzt werden soll. Der Parameter kann positiv oder negativ sein. Mit `nFrom` wird der Methode mitgeteilt, welcher Ausgangspunkt verwendet werden soll (siehe Tabelle 11.2).

Modus	Bedeutung
<i>CFile::begin</i>	Positionieren vom Anfang der Datei aus.
<i>CFile::current</i>	Versetzen ab der aktuellen Position.
<i>CFile::end</i>	Positionieren vom Ende der Datei aus. <i>lOff</i> muß dann negativ sein.

Tabelle 11.2: Ausgangspunkt beim Positionieren in einer Datei

Bei der letzten wichtigen Methode der Klasse *CFile* geht es um das Schreiben in eine Datei. Die Funktionalität ähnelt sehr dem Lesen. Auch hier gibt es zwei Methoden: *Write* und *WriteHuge*. Die beiden Parameter der Methoden haben die identische Bedeutung wie bei *Read* und *ReadHuge*. `lpBuf` ist der Zeiger auf den Puffer, in den geschrieben werden soll, und in `nCount` steht die Anzahl der Bytes.

```

CFile::Write
    virtual void Write( const void* lpBuf, UINT nCount );
    throw( CFileException );
CFile::WriteHuge
    void WriteHuge( const void* lpBuf, DWORD dwCount );
    throw( CFileException );

```

Der einzige Unterschied besteht im Rückgabewert der Methoden. Die Schreibfunktionen geben **nicht** die Anzahl der geschriebenen Bytes zurück. Auch *Write* unterstützt die Ausnahmebehandlung.

```

char  szBuffer[] = "Dieser Text soll in die Datei \
                    geschrieben werden!";

UINT  nPos = 0;
CFile myFile;
myFile.Open("myDat.dat", CFile::modeCreate |
            CFile::modeReadWrite);
myFile.Write(szBuffer, sizeof(szBuffer));
myFile.Seek(0, CFile::begin);
nPos = myFile.Read(szBuffer, sizeof(szBuffer));
myFile.Close();

```

Listing: Lesen und Schreiben in eine Datei

11.5.4 Ausnahmebehandlung

Gerade beim Umgang mit Dateien ist eine konsequente Ausnahmebehandlung wichtig. Der Fehlermöglichkeiten gibt es viele. Außer Programmierfehlern, wie z.B. unzulässiger Pfad- oder Dateinamen, können auch Fehler vom Betriebssystem oder der Hardware ausgehen. Auch Zugriffskonflikte auf Grund unzureichender Berechtigung können zu Fehlern führen. Stellen Sie sich vor, Sie wollen Ihre wertvollen Daten in eine Datei schreiben, und dabei treten Fehler auf; dann müssen Sie darauf reagieren können.

CFileException ist die Ausnahmeklasse der MFC für die Dateibearbeitung. Sie wird von der MFC immer dann benutzt, wenn beim Arbeiten mit der Klasse *CFile* oder davon abgeleiteten Klassen ein Fehler aufgetreten ist. Dabei sind zwei verschiedene Anwendungsmöglichkeiten implementiert.

Bei der Methode *Open* gibt es einen optionalen dritten Parameter, einen Zeiger auf ein *CFileException* Objekt. Die Methode *Open* liefert bereits 0 als Rückgabewert, wenn ein Fehler beim Öffnen der Datei aufgetreten ist. Über die Ausnahmeklasse können Sie, wenn Sie wollen, die genaue Ursache ermitteln.

Die zweite Anwendung findet die Klasse bei den Methoden *Read* und *Write*. Wenn beim Lesen oder Schreiben ein Fehler auftritt, wird eine Ausnahme ausgelöst, die Sie dann in einem *try/catch*-Blocks auswerten können.

Die Klasse besitzt Member-Variablen, die den aufgetretenen Fehlercode enthalten. In *m_cause* steht der Fehlerstatus der Ausnahmeklasse. Er hat einen der folgenden Werte:

Fehlerstatus	Bedeutung
<i>CFileException::none</i>	Kein Fehler aufgetreten.
<i>CFileException::generic</i>	Ein allgemeiner Fehler ist aufgetreten.
<i>CFileException::fileNotFound</i>	Die angegebene Datei konnte nicht gefunden werden.
<i>CFileException::badPath</i>	Der Pfad zur Datei oder Teile davon sind ungültig.
<i>CFileException::tooManyOpenFiles</i>	Die maximale Anzahl gleichzeitig geöffneter Dateien ist überschritten.
<i>CFileException::accessDenied</i>	Keine Berechtigung, um auf diese Datei zuzugreifen.
<i>CFileException::invalidFile</i>	Die benutzte Datei ist nicht gültig.
<i>CFileException::removeCurrentDir</i>	Das aktuelle Arbeitsverzeichnis kann nicht gelöscht werden.
<i>CFileException::directoryFull</i>	Kein freier Verzeichniseintrag mehr vorhanden.
<i>CFileException::badSeek</i>	Der Datenzeiger wurde außerhalb des zulässigen Bereichs positioniert.
<i>CFileException::hardIO</i>	Hardwarefehler.
<i>CFileException::sharingViolation</i>	SHERE.EXE ist nicht gestartet, oder der Datenbereich ist gesperrt.
<i>CFileException::lockViolation</i>	Ein zu sperrender Bereich ist bereits gesperrt.
<i>CFileException::diskFull</i>	Der Datenträger ist voll.
<i>CFileException::endOfFile</i>	Das Ende der Datei wurde erreicht.

Tabelle 11.3: Inhalte und Bedeutung von *m_cause*

In der Variablen *m_IOException* steht die Fehlernummer, die das Betriebssystem gemeldet hat, und auf *m_strFileName* der Name der betroffenen Datei.

```
try
{
    myFile.Write(szBuffer, sizeof(szBuffer));
}
catch(CFileException* e)
{
    e->ReportError();
    e->Delete()
}
```

Listing: Lokale Behandlung von Ausnahmen

11.5.5 Ein Beispiel

Die interessante Klasse *CFile* soll nun in einem kurzen, aber anschaulichen Beispiel zur Anwendung kommen. Dabei wird auf die abgeleitete Klasse *CStdioFile* zurückgegriffen. Außerdem wird eine eigene Klasse von *CListBox* abgeleitet, die eine Drag&Drop-Funktion unterstützt. An dieser Stelle wird nicht die Funktionalität für das Anlegen von eigenen Dialogfeldklassen und das Erstellen von Dialogvorlagen erläutert. Dazu lesen Sie bitte unter »Dialogfelder und die Klasse *CDialog*« nach.

In dem Beispiel besteht ein Dialogfeld aus drei Listenfeldern, die die Daten aus verschiedenen Textdateien aufnehmen sollen. Dabei liefern zwei Listenfelder die Ausgangstexte für das dritte Listenfeld. Zum Öffnen und Speichern der Texte wird der Dialog *Datei öffnen* verwendet. Das dritte Listenfeld wird durch einfaches Ziehen der Texte mit der Maus aus den beiden anderen Listenfeldern gefüllt.

Um dieses zu ermöglichen, müssen Sie eine eigene Klasse schreiben, die von *CListBox* abgeleitet ist. Dazu aktivieren Sie den Klassenassistenten und gelangen über die Schaltfläche **KLASSE HINZUFÜGEN** zum Dialog **NEUE KLASSE**. Dort tragen Sie den Namen *CDragListbox* für die neue Klasse ein und wählen *CListBox* als Basisklasse.

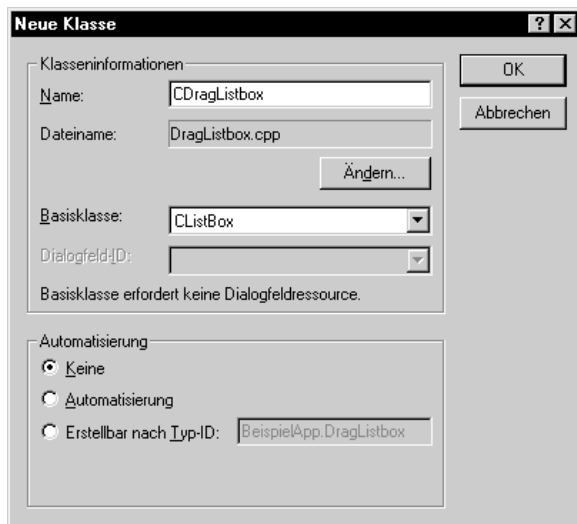


Abbildung 11.6: Neue Listfeld-Klasse

Um diese Klasse mit der Drag&Drop-Funktionalität zu versehen, müssen nur zwei Methoden überlagert werden. Sie müssen nur auf die Nachrichten beim Drücken (*WM_LBUTTONDOWN*) und Loslassen (*WM_*

LBUTTONUP) der Maustaste reagieren. Legen Sie mit dem Klassenassistenten die beiden Methoden für diese Nachrichten in der neuen Klasse `CDragListBox` an. Sie heißen *OnLButtonDown* und *OnLButtonUp* und wurden bereits von `CWnd` ererbt.

Um dem Ganzen ein professionelles Aussehen zu geben, soll sich der Mauszeiger beim Drücken der rechten Maustaste verändern, so lange, bis die Taste wieder losgelassen wird. Dazu verwenden Sie am besten den für das Kopieren bekannten Cursor.

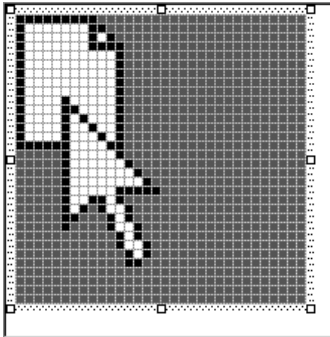


Abbildung 11.7: Der Kopie-Cursor

Nun kann es an das Ausprogrammieren der beiden Methoden gehen. Beim Drücken der Maustaste soll sich der Cursor ändern. Dazu dient die Methode *SetCursor*, die als Parameter ein Handle auf den neuen Cursor erwartet und das Handle des alten Cursors zurückliefert.

```
HCURSOR SetCursor(HCURSOR hCursor
                  // handle to cursor);
```

In der Anwendung sieht das dann so aus:

```
void CDragListBox::OnLButtonDown(UINT nFlags, CPoint point)
{
    hcOld = SetCursor(AfxGetApp()->LoadCursor(IDC_CURSOR));
    CListBox::OnLButtonDown(nFlags, point);
}
```

Listing: Ändern des aktuellen Cursors

Beim Loslassen der Maustaste ist einiges mehr zu tun. Als erstes müssen Sie prüfen, ob die Maustaste über einem Fenster losgelassen wurde. Das geschieht mit Hilfe der Methode *GetCapture* von `CWnd`. Dann muß das Fenster unter der Maus ermittelt werden. Voraussetzung dafür ist das Um-

rechnen der Koordinaten von Client- auf Bildschirmformat. Das übernimmt die Methode *ClientToScreen* der Klasse *CWnd*. Mit den so gewonnenen Koordinaten kann mit einer weiteren Methode von *CWnd*, *WindowFromPoint*, endlich das eigentliche Fenster bestimmt werden. Jetzt müssen Sie nur noch ausschließen, daß dieses Fenster gar nicht in dem Dialog, in dem die Listenfelder stehen, befindet. Dazu vergleichen Sie das Elternfenster des gerade ermittelten Fensters mit dem Elternfenster der eigenen Klasse. Beide müssen gleich sein, wenn der Benutzer die Maustaste über dem Dialog losgelassen hat. Um zu verhindern, daß die Drag&Drop-Funktion auf dem eigenen Listenfeld oder dem zweiten Listenfeld mit dem Ausgangstext wirkt, überprüfen Sie mit der Methode *IsKindOf* von *CObject*, welche Basisklasse das ermittelte Fenster hat. Erst jetzt wird mit *GetText* die Zeile aus dem ersten Listenfeld ausgelesen und mit *AddString* in das Ziellistenfeld übertragen. Zum Schluß müssen Sie nur noch den alten Cursor wieder herstellen. Diese lange Beschreibung sieht im Programmcode wesentlich einfacher aus.

```
void CDragListBox::OnLButtonUp(UINT nFlags, CPoint point)
{
    CString strKey;
    CListBox* pLB;
    if (GetCapture())
    {
        ClientToScreen(&point);
        pLB = (CListBox*)WindowFromPoint(point);
        if (!pLB->IsKindOf(RUNTIME_CLASS(CDragListBox)) &&
            (GetParent()==pLB->GetParent()))
        {
            GetText(GetCurSel(), strKey);
            pLB->AddString(strKey);
        }
        SetCursor(hcOld);
    }
    CListBox::OnLButtonUp(nFlags, point);
}
```

Listing: Übergabe einer Listenfeldzeile beim Loslassen der Maus

Mit dieser eigenen Klasse sind Sie nun in der Lage, die Drag&Drop-Funktion anzuwenden. Im Beispiel sind zwei unterschiedliche Möglichkeiten zum Verbinden der Steuerungen mit der Klasse verwendet worden. Einmal mit Hilfe des Klassenassistenten und über die Methode *DoDataExchange* von *CWnd*.


```

void CDlgFileListBox::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CDlgFileListBox)
    DDX_Control(pDX, IDC_LIST3, m_lbAusgabe);
    DDX_Control(pDX, IDC_LIST2, m_lbDragFile2);
    //}}AFX_DATA_MAP
}

```

Listing: Verbinden der eigenen Klasse mit dem Listenfeld

Die zweite Möglichkeit ergibt sich über die Methode *SubclassDlgItem* von *CWnd*. Damit können Sie dynamisch eine Klasse mit der Vorlage verbinden.

```
m_lbDragFile1.SubclassDlgItem(IDC_LIST1, this);
```

Die Daten der Listenfelder werden aus einer Textdatei eingelesen. Für das Lesen und Schreiben in Textdateien gibt es eine eigene, von *CFile* abgeleitete Klasse in der MFC, die spezielle Methoden zur Verfügung stellt. Die bereits erwähnte Klasse *CStdioFile* stellt die Methode *ReadString* zum zeilenweisen Lesen und *WriteString* zum Schreiben in eine Textdatei zur Verfügung.

```

virtual LPTSTR ReadString(LPTSTR lpsz, UINT nMax);
    throw(CFileException);
BOOL ReadString(CString& rString);
    throw(CFileException);
virtual void WriteString(LPCTSTR lpsz);
    throw(CFileException);

```

Beide Methoden unterstützen die oben bei *CFile* beschriebene Ausnahmebehandlung. Als Parameter erwarten die Methoden entweder ein *CString*-Objekt oder, wie auch bei *CFile*, einen Zeiger auf einen Puffer mit entsprechender Längenangabe. Die Methode *ReadString* liefert FALSE zurück, wenn das Dateiende erreicht ist. Zur Bestimmung des Dateinamens der zu öffnenden Textdatei verwenden Sie den bereits bei »Einfache Dialoge« beschriebenen Dialog *Datei öffnen*. Aufgerufen werden die Funktionen über eine Schaltfläche im Dialog der Beispielanwendung.

Die Datei wird mit den Optionen *CFile::modeRead* und *CFile::typeText* geöffnet. Nach dem erfolgreichen Öffnen der Textdatei wird zeilenweise mit *ReadString* gelesen und sofort *AddString* an das Listenfeld angehängt, bis das Ende der Textdatei erreicht ist. Vorher wird mit der Methode *ResetContent* von *CListBox* der Inhalt des Listenfeldes gelöscht. Dadurch können Sie mehrfach eine Datei einlesen. Vergessen Sie nicht, die Datei mit *Close* wieder zu schließen.

```

void CDlgFileListbox::OnDateiLesen1()
{
    CString strInput;
    CFileDialog dlgOpen(TRUE, "txt", "Input1.txt",
                        OFN_HIDEREADONLY,
                        "Textdateien (*.txt) |*.txt||",
                        this);
    if(dlgOpen.DoModal()==IDOK)
    {
        if (!m_fText.Open(dlgOpen.GetPathName(),
                        CFile::modeRead | CFile::typeText))
        {
            AfxMessageBox("Fehler beim Öffnen der Datei!");
            return;
        }
        m_lbDragFile1.ResetContent();
        while (m_fText.ReadString(strInput))
            m_lbDragFile1.AddString(strInput);
        m_fText.Close();
    }
}

```

Listing: Zeilenweises Lesen in einer Textdatei.

Dies Funktion implementieren Sie zweimal, denn das Füllen des zweiten Listenfeldes erfolgt analog, nur daß Sie den Text nach `m_lbDragFile2` übernehmen.

Das Schreiben in eine Textdatei ist ähnlich einfach. Die Klasse *CFileDialog* dient auch hier zur Bestimmung des Dateinamens. Für den Modus *Datei speichern unter* wird als erster Parameter ein `FALSE` übergeben. Beim Öffnen der Datei werden jetzt die Optionen *CFile::modeWrite*, *CFile::typeText* und *CFile::modeCreate* benutzt. Dadurch wird jedesmal eine neue Datei angelegt. Zum Schreiben werden alle Elemente aus dem Listinfeld ausgelesen und mit *WriteString* in die offene Datei geschrieben. Vorher muß jedoch an jede Zeile das Zeilenende-Kennzeichen (`\n`) angefügt werden.

```

void CDlgFileListbox::OnOK()
{
    CString strOutput;
    CFileDialog dlgOpen(FALSE, "txt", "Output.txt",
                        OFN_HIDEREADONLY,
                        "Textdateien (*.txt) |*.txt||",
                        this);
    if(dlgOpen.DoModal()==IDOK)
    {

```

```

    if (!m_fText.Open(dlgOpen.GetPathName(),
        CFile::modeWrite |
        CFile::typeText |
        CFile::modeCreate))
    {
        AfxMessageBox("Fehler beim Öffnen der Datei!");
        return;
    }
    for (int i = 0; i < m_lbAusgabe.GetCount(); i++)
    {
        m_lbAusgabe.GetText(i, strOutput);
        strOutput += '\n';
        m_fText.WriteString(strOutput);
    }
    m_fText.Close();
}
CDialog::OnOK();
}

```

Listing: Zeilenweises Schreiben in eine Textdatei

Der Rest ist noch etwas Kosmetik. Um die Anzeige mit Hilfe von Tabulatoren in dem Listenfeld zu verbessern, weisen Sie den Listenfeldern neue Tabulatorabstände zu. Dazu gib es die Methode *SetTabStops* von *CListBox*. Damit läßt sich in dem Text ein tabellenartiges Aussehen realisieren. Das funktioniert natürlich auch mit der abgeleiteten Klasse. Das Ganze wird in der Funktion *OnInitDialog* durchgeführt.

```

BOOL CDlgFileListbox::OnInitDialog()
{
    CDialog::OnInitDialog();
    // auch eine Möglichkeit eine Klasse
    // mit einer Ressource zu verbinden
    m_lbDragFile1.SubclassDlgItem(IDC_LIST1, this);
    // TabStop für das Input-Listenfeld
    for (int i = 1; i < 13; i++)
    {
        tabst[i-1] = 15 * i;
    }
    m_lbDragFile2.SetTabStops(12, (LPINT)tabst);
    // TabStop für das Output-Listenfeld
    for (i = 1; i < 13; i++)
    {
        tabst[i-1] = 12 * i;
    }
}

```

```

m_lbAusgabe.SetTabStops(12, (LPINT)tabst);
return TRUE;
}

```

Listing: Setzen der Tabulatorabstände für die Listenfelder

Damit können Sie sogar mit einfachen Zeichen kleine Grafiken in einem Listenfeld darstellen. Doch das ist nur ein Nebeneffekt. Ihr fertiger Dialog sollte dann etwa wie Abbildung 11.8 aussehen.

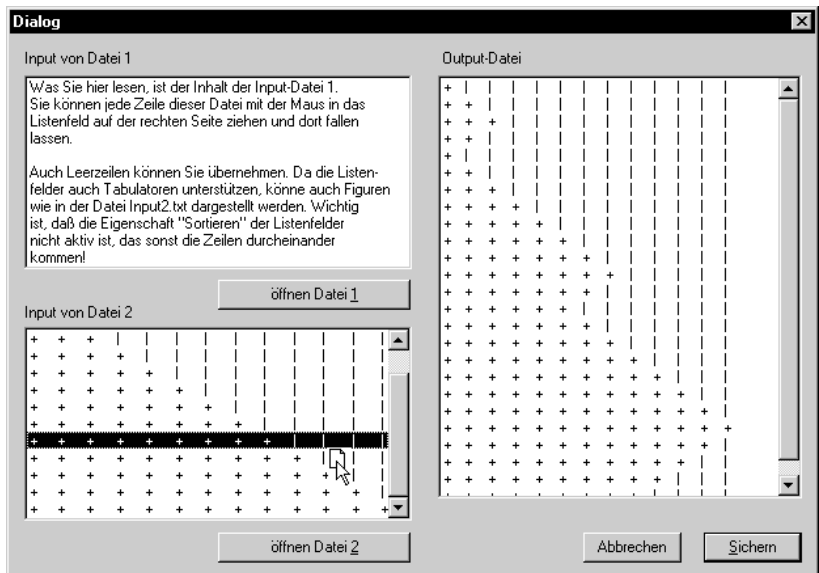


Abbildung 11.8: Der Beispiel-Dialog

Dieses Beispiel soll Ihnen die Möglichkeiten der Klasse *CFile* zeigen. Es ist an Ihnen, das Programm auszubauen und zu testen. Es befindet sich auf der beiliegenden CD. Als erstes sollten Sie eine Ausnahmebehandlung bei den entsprechenden Methoden implementieren und dann das Verhalten in Ausnahmesituationen testen.



Die drei Formen der
MFC-Anwendung

TEIL IV



Dialogfelder und die Klasse *CDialog*

12

Kapitelübersicht

12.1 Grundlagen der Programmierung	440
12.2 Dialogvorlagen mit dem Dialogeditor	442
12.2.1 Bedienelemente des Dialogeditors	442
12.2.2 Festlegen der Tabulator-Reihenfolge	450
12.2.3 Test der Dialogvorlage	452
12.3 Dialoge ohne eigene Klasse	452
12.4 Ableiten eigener Dialogfeldklassen	455
12.4.1 Eigenschaften eigener Dialogfeldklassen	455
12.4.2 Dialogfeld-Klasse mit dem Klassen-Assistenten	459
12.5 Dialogbasierende Anwendung - ein Beispiel	462
12.5.1 Vorteile und Besonderheiten	462
12.5.2 Aufgabe	466
12.5.3 Bedienung	467
12.5.4 Implementierung	468
12.5.5 Projektanlage mit dem MFC-Anwendungs-Assistenten	468
12.5.6 Gestaltung der Dialogvorlage	470
12.5.7 Zusätzlicher Programmcode	471
12.6 Sonstiges	483
12.7 Nicht-modale Dialoge	484
12.7.1 Unterschiede zu modalen Dialogfeldern	484
12.7.2 Entwurf der Dialogvorlage	485
12.7.3 Programmierung	486
12.7.4 Anwendungsbeispiele	488

12.1 Grundlagen der Programmierung

Die bisherigen Kapitel haben Ihnen gezeigt, wie Sie mit Hilfe der Entwicklungsumgebung einfache Programme erstellen, Textinformationen auf den Bildschirm bringen und das Programm mit einem Menüsystem ausstatten können. Diese Fähigkeiten reichen natürlich noch nicht aus, um mit dem Benutzer in einen sinnvollen Dialog zu treten, denn dazu ist es erforderlich, das Dialogfeldkonzept von Windows-Programmen zu verstehen und demgemäß programmieren zu können.

Dialogfelder sind verschiebbare Fenster von meist fester Größe, die über eine Reihe interaktiver Steuerelemente verfügen. In der Regel aktiviert das Programm ein Dialogfeld auf Anforderung des Benutzers, erlaubt das lokale Ändern der Dialogdaten mit Hilfe der angezeigten Steuerelemente und wertet die Ergebnisse aus, nachdem das Dialogfeld durch den Benutzer geschlossen wurde. Dialogfelder dienen also dazu, Daten und Informationen in einer definierten Form anzuzeigen oder diese vom Anwender zu erhalten. Der Benutzer verwendet Dialogfelder, um in einen Austausch mit Ihrem Programm zu treten.

Diese Art der Interaktion mit dem Bediener ist nicht zwangsläufig. Komplexere Windows-Programme besitzen oft nur wenige Dialogfelder und erledigen die Kommunikation mit dem Anwender über ein grafisch anspruchsvoll gestaltetes Hauptfenster mit einer Vielzahl aktiver Bildschirmflächen, die oft mechanischen Steuerelementen nachgebildet wurden. Ein solches Programm ist nicht an die vordefinierten Steuerelemente gebunden, die bei der Dialogfeldprogrammierung zur Verfügung stehen, sondern kann sich den Erfordernissen einer speziellen Anwendungssituation viel besser anpassen.

Leider ist die Programmierung dieser Schnittstellen sehr aufwendig und darüber hinaus nicht standardisiert. Als angehender Windows-Programmierer sollten Sie sich daher zunächst auf die reine Dialogfeldprogrammierung beschränken und die damit verbundenen Möglichkeiten ausschöpfen. Die folgenden Kapitel versuchen deshalb, das Erstellen und Einbinden von Dialogfeldern in MFC-Programme zu erklären. Besonderer Wert wird dabei auf die Darstellung der verschiedenen Steuerelemente gelegt, die durch spezielle Klassen und Nachrichtenbehandlungsroutinen wesentlich einfacher zu programmieren sind als in herkömmlichen SDK-Programmen.

In Programmen, die auf SDK-Programmierung beruhen, wird ein Dialogfeld mit der Funktion *DialogBox* auf den Bildschirm gebracht. Der wichtigste Parameter ist dabei ein Zeiger auf eine Dialogfunktion, in der alle spezifischen Nachrichten verarbeitet werden. Ähnlich der Klasse *CWnd*, die die konventionelle Bearbeitung der Hauptfenster-Nachrichten ersetzt, gibt es in den MFC die Klasse *CDialog*, mit der die Nachrichtenbearbei-

tung für Dialogfelder sehr vereinfacht werden kann. *CDialog* betreibt dabei die Schleife für die Dialognachrichten und verfügt über eine eigene Dialogfunktion.

CDialog erlaubt es Ihnen, sowohl *modale* als auch *nicht-modale* Dialogfelder zu erzeugen. Während Sie ein sehr einfaches modales Dialogfeld direkt durch die Klasse *CDialog* erzeugen können, müssen Sie für komplexere oder nicht-modale Dialogfelder auf jeden Fall eigene Klassen ableiten. Die beiden Dialogarten unterscheiden sich dabei vor allem durch die Art, wie sie ihre Dialogfelder aufrufen und beenden. Die Behandlung des laufenden Dialogfeldes erfolgt dagegen weitgehend übereinstimmend. Ist ein Dialogfeld modal, so muß der Dialog erst beendet werden, bevor in der zugehörigen Anwendung mit einem anderen Programmteil fortgefahren werden kann. Alle Versuche, Teile darunterliegender Fenster zu aktivieren, lösen lediglich das akustische Signal für »Fehler« aus. Ist das Dialogfeld nicht-modal, so laufen Anwendung und Dialogfeld parallel, und es ist möglich, in mehreren Fenstern abwechselnd zu arbeiten.

Beispiele für diese unterschiedlichen Dialogtypen lassen sich leicht aufzählen: die Funktion *Suchen und Ersetzen* in Textverarbeitungen ist meist ein nicht-modales Dialogfeld, dagegen ist das *Info über ...*-Dialogfeld vieler Windows-Programme modal.



Da modale Dialogfelder in der Praxis wesentlich wichtiger sind als nicht-modale, beschäftigt sich dieser Abschnitt ausschließlich mit modalen Dialogfeldern. Das Erzeugen nicht-modaler Dialogfelder wird dagegen in einem späteren Abschnitt behandelt.

Voraussetzung für die Darstellung eines Dialogfeldes auf dem Bildschirm ist eine dazugehörige Dialogressource. Um eine entsprechende Dialogvorlage bereitzustellen, stehen Ihnen drei Möglichkeiten zur Verfügung.

Erstens: Die Verwendung eines Templates, das es Ihnen ermöglicht, die Ressource quasi zur Laufzeit des Programmes zu erzeugen. Den Methoden *InitModalIndirect* und *CreateIndirect* der Klasse *CDialog* wird dann ein Zeiger auf eine Templatestruktur vom Typ *DLGTEMPLATE* übergeben. Nach den darin enthaltenen Vorgaben für die Eigenschaften der Dialogvorlage wird dann ein modaler bzw. nicht-modaler Dialog erzeugt. Diese Methode ist nicht sehr verbreitet und wird auch hier nicht weiter beschrieben.

Zweitens: Durch das Bereitstellen einer Dialogfeldvorlage in einer Resource-Script-Datei (RC-File), welches Sie per Hand erstellen und in Ihr Projekt einbinden müssen. Mit einer speziellen Resource-Script-Sprache werden in dieser Datei u.a. die Informationen für Dialogvorlagen abgelegt. Mit dem Resource-Compiler wird das Script in eine für das Betriebssystem lesbare Form gebracht und durch den Linker an das fertige Programm gebunden. Ein funktionstüchtige Dialogvorlage könnte wie folgt aussehen:



```
IDD_MY_DIALOG DIALOGEX 0, 0, 185, 95
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Mein Dialog"
{
    DEFPUSHBUTTON "OK", IDOK, 130, 70, 50, 14
    PUSHBUTTON "Abbrechen", IDCANCEL, 70,70,50,14
    LTEXT "Das ist ein Beispieldialog", IDC_STATIC, 50, 30, 80, 8
}
```

Es können hier alle Eigenschaften des Dialogfeldes in einer lesbaren Form beschrieben werden. Alle Informationen über Position, Größe und Form des Dialogfeldes und der eingebetteten Steuerelemente werden hier definiert. Doch die Syntax soll an dieser Stelle nicht weiter beschrieben werden, denn Sie besitzen mit der Entwicklungsumgebung ein spezielles Werkzeug für die Erstellung selbst komplexer Dialogvorlagen. Diese Technik wird hier nur erwähnt, da es die traditionelle Methode für die Erstellung von Dialogfeldern ist bzw. war.

Wenn Sie mit dem Microsoft Developer Studio arbeiten, ist die Kenntnis dieser Methode für Sie nur als Grundlagenwissen interessant, denn das Erzeugen des Ressource Scripts übernimmt in diesem Fall der Dialogeditor für Sie. Womit wir bereits bei der dritten Möglichkeit sind: Der integrierte Dialogeditor ermöglicht Ihnen einen komfortablen Entwurf und einen Test der Dialogfelder. Da er eine sehr umfangreiche Funktionalität besitzt, ist ihm das Kapitel 12.2 gewidmet.

In diesem Kapitel wird an einem ausführlichen Beispiel sowohl die Verwendung der bereits beschriebenen Steuerungen und deren Klassen in den MFC als auch das Erstellen einer Dialogvorlage und der dazugehörigen Klasse *CDialog* erläutert. Zudem werden einige weitere Anwendungsmöglichkeiten der Dialogklasse der MFC vorgestellt.

12.2 Dialogvorlagen mit dem Dialogeditor

12.2.1 Bedienelemente des Dialogeditors

Den Dialogeditor starten Sie, indem Sie entweder eine vorhandene Dialogvorlage öffnen oder im Menü EINFÜGEN den Befehl RESSOURCE sowie anschließend den Ressourcentyp *Dialog* markieren und danach auf NEU klicken. Außerdem ist es möglich, durch Drücken der rechten Maustaste auf den Ordner DIALOG im Arbeitsbereichsfenster *Ressourcen* ein Menü zu aktivieren, um einen neuen Dialog einzufügen.

Das Dialogeditorfenster

Dieses Fenster der Entwicklungsumgebung zeigt eine Dialogvorlage in einem Entwurfs- und Bearbeitungsmodus an. Hier können Sie Dialogfelder erstellen, Steuerelemente einfügen und positionieren sowie fertige Dialog-

felder testen. Des weiteren können Sie Dialogvorlagen für zukünftige Dialogfelder speichern. Konstruktionslinien und ein Raster vereinfachen das Ausrichten der Steuerelemente.

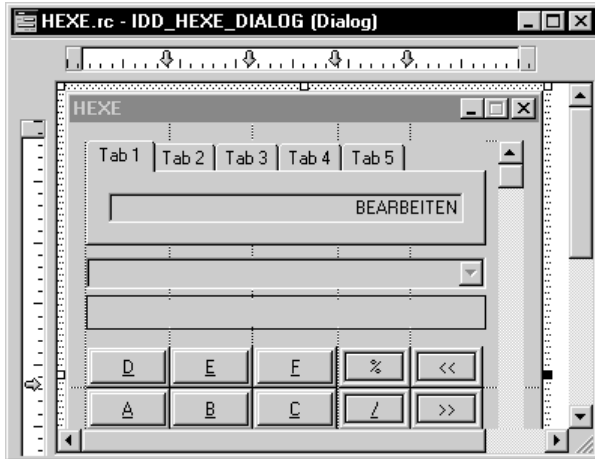


Abbildung 12.1: Das Dialogeditorfenster

In diesem Fenster steht oben und links je ein Lineal zur Verfügung, mit dem die blau gestrichelten Führungslinien positioniert werden können. Diese erleichtern Ihnen die Ausrichtung und Anordnung der Steuerungen.

Die Steuerelementtypen

Das eigentliche Anlegen der Steuerungen innerhalb der Dialogvorlage geschieht mit der Steuerelemente-Symbolleiste. Dazu wählen Sie das gewünschte Steuerelement aus und ziehen es mit der Maus über die gewünschte Position innerhalb des Dialogfeldes.



Abbildung 12.2: Steuerelement-Symbolleiste

Eine weitere Möglichkeit, einem Dialogfeld Steuerelemente hinzuzufügen, erfolgt dadurch, daß Sie vorhandene Steuerelemente neu positionieren oder Steuerelemente von anderen Dialogfeldern durch Verschieben bzw. mit KOPIEREN und EINFÜGEN über die Zwischenablage vervielfältigen.

Die Steuerelementtypen der Symbolleiste entsprechen den Standardsteuerungen, die bereits im Kapitel 10 erläutert wurden. In Abbildung 12.2 sind das von links oben nach rechts unten:

- ▼ Auswählen (aktiver Mauszeiger)
- ▼ Bild
- ▼ Textfeld
- ▼ Eingabefeld
- ▼ Gruppenfeld
- ▼ Schaltfläche
- ▼ Kontrollkästchen
- ▼ Optionsfeld
- ▼ Kombinationsfeld
- ▼ Listenfeld
- ▼ Vertikale und
- ▼ Horizontale Bildlaufleiste
- ▼ Drehfeld
- ▼ Statusanzeige
- ▼ Regler
- ▼ Zugriffstaste
- ▼ Liste
- ▼ Strukturansicht
- ▼ Registerkarte
- ▼ Animation
- ▼ Rich Edit Steuerelement
- ▼ Datums- /Zeitauswahl
- ▼ Monatskalender
- ▼ IP-Adresse
- ▼ Benutzerdefiniertes Steuerelement und
- ▼ Erweitertes Kombinationsfeld.

Die Formatierung von Steuerungen

Für die Formatierung der eingefügten Steuerungen steht eine Symbolleiste zur Verfügung. Mit deren Hilfe können Steuerelemente automatisch ausgerichtet und deren Größe angepaßt werden. Die entsprechenden Befehle sind auch über das *Layout*-Menü erreichbar



Abbildung 12.3: Dialogfeld-Symbolleiste

Diese Symbolleiste teilt sich in sechs Bereiche. Ganz links ist der Schalter zum Aktivieren des Testmodus' für die Dialogvorlage angeordnet. Rechts außen befinden sich die Schalter zum Aktivieren bzw. Deaktivieren der Raster- und Führungslinien. Um die weiteren Funktionen ausführen zu können, müssen die zu bearbeitenden Steuerungen markiert werden.

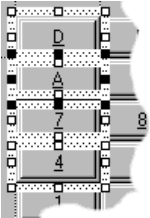


Abbildung 12.4: Markierung von Steuerungen

Hierbei spielt die Reihenfolge der Markierung eine wichtige Rolle, denn sie bestimmt, welches das *dominante Element* ist. Bei mehreren markierten Steuerungen ist das dominante Element an den blauen bzw. schwarzen Begrenzungskästchen zu erkennen, während alle anderen Steuerungen durchsichtige Kästchen haben. Dominant ist jeweils das zuletzt markierte Element; dies kann aber geändert werden, indem eine Steuerung bei gedrückter `[Strg]`-Taste erneut markiert wird.

Die Gruppe der vier linken Schalter der Dialogfeld-Symbolleiste ermöglicht das Ausrichten der ausgewählten Elemente nach links, rechts, oben oder unten. Die nächsten beiden Schalter dienen dem vertikalen bzw. horizontalen Zentrieren der Steuerelemente innerhalb des Dialogfeldes. Die folgenden beiden Schalter ermöglichen ein Optimieren der Abstände zwischen den markierten Elementen nach oben und unten oder nach links und rechts. Die letzten drei Schalter lösen die Größenanpassung der ausgewählten Steuerungen aus. Nach dem Aufruf der gewünschten Funktion werden die entsprechenden Veränderungen an den ausgewählten bzw.

nicht-dominanten Steuerungen vorgenommen, während die dominante bzw. die nicht-markierten Steuerungen unverändert bleiben.

Die Eigenschaften der Dialogvorlage

Wie bei den einzelnen Steuerungen, können auch bei Dialogvorlagen verschiedene Eigenschaften und Attribute für das spätere Aussehen und das Verhalten des Dialoges bereits in der Entwurfsansicht festgelegt werden. Um das Eigenschaftsfenster zu aktivieren, stehen drei Möglichkeiten zur Verfügung; diese gelten sowohl für die gesamte Dialogvorlage als auch für die eingefügten Steuerungen, die beschriebenen Aktionen gelten jeweils für das aktuell markierte Objekt.

1. durch Betätigen der Tastenkombination **Alt** + **↵** (wie bei Windows allgemein üblich);
2. über das Hauptmenü mit **ANSICHT|EIGENSCHAFTEN**;
3. über das Menü der rechten Maustaste.



Abbildung 12.5: Menü im Dialogeditorfenster »Rechte Maustaste«

Das *Eigenschaften*-Dialogfeld hat vier Seiten. Einige der Eigenschaften und Attribute sind Ihnen bereits Kapitel 10 bekannt und haben hier die gleiche Wirkung bzw. Funktion.



Abbildung 12.6: Dialogfeld »Eigenschaften«, Allgemein

Dialogfelder und die Klasse CDialog

Die erste Seite zeigt die allgemeinen Eigenschaften des Dialogfelds. Sie werden in der folgenden Tabelle beschrieben:

Bezeichnung	Bedeutung
ID	Symbolname vom Typ Integer oder String, der in der Schnittstellendatei definiert wird.
Beschriftung	Text, mit dem das Dialogfeld beschriftet wird. Erscheint in der Titelleiste des Dialogs.
Schriftname	Schriftart, die für die Steuerelemente des Dialogfelds verwendet wird; Wert kann mit der Schaltfläche SCHRIFTART in der linken unteren Ecke der Eigenschaftenseite geändert werden.
Menü	Enthält den Symbolnamen des im Dialogfeld verwendeten Menüs, sofern vorhanden.
Schriftgrad	Schrifthöhe in Punkt, die für die Steuerelemente des Dialogfelds verwendet wird; der Wert kann ebenfalls mit der Schaltfläche SCHRIFTART geändert werden.
Schriftart...	Mit dieser Schaltfläche lassen sich Art und Größe der Schrift verändern
X-Pos	X-Koordinate der linken oberen Ecke des Dialogfelds in <i>Dialogfeldeinheiten</i> (dialogbox units).
Y-Pos	Y-Koordinate der linken oberen Ecke des Dialogfelds in <i>Dialogfeldeinheiten</i> (dialogbox units)
Klassenname	Bezeichner für eine registrierte Dialogfeldklasse des Betriebssystems; bei MFC-Unterstützung nicht aktiv.

Tabelle 12.1: Die allgemeinen Eigenschaften des Dialogfelds

Auf der zweiten Seite für die Eigenschaften des Dialogfelds werden die Format-Eigenschaften angezeigt und geändert.



Abbildung 12.7: Dialogfeld »Eigenschaften«, Formate

Im einzelnen haben die Felder folgende Bedeutung:

Bezeichnung	Bedeutung
Format	Kombinationsfeld mit Fensterarten: überlappend, Einblendfenster (Kontextmenü) und untergeordnetes Fenster.
Rand	Kombinationsfeld mit Fensterrandtypen: kein Rand, dünner Rand, änderbare Größe des Dialogfeldes und Dialogfeldrahmen.
Titelleiste	Titelleiste für Überschrift und Systemmenü des Dialogfeldes; ist inaktiviert, wenn das Dialogfeld keinen Rand hat.

Tabelle 12.2: Die Format-Eigenschaften

Bezeichnung	Bedeutung
Systemmenü	Systemmenü für das Dialogfeld. Dieses Kontrollkästchen ist deaktiviert, wenn keine Titelleiste vorhanden ist.
Minimieren-Schaltfläche	bewirkt Minimieren-Schaltfläche in der Titelleiste des Dialogfelds. Nur aktiv, wenn Titelleiste vorhanden.
Maximieren-Schaltfläche	bewirkt Maximieren-Schaltfläche in der Titelleiste des Dialogfelds. Nur aktiv, wenn Titelleiste vorhanden.
Nebengeordnete ausschneiden	kann nur bei untergeordnetem Fenstertyp (<i>Format</i>) verwendet werden; beim Neuzeichnen eines einzelnen Fensters bewirkt diese Eigenschaft, daß alle anderen untergeordneten Fenster der obersten Ebene aus dem Bereich des zu aktualisierenden Fensters ausgeschnitten werden.
Untergeordnete ausschneiden	kann bei Erstellung eines übergeordneten Fenstertyps verwendet werden. Schließt beim Zeichnen innerhalb des übergeordneten Fensters den Bereich aus, der durch untergeordnete Fenster belegt wird; Eigenschaft nicht verwenden, wenn das Dialogfeld ein Gruppenfeld enthält.
Horizontaler Bildlauf	erzeugt eine horizontale Bildlaufleiste für das Dialogfeld.
Vertikaler Bildlauf	erzeugt eine vertikale Bildlaufleiste für das Dialogfeld.

Tabelle 12.2: Die Format-Eigenschaften



Wenn Bildlaufleisten für ein Dialogfeld verwendet werden, das einen Rand vom Typ *Dialogfeldrahmen* verwendet, überlappen die Bildlaufleisten in der Darstellung den Rand des Dialogfelds, anstatt innerhalb des Randes zu liegen. Auch wird der Inhalt des Dialogfelds nicht richtig ausgeschnitten. Dieses Verhalten ist durch Windows bedingt. Um es zu vermeiden, muß bei der Erstellung eines Dialogfelds mit Bildlauf ein anderes Randformat verwendet werden.

Die dritte Seite der Eigenschaften eines Dialogfelds enthält weitere Format-Eigenschaften.

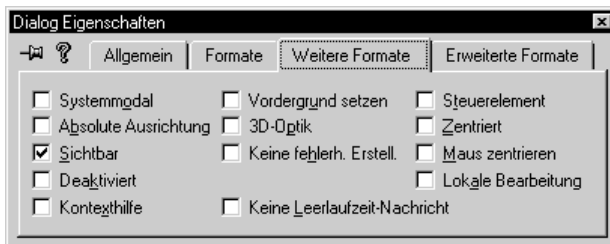


Abbildung 12.8: Dialogfeld »Eigenschaften«, weitere Formate

Die Bedeutung der einzelnen Felder sind in Tabelle 12.3 beschrieben:

Dialogfelder und die Klasse CDialog

Bezeichnung	Bedeutung
Systemmodal	Eigenschaft eines Dialogfelds, das ein Umschalten zu einem anderen Fenster oder Programm verhindert.
Absolute Ausrichtung	legt die Ausrichtung des Dialogfeld fest; relativ zum Bildschirm oder relativ zum Eltern-Fenster.
Sichtbar	gibt an, ob das Dialogfeld beim ersten Anzeigen sichtbar ist oder nicht.
Deaktiviert	legt fest, ob das Dialogfeld zu Beginn aktiviert ist oder nicht.
Kontexthilfe	fügt in die Titelleiste des Dialogfelds eine Fragezeichen-Schaltfläche ein.
Vordergrund festlegen	Windows bringt das Dialogfeld mit der Funktion <i>SetForegroundWindow</i> in den Vordergrund.
3D-Ansicht	das Dialogfeld erhält eine magere Schriftart und die Ränder der Steuerelemente erscheinen dreidimensional.
Keine Erstellungsfehler	das Dialogfeld wird auch erzeugt, wenn bei der Erstellung Fehler aufgetreten sind.
Keine Leerlauf-Nachricht	verhindert das Senden der WM_ENTERIDLE-Nachricht an das Eltern-Fenster.
Steuerelement	es wird ein Dialogfeld erzeugt, welches als untergeordnetes Fenster eines anderen Dialogfelds arbeitet, analog einer Seite in einem Eigenschaftsblatt; diese Eigenschaft ermöglicht es, mit der TAB -TASTE zwischen den Steuerelementfenstern eines untergeordneten Dialogfelds zu wechseln.
Zentriert	das Dialogfeld wird im Arbeitsbereich zentriert.
Maus zentrieren	der Mauszeiger wird im Dialogfeld zentriert.
Lokale Bearbeitung	Eingabefeld-Steuerelemente des Dialogfelds benutzen den Speicher im Datensegment der Anwendung.

Tabelle 12.3: Weitere Format-Eigenschaften

Die vierte und letzte Seite enthält erweiterte Formate als Eigenschaften für das Dialogfeld.

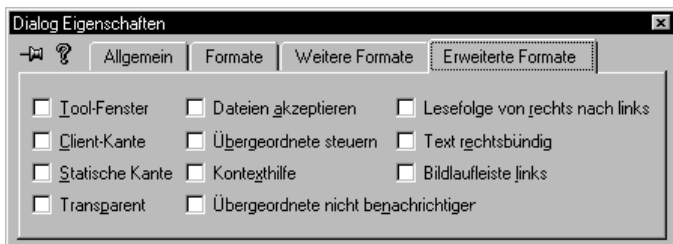


Abbildung 12.9: Dialogfeld »Eigenschaften«, erweiterte Formate

Bezeichnung	Bedeutung
Tool-Fenster	es wird ein Fenster mit kleiner Titelleiste erzeugt; es kann z.B. als Symbolleiste oder Eigenschaftsfenster dienen.
Client-Kante	der Rand des Dialogfelds erhält eine vertiefte Kante.
Statische Kante	erzeugt einen zusätzlichen, kaum sichtbaren Rand um das Dialogfeld.
Transparent	Fenster, die unter einem Dialogfeld mit dieser Eigenschaft liegen, werden von ihm nicht überdeckt.
Dateien akzeptieren	ermöglicht dem Empfang von Drag-and-Drop-Dateien; es werden WM_DROPFILES-Nachrichten an das betroffene Steuerelement gesendet.
Übergeordnete steuern	Ermöglicht es, mit der <code>[TAB]</code> -TASTE zwischen den untergeordneten Fenstern des Dialogfelds zu wechseln.
Kontexthilfe	fügt in die Titelleiste des Dialogfelds eine Fragezeichen-Schaltfläche ein.
Übergeordnete nicht benachrichtigen	das untergeordnete Fenster sendet keine WM_PARENTNOTIFY-Nachricht an das übergeordnete Fenster.
Lesefolge von rechts nach links	für Sprachen wie Hebräisch oder Arabisch; Text wird von rechts nach links geschrieben.
Text rechtsbündig	Text wird innerhalb des Dialogfelds rechtsbündig ausgerichtet.
Bildlaufleiste links	ist eine vertikale Bildlaufleiste vorhanden, wird sie links vom Client-Bereich dargestellt.

Tabelle 12.4: Erweiterte Format-Eigenschaften



Kontexthilfe ist sowohl auf der dritten als auch auf der vierten Seite des *Eigenschaftens*-Dialogs vorhanden. Die Funktionalität ist nach außen hin identisch. Wenn der Benutzer auf das Fragezeichen klickt, ändert sich die Cursordarstellung in ein Fragezeichen mit einem Pfeil. Klickt der Benutzer dann auf ein Steuerelement, empfängt dieses eine WM_HELP-Nachricht. Der Unterschied besteht in den Styles der Dialogvorlage im Ressourcenscript. Durch den Schalter auf der dritten Seite (*Weitere Formate*) wird `STYLE DS_CONTEXTHELP` gesetzt. Dagegen wird auf der vierten Seite `EXSTYLE WS_EX_CONTEXTHELP` verwendet.

Das *Dialogfeld-Eigenschaften*-Fenster bleibt sichtbar, wenn Sie den Knopf in der linken oberen Ecke betätigen. Diese Einstellung empfiehlt sich, wenn Sie während einer Bearbeitungssitzung häufig zwischen dem Einstellen von Eigenschaften und dem Bearbeiten von Objekten wechseln.

12.2.2 Festlegen der Tabulator-Reihenfolge

Die Tabulator-Reihenfolge bestimmt die Abfolge, in der die Steuerelemente innerhalb der Dialogvorlage durchlaufen werden und den Fokus erhalten, wenn der Benutzer wiederholt die `[TAB]`-Taste drückt. Darüber hinaus bekommt das allererste Steuerelement den Eingabefokus, wenn der Dialog aufgerufen wird. Auch für das Definieren von Zugriffstasten und das Verhalten von Optionsfeldern in einer Gruppe hat die Tabulator-

Reihenfolge eine Bedeutung. Außerdem hat die Tabulator-Reihenfolge in Ihrem Dialogfeld für überlappende Steuerelemente eine Auswirkung. Die Anzeige der Steuerelemente kann sich ändern, wenn Sie die Tabulator-Reihenfolge ändern. Es werden nämlich die Steuerelemente, die in der Tabulator-Reihenfolge zuerst aufgeführt sind, immer über den überlappenden Steuerelementen angezeigt, die in der Tabulator-Reihenfolge später kommen.

Um die Reihenfolge zu ändern, kann der Menüpunkt **LAYOUT|TABULATOR-REIHENFOLGE** verwendet oder die Tastenkombination **[Strg] [D]** gedrückt werden. In diesem Modus sind alle Steuerungen mit einer Nummer versehen, welche ihre Position innerhalb der Tabulator-Reihenfolge anzeigt. Das Drücken einer beliebigen Taste schaltet wieder auf die Normaldarstellung zurück. Um die Steuerelemente in einer bestimmten Weise anzuordnen, müssen sie in der gewünschten Reihenfolge angeklickt werden. Dabei bekommen sie fortlaufende Nummern – beginnend mit 1 bis zur Anzahl der Steuerelemente – zugewiesen. Werden nicht alle Steuerelemente angeklickt, so bleibt die Reihenfolge derjenigen, die nicht berücksichtigt wurden unverändert.

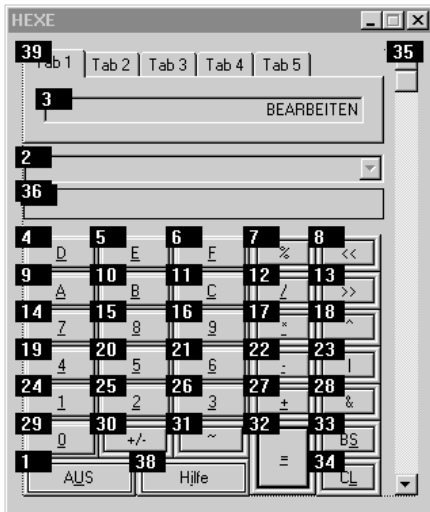


Abbildung 12.10: Tabulatorreihenfolge im Dialogeditor

Um die Reihenfolge für mehrere Steuerelemente zu ändern, legen Sie das erste zu verändernde Steuerelement fest, indem Sie das Steuerelement vor diesem Steuerelement durch gleichzeitiges Drücken der **[Strg]**-Taste markieren. Das markierte Steuerelement bestimmt nun die Nummer des Steuerelements, auf das Sie anschließend klicken.

12.2.3 Test der Dialogvorlage

Sie können die Funktionalität eines Dialogfelds im Dialog-Editor testen, ohne Ihr Programm zu kompilieren. Dadurch können Sie sofort prüfen, wie die Steuerelemente angezeigt und ausgeführt werden, wodurch das Erstellen der Benutzeroberfläche wesentlich vereinfacht und beschleunigt wird.

Den Testmodus können Sie entweder über das Menü LAYOUT|TESTEN, über die Tastatur mit **[Strg] [G]** oder über die Dialogfeldsymbolleiste starten. Zum Beenden drücken Sie die **[Esc]**-Taste oder die Schaltfläche mit dem Symbolnamen IDOK oder IDCANCEL.

Im Testmodus können Sie:

- ▼ Text eingeben
- ▼ Elemente in Kombinationsfeldlisten markieren
- ▼ Optionen aktivieren und deaktivieren
- ▼ Befehle wählen
- ▼ die Tabulator-Reihenfolge überprüfen
- ▼ die Gruppierung von Steuerelementen testen
- ▼ die Tastenkombinationen bei Dialogfeldern testen (nur für Steuerelemente mit zugewiesenen Zugriffstasten).



Im Testmodus besteht keine Verbindungen der Dialogfelder zum Programmcode des Dialogfeldes.

12.3 Dialoge ohne eigene Klasse

Meist wird ein modales Dialogfeld in ein Programm eingebunden, indem eine Instanz von *CDialog* abgeleitet wird. Diese Instanz enthält Variablen, um die Daten des Dialogfeldes zu speichern, und Methoden, um auf Nachrichten von Steuerelementen zu reagieren. Da *CDialog* von *CWnd* abgeleitet ist, benötigt die Klasse eigene Nachrichtenbehandlungsroutinen, um die Nachrichten mit den Methoden zu verbinden.

Um die Programmierung der Dialogfelder zu erleichtern, wurden in *CDialog* für die drei wichtigsten Nachrichten bereits virtuelle Methoden vorgesehen, die ohne eigene Einträge in den Nachrichtenbehandlungsroutinen auskommen (Tabelle 12.5). Diese können entweder unverändert übernommen oder in einer abgeleiteten Klasse überlagert werden.

Aufgrund des Standardverhaltens dieser virtuellen Methoden ist es sogar möglich, einen einfachen Dialog zu betreiben, ohne eine neue Klasse von *CDialog* abzuleiten – allerdings hat das Dialogfeld dann höchstens zwei aktive Steuerelemente: eine *OK*- und eine *Abbruch*-Schaltfläche.

Dialogfelder und die Klasse CDialog

Methode	Nachricht	Aufruf der Methode	Standardverhalten
BOOL OnInitDialog()	WM_INITDIALOG	Während der Initialisierung des Dialogfeldes, unmittelbar, bevor sie angezeigt wird.	Gibt TRUE zurück und setzt den Eingabefokus auf das erste Steuerelement.
void OnOK()	WM_COMMAND mit IDOK in wParam	Nachdem im Dialogfeld die OK-Schaltfläche gedrückt wurde (vorausgesetzt, er hat den Bezeichner IDOK).	Beendet den Dialog durch Aufruf von EndDialog(IDOK).
void OnCancel()	WM_COMMAND mit IDCANCEL in wParam	Nachdem im Dialogfeld die Abbruch-Schaltfläche gedrückt wurde (vorausgesetzt, er hat den Bezeichner IDCANCEL).	Beendet den Dialog durch Aufruf von EndDialog(IDCANCEL).

Tabelle 12.5: Die wichtigsten Methoden von CDialog

Um solch einen einfachen Dialog aufzurufen, müssen Sie zuerst eine Instanz von *CDialog* erzeugen und anschließend die Methode *DoModal* aufrufen. Zuvor müssen Sie eine Dialogvorlage mit dem Dialog-Editor erstellen. *CDialog* besitzt zwei Konstruktoren, die sich durch die Art der Bezeichnung der Dialogressource unterscheiden: Einer erwartet den Namen der Vorlage als Zeichenfolgen, der andere als Zahl. Beide besitzen einen zweiten Parameter *pParentWnd*, in dem ein Zeiger auf das Vaterfenster des Dialogfeldes übergeben werden kann:

```
class Cdialog:
    CDialog(const char FAR* lpTemplateName,
            CWnd* pParentWnd=NULL);
    CDialog(UINT nIDTemplate,
            CWnd* pParentWnd = NULL);
    virtual int DoModal();
```

Welchen der beiden Konstruktoren Sie verwenden müssen, hängt davon ab, wie die Dialogressource benannt wurde. Wenn der Bezeichner der Vorlage in der Implementierungsdatei als numerischer Wert definiert oder direkt eine Zahlenkonstante als Name angegeben wurde, muß dieser Wert auch im Konstruktor von *CDialog* angegeben werden. Die zweite Version findet Anwendung, wenn Sie für den Bezeichner eine Zeichenfolge verwendet haben, die nicht als numerische Konstante deklariert wurde.

CDialog besitzt übrigens noch einen dritten Konstruktor, der über keine Parameter verfügt. Er wird für das Erzeugen von Dialogfeldern ohne Dialogvorlage benötigt und spielt hier keine Rolle.

Nachdem ein Objekt der Klasse *CDialog* erzeugt wurde, wird das zugehörige Dialogfeld aber noch nicht sofort auf dem Bildschirm angezeigt. Das passiert erst, nachdem Sie die Methode *DoModal* aufgerufen haben. *DoMo-*

dal bereitet den Dialog vor, sendet die Nachricht WM_INITDIALOG (wodurch *OnInitDialog* aufgerufen wird) und zeigt das Dialogfeld auf dem Bildschirm an.

Die modale Eigenschaft des Dialogfeldes schlägt sich im Programm in der Weise nieder, daß *DoModal* erst dann terminiert, wenn die Methode *EndDialog* des Dialogobjekts aufgerufen wurde.

```
class Cdialog:
    void EndDialog(int nResult);
```

Der Rückgabewert von *DoModal* ist dabei gleich dem an *EndDialog* übergebenen Parameter *nResult*.

Durch die Standardimplementierung der Methoden *OnOK* und *OnCancel* wird genau dann *EndDialog* aufgerufen, wenn eine Schaltfläche mit dem Bezeichner IDOK (Wert 1) oder IDCANCEL (Wert 2) gedrückt wurde. Durch das Drücken der Schaltfläche wird nämlich eine entsprechende WM_COMMAND-Nachricht an die Nachrichtenbehandlungsroutinen gesendet, und diese reagiert durch Aufruf von *OnOK* oder *OnCancel*, die ihrerseits *EndDialog* ausführen. Im Falle von *OnOK* wird IDOK an *EndDialog* übergeben, bei *OnCancel* IDCANCEL, so daß Sie anhand des Rückgabewerts von *DoModal* erkennen können, welche der beiden Schaltflächen vom Anwender gedrückt wurde.

Ein Beispiel für ein direkt mit der Klasse *CDialog* erzeugtes Dialogfeld ist der Hilfedialog im Beispielprogramm HEXE. Er wurde mit dem Dialogeditor unter Verwendung des symbolischen Namens IDD_HILFE angelegt. Um den Konstruktor korrekt aufzurufen, ist in diesem Fall der Konstruktor mit dem numerischen Bezeichner zu verwenden:

```
void CHEXEDlg::OnHilfe()
{
    CDialog dlg(IDD_HILFE, this);
    dlg.DoModal();
}
```

Listing: Aufruf eines Dialoges direkt mit der Basisklasse CDialog

Durch den Aufruf von *DoModal* wird das *Hilfe*-Dialogfeld auf dem Bildschirm angezeigt; durch Drücken der OK-Schaltfläche kann ihn der Benutzer beenden. Es macht gar nichts, daß das *Hilfe*-Dialogfeld nur eine OK-, aber keinen *Abbruch*-Schaltfläche hat. Die Methode *OnCancel* ist deshalb trotzdem nicht überflüssig, denn sie wird durch das Drücken der [Esc]-Taste aufgerufen. Das Dialogfeld hätte genausogut auch nur eine *Abbruch*-Schaltfläche haben können. Da in dem Dialogfeld nur eine einzige Schalt-

fläche existiert, die den Dialog beenden kann, kümmert sich die Methode *OnHilfe* auch nicht um den Rückgabewert von *DoModal*, denn der ist hier nicht von Interesse.

12.4 Ableiten eigener Dialogfeldklassen

Derart einfache Dialogfelder besitzen natürlich nicht sehr viele Anwendungen; viel mehr als einen kurzen Hilfetext oder eine einfache Nachricht und Hinweise an den Benutzer können Sie damit nicht anzeigen. Für weitergehende Dialoge muß eine eigene Klasse angelegt werden, die von *CDialog* abgeleitet ist.

12.4.1 Eigenschaften eigener Dialogfeldklassen

Wollen Sie in einem komplexen Dialogfeld neben einer OK- und/oder einer ABBRUCH-Schaltfläche noch weitere Steuerelemente verwalten, müssen Sie eine eigene Klasse von *CDialog* ableiten. Da die abgeleitete Klasse alle Eigenschaften der Elternklasse erbt, braucht sie nur an den Stellen verändert zu werden, an denen die Fähigkeiten von *CDialog* unzureichend sind. Diese Veränderungen treten folgendermaßen in Erscheinung:

1. Die abgeleitete Klasse besitzt zusätzliche Variablen.
2. Sie überlagert Methoden der Basisklasse *CDialog*.
3. Sie definiert (und implementiert) ganz neue Methoden.

Eine Dialogfeldklasse sollte dann eigene Member-Variablen besitzen, wenn sie zusätzliche Daten speichern soll, wenn es also nicht ausreicht, die Steuerelemente beim Initialisieren mit Werten zu laden und die veränderten Werte beim Drücken der OK-Schaltfläche wieder auszulesen. Eigene Variablen sind beispielsweise erforderlich, um die Inhalte der Steuerelemente zwischenspeichern oder zu verarbeiten, zur Kommunikation der Methoden untereinander oder für andere anwendungsspezifische Zwecke. Besitzt eine Dialogfeld beispielsweise eine Schaltfläche, mit der zyklisch zwischen drei verschiedenen Zuständen umgeschaltet werden kann, so muß der aktuelle Zustand irgendwo festgehalten werden. Im Hinblick auf die Lokalität und Kapselung der Daten ist dafür sicherlich die Dialogfeldklasse am besten geeignet.

Der zweite Unterschied zwischen *CDialog* und einer daraus abgeleiteten Klasse betrifft das Überlagern von Methoden der Basisklasse. Zunächst einmal muß in der neuen Klasse der Konstruktor überlagert werden, um die Klasse instantiieren zu können. Werden keine weiteren Methoden überlagert oder hinzugefügt, ändert sich funktional nichts gegenüber *CDialog*; daher werden meist zusätzlich die Methoden *OnInitDialog* und *OnOK* überlagert, manchmal auch *OnCancel*.

Der Konstruktor der abgeleiteten Klasse hat die Aufgabe, die internen Datenstrukturen zu initialisieren und den Konstruktor der Basisklasse aufzurufen. Diesem übergibt er dann auch gleich den richtigen Bezeichner der Dialogvorlage, so daß er selbst ihn nicht mehr als Parameter benötigt. Da Instanzen von Dialogfeldklassen in MFC-Anwendungen zumeist als lokale Variablen zur Laufzeit angelegt werden, sollten Sie im Konstruktor alle Initialisierungen sauber durchführen und so programmieren, daß Sie ihn mehrfach aufrufen können. Falls die Dialogfeldklasse dynamische Datenstrukturen besitzt, oder falls es aus anderen Gründen erforderlich ist, können Sie selbstverständlich auch den Destruktor überlagern.

```
CMyDlg::CMyDlg()
{
    // dynamisches Anlegen einer Klasse im
    // Konstruktor
    m_dynArray = new CObArray;
    // Initialisieren der Variablen
    m_nMutationen = 0;
    m_bSichtbar = TRUE;
    m_nPause = 0;
}

...
CMyDlg::~CMyDlg()
{
    // dynamische Klasse entfernen
    delete m_dynArray;
}
```

Listing: Konstruktor und Destruktor einer abgeleiteten Dialogklasse

Nach der Erzeugung des Objekts und dem Aufruf des Konstruktors erfolgt – noch vor der Anzeige des Dialogfeldes auf dem Bildschirm – ein Aufruf von *OnInitDialog*. In *OnInitDialog* sollten Sie für die *visuelle* Initialisierung des Dialogfeldes sorgen, z. B. wie das Vorbelegen von Bearbeitungsfeldern, das Setzen des Zustands von Schaltflächen oder das Füllen von Listen, das Aktivieren oder Deaktivieren einzelner Steuerelemente oder das Setzen des Eingabefokusses erledigen. Da das Dialogfeld beim Aufruf des abgeleiteten Konstruktors noch nicht vollständig initialisiert ist, können diese Aufgaben erst in *OnInitDialog* erledigt werden.



```
BOOL CMyDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    // Schriftart für Bearbeitungsfeld auf "FETT"
    // setzen
}
```



```

LOGFONT LogFont;
m_edErg.GetFont()->GetLogFont(&LogFont);
LogFont.lfWeight = FW_BOLD;
m_ergFont.CreateFontIndirect(&LogFont);
m_edErg.SetFont(&m_ergFont);

return TRUE;
// ...
}

```

Listing: Anpassen der Schriftart eines Steuerelementes in der Methode OnInitDialog

Die Methode *OnOK* wird aufgerufen, wenn die *OK*-Schaltfläche gedrückt wird. Sie kümmert sich darum, die vom Benutzer geänderten Daten zu verarbeiten und das Dialogfeld zu beenden. In vielen Dialogfeldern ist *OnOK* die aufwendigste Methode, weil sie das Einlesen, Überprüfen und Verarbeiten der Eingabedaten erledigen muß. Neben einer *OK*-Schaltfläche zur ordnungsgemäßen Verarbeitung der Eingabedaten sollte in einem Dialogfeld immer auch eine *Abbruch*-Schaltfläche vorgesehen werden. Besitzt diese den Bezeichner *IDCANCEL*, so wird automatisch die Methode *OnCancel* aufgerufen. Im Gegensatz zu *OnOK* braucht *OnCancel* nicht so häufig überlagert zu werden, denn wenn der Benutzer abbrechen will, brauchen die Daten ja gerade *nicht* eingelesen und verarbeitet zu werden. Sinnvoll ist es allerdings, *OnCancel* zu überlagern, um den Benutzer bei einem versehentlichen Drücken der *Abbruch*-Schaltfläche oder der Esc-Taste vor einem potentiellen Datenverlust zu warnen, insbesondere dann, wenn er sehr viele Änderungen in einem Dialogfeld durchgeführt hat.

Um den Bediener vor einem versehentlichen Verlassen des Dialogfeldes zu schützen, können Sie ihn durch eine einfache *Message.Box* zu einer Bestätigung auffordern und dessen Eingabe auswerten.

```

void CHEXEDlg::OnCancel()
{
    if (AfxMessageBox("HEXE wirklich beenden?",
        MB_YESNO) == IDNO)
        return;
    CDialog::OnCancel();
}

```

Listing: Überlagern der Methode OnCancel, um ein versehentliches Verlassen zu vermeiden

Mit Hilfe eines eigenen Konstruktors und dem Überlagern der Methoden *OnInitDialog*, *OnOK* und *OnCancel* können schon eine ganze Reihe von Dialogen ausgeführt werden. Weitere Methoden werden erst dann erfor-

derlich, wenn zur Laufzeit des Dialogfeldes eine Interaktion mit dem Programm erforderlich ist, etwa weil eine Schaltfläche gedrückt wurde, Abhängigkeiten zwischen verschiedenen Steuerelementen bestehen, Eingaben sofort auf Plausibilität überprüft werden müssen oder gar Nachrichten an das Elternfenster gesendet werden sollen. In diesen Fällen ist es erforderlich, neue Methoden zu definieren und über die Nachrichtenbehandlungsroutinen mit der Dialogklasse zu verbinden.

Die Schleife für die Dialognachrichten einer Dialogklasse sieht prinzipiell genauso aus wie diejenige einer herkömmlichen Fensterklasse. Unterschiede gegenüber dieser bestehen in der Art der Nachrichten, für die sich ein Dialogfeld interessiert. Hier finden sich in erster Linie die *Benachrichtigungscodes* (im Original heißen sie *Notifier-Messages*) von Steuerelementen, mit denen sie bestimmte Änderungen anzeigen. Diese Benachrichtigungscodes sind je nach Art und Eigenschaften eines Steuerelementes unterschiedlich: Schaltflächen senden beispielsweise WM_COMMAND-Nachrichten (genau wie Menüpunkte), andere Steuerelemente senden andere Benachrichtigungscodes. In den nächsten Abschnitten werden die einzelnen Steuerelemente und ihre Benachrichtigungscodes im Detail erklärt.

In Spezialfällen kann ein Dialogfeld auch auf normale Windows-Nachrichten reagieren, etwa auf WM_PAINT. Anwendungen dafür sind aber relativ selten und werden daher normalerweise kaum benötigt. Die Standard-Dialogfeldroutine der MFC erledigt alle Routineaufgaben automatisch: Das Zeichnen des Fensters, die Anzeige der Steuerelemente und das Bearbeiten der Tastaturnachrichten, so daß Sie sich als Programmierer voll auf den logischen Ablauf des Dialogfeldes konzentrieren können.

Hier ein Beispiel, wie die CWnd-Methode *OnCtlColor* in einem Dialogfeld überlagert wird, um die Farbe eines Bearbeitungsfeldes zu ändern:

```
HBRUSH CHXEDlg::OnCtlColor(CDC* pDC, CWnd* pWnd,
                           UINT nCtlColor)
{
    HBRUSH hbr = CDialog::OnCtlColor(pDC, pWnd,
                                     nCtlColor);
    if (pWnd->GetDlgCtrlID() == IDD_OUTPUT_ERG)
    {
        // blaue Schrift auf weißem Hintergrund
        pDC->SetBkColor(RGB(255,255,255));
        pDC->SetTextColor(RGB(0,0,255));
    }
    return hbr;
}
```

Listing: Anpassen der Text- und Hintergrundfarben von Steuerelementen

12.4.2 Dialogfeld-Klasse mit dem Klassen-Assistenten

Haben Sie die Anwendung mit dem MFC-Anwendungs-Assistenten erzeugt, können Sie zum Anlegen von Klassen, Variablen und Nachrichtenbehandlungsroutinen der Klassen-Assistenten benutzen. Dieser vereinfacht den Entwicklungsprozeß erheblich, da damit große Teile des Quelltextes automatisch generiert werden. Der Klassen-Assistent erzeugt die Implementierungs- und Schnittstellendatei und fügt die Deklaration und den Funktionsrumpf ein, so daß hier Fehler vermieden werden. Weiterhin sorgt er für das korrekte Einbinden von Member-Variablen und Member-Funktionen. Außerdem unterstützt er das Überlagern der Funktionen der Basisklasse (siehe Kapitel 3.2).

Dem Arbeiten mit dem Klassen-Assistenten geht das Erstellen eines Dialogfeldes im Dialog-Editor des Developer Studios voraus. Wenn Sie mit dem Dialog-Editor ein neues Dialogfeld angelegt oder ein vorhandenes kopiert haben, muß für diese Ressource in dem Projekt eine neue von *CDialog* abgeleitete Klasse angelegt werden. Um dieses Dialogfeld mit dem entsprechenden Programmcode zu versehen, muß der Klassen-Assistent aktiviert werden. Dies geschieht über die Menüs der rechten Maustaste im Dialogeditor bzw. über das Hauptmenü ANSICHT|KLASSEN-ASSISTENT ... (siehe Abbildung 12.11) oder über die Tastenkombination **[Strg] [W]**.

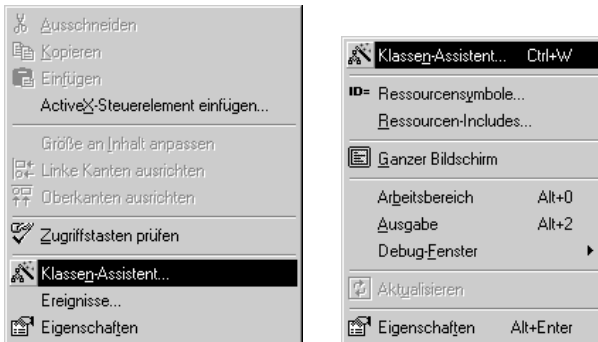


Abbildung 12.11: Menü »Rechte Maustaste« und »Ansicht«

Beim ersten Aufruf zu einem neuen Dialogfeld-Symbolnamen erscheint der folgende Dialog: Sie können entweder eine neue Klasse erstellen lassen, oder den Symbolnamen (hier IDD_DIALOG1) mit einer bereits vorhandenen Ableitung von *CDialog* verbinden.

Das Zuweisen des symbolischen Namens zu einer vorhandenen Klasse ist z. B. bei der Neuanlage der Klassen-Assistenten-Datei (*.clw) erforderlich.



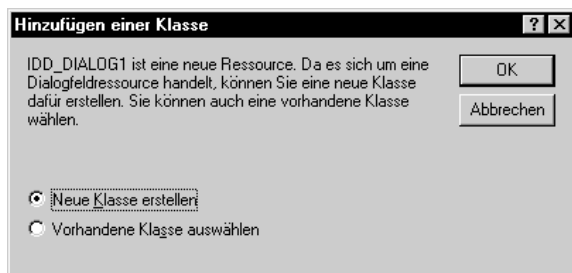


Abbildung 12.12: Erzeugen einer neuen Dialogfeldklasse

Wird dieser Dialog mit OK bestätigt, müssen die erforderlichen Angaben zum Erstellen einer neuen Klasse in dem Dialog *Neue Klasse* eingegeben werden (siehe Abbildung 12.13). An erster Stelle steht der Name der neuen Klasse. Dieser ist frei wählbar, entsprechend den im Kapitel »Klassen-Assistent« erläuterten Regeln. Die Dateinamen für Schnittstellen- und Implementierungsdatei (*MyDlg.h* und *MyDlg.cpp*) werden automatisch erzeugt. Sie können diese jedoch über die Schaltfläche *ÄNDERN* Ihren Bedürfnissen entsprechend modifizieren oder bereits vorhandene Dateien auswählen, um mehrere Klasse in einer gemeinsamen Datei abzulegen. Die Basisklasse, von der Ihre eigene abgeleitet werden soll, wird ebenfalls vorgelegt (*CDialog*). Die Basisklasse kann bequem aus dem Listefeld ausgewählt werden. Automatisch vorgelegt ist auch die Dialogfeld-ID. Soll die Klasse für eine andere Ressource angelegt werden, können Sie auch diese aus einem Listefeld auswählen. In dem Gruppenfeld *Automatisierung* (früher als OLE-Automatisierung bezeichnet) können Sie über die Art der Offenlegung entscheiden, also ob Sie Objekte bearbeiten möchten oder Ihre Klasse freigeben wollen (oder keines von beiden). Im Fall der Freigabe müssen Sie einen Namen vergeben.

Nach dem Drücken der OK-Schaltfläche wird die Klasse für den Dialog in den angegebenen Dateien angelegt. Der so entstandene Programmcode ist übersetzbar und enthält bereits einige abgeleitete Funktionen von *CDialog*. Der Dialog *Neue Klasse* ist auch über das Menü *EINFÜGEN|NEUE KLASSE* bzw. im Dialog *MFC-Klassen-Assistent* mit der Schaltfläche *KLASSE HINZUFÜGEN* zu erreichen.

```
class CMyDlg : public CDialog
{
// Konstruktion
public:
    CMyDlg(CWnd* pParent = NULL);    // Standardkonstruktor
// Dialogfelddaten
//{{AFX_DATA(CMyDlg)
```

```

enum { IDD = IDD_DIALOG1 };
    // HINWEIS: Der Klassen-Assistent fügt hier
    //      Datenelemente ein
    ///}AFX_DATA
// Überschreibungen
// Vom Klassen-Assistenten generierte virtuelle
// Funktionsübersreibungen
///{AFX_VIRTUAL(CMyDlg)
protected:
virtual void DoDataExchange(CDataExchange* pDX);
                                // DDX/DDV-Unterstützung

    ///}AFX_VIRTUAL
// Implementierung
protected:
    // Generierte Nachrichtenzuordnungsfunktionen
    ///{AFX_MSG(CMyDlg)
        // HINWEIS: Der Klassen-Assistent fügt hier
        //      Member-Funktionen ein
    ///}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

```

Listing: Vom Klassen-Assistenten erzeugte Schnittstellendatei zu einer Dialogvorlage

```

CMyDlg::CMyDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CMyDlg::IDD, pParent)
{
    ///{AFX_DATA_INIT(CMyDlg)
        // HINWEIS: Der Klassen-Assistent fügt hier
        //      Elementinitialisierung ein
    ///}AFX_DATA_INIT
}

void CMyDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    ///{AFX_DATA_MAP(CMyDlg)
        // HINWEIS: Der Klassen-Assistent fügt hier
        //      DDX- und DDV-Aufrufe ein
    ///}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CMyDlg, CDialog)
    ///{AFX_MSG_MAP(CMyDlg)
        // HINWEIS: Der Klassen-Assistent fügt hier

```

```
//          Zuordnungsmakros für Nachrichten ein  
//}}AFX_MSG_MAP  
END_MESSAGE_MAP()
```

Listing: Vom Klassen-Assistenten erzeugte Implementierungsdaten zu einer Dialogvorlage

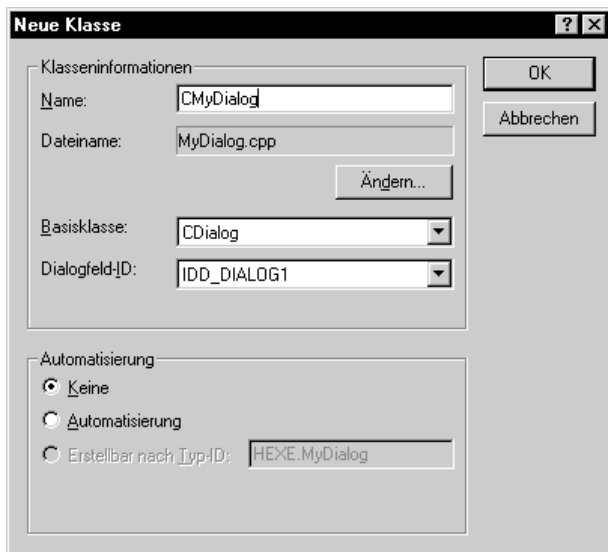


Abbildung 12.13: Klasseninformationen zu einer neuen Klasse

Die so vom Assistenten erzeugten Dateien enthalten neben dem Programmcode für die abgeleitete Klasse auch Schlüsselwörter für die weitere Anwendung des Klassen-Assistenten in Form von Kommentar wie beispielsweise `//{{AFX_DATA_INIT(CMyDlg)}` oder `//{{AFX_VIRTUAL(CMyDlg)}`.

12.5 Dialogbasierende Anwendung – ein Beispiel

12.5.1 Vorteile und Besonderheiten

Eine dialogbasierende Anwendung ist ein Windows-Programm, bei dem das Hauptfenster ein Dialog von zumeist fester Größe ist. Diese Art der Anwendungen stellt eine gute Alternative zu MDI- und SDI-Anwendungen der MFC dar, wenn es darum geht, mit einer kleinen Programmstruktur auszukommen. Eine verbreitete Anwendung sind Setup-Programme (Assistenten und Zauberer), Eigenschaftsdialoge zur Konfiguration von Software oder kleine Dienstprogramme, deren Anzeigeelemente sich auf wenige Steuerungen beschränken.

Solche Programme können aber auch anspruchsvolle Lösungen repräsentieren. So ist es möglich, diese Programme mit einem Hilfesystem, aber auch mit Datenbankabfragen zu versehen.

Der Anwendungs-Assistent vom Developer Studio unterstützt das Erzeugen und Bearbeiten solcher dialogbasierenden Anwendungen optimal. Mit ihm können Sie leicht ein entsprechendes Programmgerüst erzeugen und Ihren Anforderungen entsprechend ausbauen.

Das folgende Beispielprogramm HEXE ist eine dialogbasierende Anwendung und stellt einen Taschenrechner dar. Der Dialog mit dem Rechner wird sofort nach dem Programmstart sichtbar; die Kontrolle bleibt während der Laufzeit immer im Dialog. Mit dem Beenden des Dialogs wird die ganze Anwendung beendet.

Die erste Besonderheit ist in der Methode *InitInstance* der Anwendung zu sehen. Normalerweise liefert diese Methode ein TRUE zurück, wenn sie beendet wird. Anschließend steht das Programm in der Nachrichtenschleife und wartet auf weitere Ereignisse. Anders ist das bei dialogbasierenden Anwendungen: Hier wird in *InitInstance* ein modaler Dialog erzeugt; nach dessen Beendigung kehrt *InitInstance* mit FALSE zurück, wodurch das Programm beendet wird. Im Beispielprogramm HEXE sieht das so aus:

```

BOOL CHEXEApp::InitInstance()
{
    // ...
#ifdef _AFXDLL
    Enable3dControls(); // Diese Funktion bei Verwendung von MFC
                       // in gemeinsam genutzten DLLs aufrufen
#else
    Enable3dControlsStatic(); // Diese Funktion bei statischen
                             // MFC-Anbindungen aufrufen
#endif

    CEXEDlg dlg;
    m_pMainWnd = &dlg;
    int nResponse = dlg.DoModal();
    if (nResponse == IDOK)
    {
        // ZU ERLEDIGEN: Fügen Sie hier Code ein, um ein Schließen
        // des Dialogfelds über OK zu steuern
    }
    else if (nResponse == IDCANCEL)
    {
        // ZU ERLEDIGEN: Fügen Sie hier Code ein, um ein Schließen
        // des Dialogfelds über "Abbrechen" zu steuern
    }
}

```

```

// Da das Dialogfeld geschlossen wurde, FALSE zurückliefern,
// so dass wir die Anwendung verlassen, anstatt das
// Nachrichtensystem der Anwendung zu starten.
return FALSE;
}

```

Listing: Die Methode InitInstance bei einer dialogbasierenden Anwendung

Die zweite Besonderheit ist im Dialog selbst zu sehen. Da das Dialogfenster das Hauptfenster der Anwendung ist, muß es auf einige zusätzliche Nachrichten reagieren. Dazu trägt der Anwendungs-Assistent folgende drei Methoden in die Nachrichtenbehandlungsroutine ein und generiert den entsprechenden Programmcode wie folgt:

```

BEGIN_MESSAGE_MAP(CHEXEDlg, CDialog)
//{{AFX_MSG_MAP(CHEXEDlg)
ON_WM_SYSCOMMAND()
ON_WM_PAINT()
ON_WM_QUERYDRAGICON()
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

```

Listing: Drei zusätzliche Behandlungsroutinen

Die entsprechenden Member-Funktionen liefert der Anwendungs-Assistent gleich mit. Die erste Methode, *OnSysCommand*, dient der Anzeige des *Info über ...*-Dialogs. Dazu werden die im Dialog eintreffenden Nachrichten überprüft und ggf. der Dialog *CAboutDlg* angezeigt.

```

void CHEXEDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
    if ((nID & 0xFFFF) == IDM_ABOUTBOX)
    {
        CAboutDlg dlgAbout;
        dlgAbout.DoModal();
    }
    else
    {
        CDialog::OnSysCommand(nID, lParam);
    }
}

```

Listing: Aufruf des Info über...-Dialogs

Zum anderen werden Methoden zur Anzeige des Programmsymbols bei minimierter Anwendung bereitgestellt. Diese Aufgabe übernimmt normalerweise das Hauptfenster. Dieses Verfahren ist allerdings nur noch für

Windows NT 3.51 von Bedeutung, da hier das Symbol auf dem Desktop angezeigt wird. Bei Windows 95 und neueren NT-Versionen wird das Programm direkt in der Taskleiste dargestellt. Ein Programmsymbol in dieser Form ist dann nicht erforderlich.

```
void CHEXEDlg::OnPaint()
{
    if (IsIconic())
    {
        CPaintDC dc(this); // Gerätekontext für Zeichnen
        SendMessage(WM_ICONERASEBKGND,
                    (LPARAM)dc.GetSafeHdc(), 0);
        // Symbol in Client-Rechteck zentrieren
        int cxIcon = GetSystemMetrics(SM_CXICON);
        int cyIcon = GetSystemMetrics(SM_CYICON);
        CRect rect;
        GetClientRect(&rect);
        int x = (rect.Width() - cxIcon + 1) / 2;
        int y = (rect.Height() - cyIcon + 1) / 2;
        // Symbol zeichnen
        dc.DrawIcon(x, y, m_hIcon);
    }
    else
    {
        CDialog::OnPaint();
    }
}

//...
HCURSOR CHEXEDlg::OnQueryDragIcon()
{
    return (HCURSOR) m_hIcon;
}
```

Listing: Methoden zur Darstellung des Programmsymbols

Für die Anzeige des *Info über ...*-Dialogs muß noch ein Menüpunkt bereitgestellt werden, der die entsprechende Systemnachricht versendet. Der erforderliche Programmcode ist ebenfalls vom Assistenten erstellt worden. In *OnInitDialog* wird das Systemmenü entsprechend erweitert.

```
BOOL CHEXEDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    //...
    ASSERT((IDM_ABOUTBOX & 0xFFFF) == IDM_ABOUTBOX);
```

```

ASSERT(IDM_ABOUTBOX < 0xF000);
CMenu* pSysMenu = GetSystemMenu(FALSE);
if (pSysMenu != NULL)
{
    CString strAboutMenu;
    strAboutMenu.LoadString(IDS_ABOUTBOX);
    if (!strAboutMenu.IsEmpty())
    {
        pSysMenu->AppendMenu(MF_SEPARATOR);
        pSysMenu->AppendMenu(MF_STRING,
            IDM_ABOUTBOX, strAboutMenu);
    }
}
SetIcon(m_hIcon, TRUE); // Großes Symbol verw.
SetIcon(m_hIcon, FALSE); // Kleines Symbol verw.
return TRUE;
}

```

Listing: Erweitern des Systemmenüs

Da das Hauptfenster der Anwendung der Dialog ist, hat es auch immer die Größe der Dialogvorlagen, wie sie mit dem Dialogeditor angelegt wurden. Eine Größenänderung des Hauptfensters durch das Ziehen mit der Maus am Fensterrahmen ist nicht möglich. Weitere Besonderheiten sind bei dialogbasierenden Anwendungen nicht zu beachten.

12.5.2 Aufgabe

Hier folgt ein ausführliches Anwendungsbeispiel zur Illustration der Möglichkeiten einer dialogbasierenden Anwendung. Dabei soll auf Bekanntes aus den vorangegangenen Abschnitten, wie die Arbeit mit den Assistenten und der Umgang mit Steuerungen, zurückgegriffen werden.

HEXE ist ein Taschenrechner, der die vier Grundrechenarten sowie logische und bitweise arbeitende Operatoren beherrscht und dem Programmierer das Rechnen in verschiedenen Zahlensystemen (auch dem hexadezimalen, *nomen est omen*) ermöglicht.

Das Programm besitzt ein einfaches Menü, das neben dem Standard-Systemmenü den Punkt *Info über ...* enthält. Der Taschenrechner verfügt über ein attraktives Tastenfeld zur Eingabe von Ziffern und Operatoren sowie über ein Multifunktionsdisplay zur Anzeige der Ergebnisse in den verschiedenen Zahlensystemen.

12.5.3 Bedienung

Die Eingabelogik entspricht dem algebraischen Operationssystem und kommt daher der gewöhnlichen Schreibweise arithmetischer Ausdrücke entgegen: Soll vier und sechs addiert werden, ist $\boxed{4} \boxed{+} \boxed{6} \boxed{=}$ einzugeben.

Der Rechner arbeitet grundsätzlich mit 32-Bit-Ganzzahlarithmetik, d.h., alle Operanden sind vom Typ `long`, und weder bei der Eingabe von Ziffern noch bei Rechenoperationen werden numerische Überläufe beachtet. Das hat den Vorteil, daß genau dieselben Effekte beobachtet werden können, die bei `long`-Arithmetik in eigenen Programmen auftreten.

An arithmetischen Operatoren stehen Addition, Subtraktion, Multiplikation, ganzzahlige Division und Modulo ($\boxed{+}$, $\boxed{-}$, $\boxed{*}$, $\boxed{/}$, $\boxed{\%}$) zur Verfügung. Daneben gibt es die bitweise arbeitenden logischen UND-, ODER- und EXKLUSIV-ODER-Verknüpfungen (auf der Tastatur dargestellt als $\boxed{\&}$, $\boxed{\vee}$ und $\boxed{\wedge}$) sowie die Links- und Rechtsschiebe-Operatoren ($\boxed{<<}$ und $\boxed{>>}$). Als unäre Operatoren stellt der Rechner Vorzeichenwechsel ($\boxed{+/-}$) und Einerkomplement ($\boxed{\sim}$) zur Verfügung. Mit der (BS)-Taste wird das Ergebnis ziffernweise gelöscht, mit $\boxed{\leftarrow}$ komplett.

Die Ausgabe erfolgt über eine Registerkarte wahlweise im binären, oktaalen, dezimalen oder hexadezimalen Format. Eine zusätzliche Ausgabezeile interpretiert Ergebnisse zwischen 0 und 255 als darstellbares Zeichen aus dem Windows-Zeichensatz (MS Sans Serif) und ersetzt so in vielen Fällen den Blick in die Zeichensatztabelle. Im dezimalen Zahlensystem arbeitet der Rechner mit vorzeichenbehafteten Zahlen, andernfalls mit vorzeichenlosen. Werte kleiner Null werden in einer anderen Farbe dargestellt. Mit dem Kombinationsfeld am oberen Ende können die letzten 20 Ergebnisse wieder in die Anzeige gerufen werden, der Buchstabe vor dem Ergebnis bezeichnet das Zahlensystem, das zum Zeitpunkt des Entstehens eingestellt war.

Als besonderen Clou gibt es die Möglichkeit, Zahlen *analog* einzugeben. Dazu dient der Schieberegler auf der rechten Seite, mit dessen Schieber Werte zwischen 0 und 255 eingestellt werden können. Das Klicken auf die Schaltflächen verändert den angezeigten Wert um eins, das Klicken zwischen Schieber und Schaltflächen verändert ihn um den Basiswert des aktiven Zahlensystems. Bei digitalen Eingaben zwischen 0 und 255 dient der Schieber als analoges Anzeigeinstrument; liegt der Wert außerhalb dieser Grenzen, bleibt er am jeweiligen Ende stehen.

Der Rechner kann entweder durch Drücken der $\boxed{\text{AUS}}$ -Schaltfläche oder durch $\boxed{\text{Esc}}$ beendet werden.

12.5.4 Implementierung

Die Arithmetik in HEXE arbeitet mit zwei `long`-Registern `areg` und `breg`. `areg` entspricht immer dem Wert in der Anzeige und wird verändert, wenn Zahlen eingegeben werden oder das Ergebnis ermittelt wird. Beim Drücken einer Operator-Taste wandert der aktuelle Inhalt von `areg` nach `breg`, so daß `areg` vor der Eingabe der nächsten Ziffer auf Null gesetzt werden kann. Die Berechnung eines Ergebnisses erfolgt, indem die Operanden `areg` und `breg` durch die Operator-Taste verknüpft werden, danach wird `breg` wieder auf Null gesetzt.

Beim Drücken einer Operator-Taste merkt sich HEXE den Operator in der Variablen `op`, so daß er später beim Drücken der Ergebnis-Taste abgerufen werden kann. `op` wird auch abgefragt, wenn eine weitere Operator-Taste gedrückt wird; auf diese Weise sind Kettenrechnungen wie $2 + 5 + 8 + 10 =$ möglich. Von der Regel »Punktrechnung vor Strichrechnung« weiß HEXE allerdings nichts.

12.5.5 Projektanlage mit dem MFC-Anwendungs-Assistenten

Jetzt geht es los! Legen Sie ein neues MFC-Projekt über DATEI|NEU|PROJEKTE an. Geben Sie ihm den Namen *HEXE*, und betätigen Sie die OK-Schaltfläche. Nun beginnt der Anwendungs-Assistent mit seiner Arbeit. Beim Anlegen des Programmgerüsts bietet er in Schritt 1 drei unterschiedliche Anwendungsarten zur Auswahl an (siehe Abbildung 12.14): Zwei dokumentbasierende und eine dialogbasierende Anwendung. Die dokumentbasierenden Anwendungen sind Voraussetzung für die *Document-View-Architektur*, die in späteren Abschnitten erläutert wird. Um das Programmgerüst für HEXE anzulegen, wählen Sie *Dialogfeldbasierend*.

In den weiteren Schritten zwei bis vier können alle Standardeinstellungen übernommen werden. Wie bereits bekannt, entsteht so ein Programmgerüst auf MFC-Basis, welches nach der Übersetzung bereits voll lauffähig ist.



Um das Programm klein zu halten, können Sie die Option *AktivX Steuerelemente* in Schritt vier deaktivieren.

Welche Dateien erzeugt nun der Anwendungs-Assistent für das Projekt HEXE? Sie finden die Dateien in dem Projektverzeichnis HEXE und dessen Unterverzeichnissen. Eine Ansicht der Dateien ist auch im *Dateien*-Fenster des Arbeitsbereiches möglich.

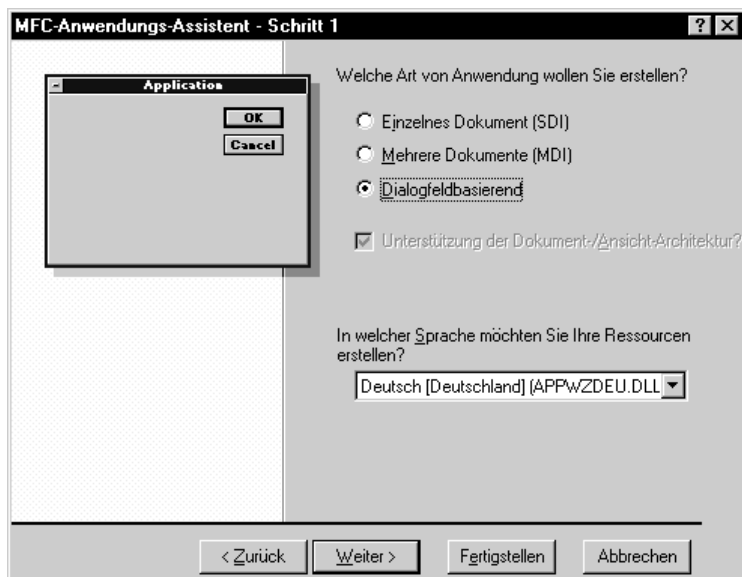


Abbildung 12.14: Anwendungs-Assistent, Dialogfeldbasierend

Dateiname	Bedeutung, Inhalt
HEXE.h HEXE.cpp	Implementierungs- und Schnittstellendatei für die Anwendung. HEXE, in diesen Dateien ist die von <i>CwinApp</i> abgeleitete Applikationsklasse enthalten; sie legt das Klassenverhalten für die Anwendung fest.
HEXEDlg.h HEXEDlg.cpp	In diesen Dateien ist das Dialogfeld mit einer von <i>CDialog</i> abgeleiteten Klasse enthalten. Zusätzlich ist noch der Info über ...-Dialog integriert (<i>CaboutDlg</i>).
Hexe.rc Resource.h	Die Ressource-Datei zum Projekt mit allen bereits vorhandenen wichtigen Ressourcen wie Icon, , Info über ...-Dialog, Dialogvorlage für den Hauptdialog und eine Zeichenkette. In Resource.h sind die zugehörigen symbolischen Konstanten definiert. Weitere Ressourcen sind im Unterverzeichnis RES abgelegt.
StdAfx.h StdAfx.cpp	Enthält die Anweisungen zum Einbinden der Schnittstellendatei der MFC. Aus diesen wird die vorcompilierte Header-Datei <i>HEXEt.pch</i> erzeugt, um den Erstellungsvorgang zu beschleunigen.
HEXE.clw	Binärdatei mit Informationen für den Klassen-Assistenten.
ReadMe.txt	Eine Textdatei mit Informationen zu den angelegten Dateien, ähnlich wie in dieser Tabelle beschrieben.

Tabelle 12.6: Vom Anwendungs-Assistenten erzeugte Dateien

Der Rechnerdialog `IDD_HEXE_DIALOG` von `HEXE` wird durch die Klasse `CHEXEDlg` repräsentiert und ist in die Dateien `HEXEDlg.h` und `HEXEDlg.cpp` implementiert worden.

Jetzt beginnt die Handarbeit. Sie müssen die Dialogvorlage bearbeiten und den zusätzlichen Programmcode generieren. Dazu steht zum einen der Dialog-Editor und zum anderen der Klassen-Assistent zur Verfügung.

12.5.6 Gestaltung der Dialogvorlage

Nun geht es darum, wie in Kapitel 12.2 beschrieben für den Taschenrechner eine Dialogvorlage im Editor zu erstellen bzw. zu bearbeiten. Dazu wird die vom Anwendungs-Assistenten erzeugte Dialogvorlage IDD_HEXE_DIALOG in den Editor geholt.

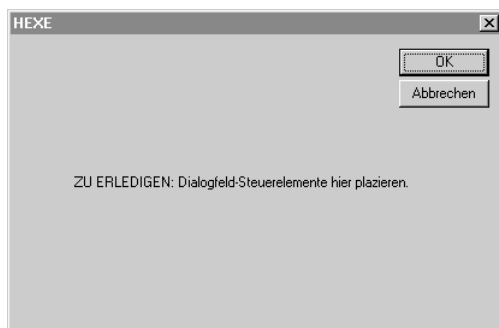


Abbildung 12.15: Neue Dialogvorlage für den Taschenrechner

Die erzeugte Dialogvorlage hat natürlich noch nichts von einem Taschenrechner. Jetzt ist Ihre Kreativität gefragt, um aus dieser allgemeinen Vorlage einen Rechnerdialog zu machen.

Als erstes werden die Eigenschaften der Dialogvorlage angepaßt. Das sind im einzelnen:

1. Vergrößern der Dialogfläche (auf 208x218 Punkte).
2. Hinzufügen der MINIMIEREN-Schaltfläche über den Dialog EIGENSCHAFTEN|FORMATE.
3. Einstellen und positionieren der Führungslinien über das Menü LAYOUT|EINSTELLUNG DER FÜHRUNGSLINIEN.
4. Entfernen der ABBRUCH-Schaltfläche.
5. Umbenennen der OK-Schaltfläche in AUS und positionieren in der linken unteren Ecke der Dialogvorlage.

Nun muß das Ausgabefeld implementiert werden. Da die Anzeige für mehrere Zahlenbereiche möglich sein soll, bietet sich hier ein Eingabefeld (IDD_OUTPUT_ERG) in Kombination mit einer Registerkarte (IDC_TAB_AUSGABE) an. Dabei muß am rechten Rand genügend Platz für die Bildlaufleiste bleiben.

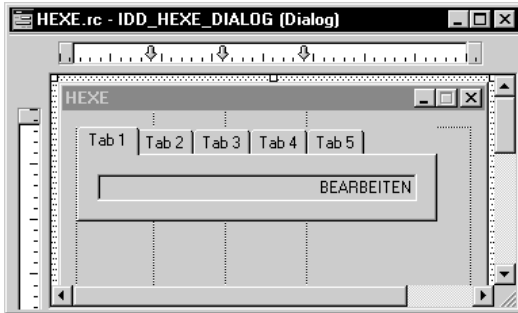


Abbildung 12.16: »Dialogvorlage« Taschenrechner in Arbeit

Der Rest ist Fleißarbeit. Die Schaltflächen zur Operator- und Zifferneingabe müssen auf die noch freie Fläche der Dialogvorlage verteilt werden. Dabei nutzen Sie zum Positionieren und Anpassen der Größe die oben beschriebenen Möglichkeiten des Dialog-Editors. Die Schaltflächen erhalten die Bezeichnung `IDD_BUTTON_0` bis `IDD_BUTTON_F`. Um Operator- und Ziffernschaltflächen optisch voneinander unterscheiden zu können, bekommen sie unterschiedliche *Format*-Eigenschaften. Dazu müssen Sie zunächst für beide die Eigenschaft `FORMAT|STANDARDSCHALTFLÄCHE` deaktivieren und dafür unter `ERWEITERTE FORMATE|MODALER RAHMEN` anschalten. Für die Operatoren wird noch *Client-Kante* aktiviert. Nun fügen Sie am rechten Rand eine vertikale Bildlaufleiste über die gesamte Höhe der Dialogvorlage ein. Testen Sie die Vorlage, und überprüfen Sie die Tabulator-Reihenfolge!

12.5.7 Zusätzlicher Programmcode

Nun geht es darum, das vom Anwendungs-Assistenten erzeugte Gerüst mit Leben zu füllen. Dabei geht es im wesentlichen um folgende Aufgaben:

1. Hinzufügen eigener Funktionen und Variablen
2. Zugriff auf Steuerungen
3. Erweitern der erzeugten Funktionen um eigenen Programmcode
4. Verbinden der Steuerung mit dem Programmcode und Zugriff auf die Steuerungen
5. Überlagern von Funktionen der Basisklassen

Variablen hinzufügen

Dabei wird der Klassen-Assistent gute Dienste leisten. Fügen Sie also, wie im Abschnitt über die Assistenten beschrieben, in die Klasse `CHEXED1g` folgende Variablen ein:

```
private:
    long areg, breg;    // die beiden Register
    int nbase;         // Zahlenbasis für Ausgabe
    int op;            // ID des Operators
    BOOL new_number;   // Kennzeichen für neue Nummer
```

Initialisieren Sie diese im Konstruktor der Klasse.

```
CHEXEDlg::CHEXEDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CHEXEDlg::IDD, pParent)
{
    ///{AFX_DATA_INIT(CHEXEDlg)
        // HINWEIS: Der Klassen-Assistent fügt hier
        // Member-Initialisierung ein
    ///}AFX_DATA_INIT
    // Beachten Sie, dass LoadIcon unter Win32
    // keinen nachfolgenden DestroyIcon-Aufruf
    // benötigt
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
    new_number = TRUE;
    op = 0;
    areg = 0;
    breg = 0;;
    nbase = 10;
}
```

Listing: Erweiterung des Konstruktors zur Initialisierung der Variablen

Fügen Sie nun die beiden Funktionen `GetBasedString` und `UpdateCalculatorDisplay` zur Ausgabeunterstützung hinzu:

```
private:
    CString GetBasedString(long num, int base,
                           BOOL bMitBasis = FALSE);
    void UpdateCalculatorDisplay();
```

Zugriff auf Steuerungen

Als nächstes müssen Sie vom Programm aus auf die Steuerungen zugreifen. Mit den MFCs ist der Zugriff auf die einzelnen Steuerungen eines Dialogfeldes sehr viel komfortabler geworden. Die Methode *GetDlgItem* von *CWnd* spielt dabei die Schlüsselrolle:

```
class Cwnd:
    CWnd *GetDlgItem(int nID);
```


Ein Aufruf von *GetDlgItem* liefert einen Zeiger auf ein temporäres *CWnd*-Objekt, das der Steuerung mit dem Bezeichner *nID* entspricht bzw. NULL, wenn keine Steuerung mit diesem Bezeichner existiert.

Zwar können auf dieses Fenster-Objekt nur alle Methoden von *CWnd* angewendet werden, doch darunter sind viele, die auch sehr gut auf Steuerungen angewendet werden können (z.B. *SetFocus* und *EnableWindow*). Beispiele dafür finden sich bereits in der Methode *OnOldResultSelected* des Rechnerdialogs:

```
void CHEXEDlg::OnOldResultSelected()
{
    //...
    GetDlgItem(IDD_BUTTON_EQUAL)->SetFocus();
    UpdateCalculatorDisplay();
}
```



Eine weitere Möglichkeit, die Methoden der Steuerelemente zu benutzen, erschließt sich durch das Anlegen eigener Instanzen mit dem Klassen-Assistenten. Auf die Member-Funktionen der Steuerelement-Klasse kann dann über die Variable zugegriffen werden. Es wird für jede Steuerung ein entsprechender Eintrag in der Funktion *DoDataExchange* vorgenommen, über die der Datenaustausch abgewickelt wird.

```
void CHEXEDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CHEXEDlg)
    DDX_Control(pDX, IDD_OUTPUT_CHAR, m_statZeichen);
    DDX_Control(pDX, IDD_OUTPUT_LIST, m_cbListe);
    DDX_Control(pDX, IDD_SCROLLBAR, m_scrZeichenWert);
    DDX_Control(pDX, IDD_OUTPUT_ERG, m_edErg);
    DDX_Control(pDX, IDC_TAB_AUSGABE, m_TabAnzeige);
    //}}AFX_DATA_MAP
}
```

Listing: Verbinden der Steuerungen der Dialogvorlage mit den entsprechenden Klassen

Eine weitere wichtige Methode, die auf Steuerungen angewendet werden kann, ist die Methode *EnableWindow* der Klasse *CWnd*.

```
class Cwnd:
    BOOL EnableWindow(BOOL bEnable = TRUE);
```

Diese beeinflußt die Reaktion von Steuerungen auf Benutzereingaben. Mit *EnableWindow* kann eine Steuerung aktiviert oder deaktiviert werden. Ist sie aktiviert, kann sie Benutzereingaben empfangen und verarbeiten und reagiert gemäß ihren speziellen Eigenschaften. Ist sie deaktiviert, kann sie zwar vom Programm aus angesteuert und verändert werden, reagiert aber nicht mehr auf Benutzereingaben. Visuell wird der deaktivierte Zustand in der Regel dadurch sichtbar gemacht, daß die Steuerung oder ihr Inhalt in einer anderen Farbe – zumeist grau – dargestellt wird. Der Parameter *bEnable* gibt an, ob die Steuerung aktiviert (TRUE) oder deaktiviert (FALSE) werden soll. Welchen der beiden Zustände eine Steuerung unmittelbar nach dem Aufbau des Dialogfeldes hat, hängt davon ab, wie sie in der Dialogboxschablone definiert wurde; der Dialogeditor läßt beide Möglichkeiten zu.

In HEXE verwenden Sie *EnableWindow*, um bei der Initialisierung des Dialogfeldes das Kombinationsfeld mit den alten Ergebniswerten zu aktivieren, denn nach dem Start sind ja noch keine Ergebnisse darin enthalten. Erst nach dem ersten Drücken der Ergebnistaste – nach der Übertragung des aktuellen Wertes in das Kombinationsfeld – wird diese aktiviert und kann vom Benutzer verwendet werden:



```
void CHEXEDlg::OnEqual()
{
    //...
    //---Combobox aktualisieren---
    m_cbListe.EnableWindow(TRUE);
    //...
}
```

Funktionen hinzufügen

Jetzt können die beiden oben eingefügten Funktionen ausprogrammiert werden. *UpdateCalculatorDisplay* ist dabei für die Anzeige des Rechenergebnisses verantwortlich. Diese Ausgabe erfolgt auf drei Arten:

1. Zahlenausgabe des Ergebnisses im Eingabefeld *m_edErg*; dazu dient die unten beschriebene Hilfsfunktion *GetBasedString*.
2. Darstellung des Zeichensatz-Zeichens entsprechend dem Wert des Ergebnisses im Eingabefeld *m_statZeichen*; dabei wird das betreffende Zeichen durch ein vorangesetztes »&« unterstrichen ausgegeben.
3. Positionieren der Bildlaufleiste laut dem Ergebniswert; eine Überprüfung des Wertebereiches (0...255) ist notwendig.

```
void CHEXEDlg::UpdateCalculatorDisplay()
{
    CString tmpStr;
```

```

//---Zahlenausgaben---
m_edErg.SetWindowText(GetBasedString(areg, nbase));
//---Zeichensatz---
if (areg >= 0 && areg <= 255)
{
    tmpStr = "          ";
    int n = areg % 16;
    for (int i = 0; i < n; ++i)
        tmpStr.SetAt(i, areg - n + i);
    tmpStr.SetAt(n, '&');
    tmpStr.SetAt(n + 1, (int)areg);
    for (i = n + 1; i < 16; ++i)
        tmpStr.SetAt(i + 1, areg - n + i);
}
m_statZeichen.SetWindowText(tmpStr);
//---Bildlaufleiste positionieren---
if (areg < 0)
    m_scrZeichenWert.SetScrollPos(255, TRUE);
else
    if (areg > 255)
        m_scrZeichenWert.SetScrollPos(0, TRUE);
    else
        m_scrZeichenWert.SetScrollPos(255 - areg, TRUE);
}

```

Listing: Ansteuern der Rechneranzeige

Im Bereich zwischen 32 und 38 funktioniert die Zeichendarstellung nicht korrekt. Das liegt an dem Zeichen »&« in diesem Zeichenkettenabschnitt, das zur Steuerung des Unterstrichs verwendet wird.



Die Funktion `GetBasedString` ist eine Hilfsfunktion zur Aufbereitung der Ausgabezeichenkette im aktuellen Zahlenformat. Dabei wird unterschieden, ob nur das Ergebnis oder auch der Zahlenbereich mitausgegeben werden soll (`bMitBasis`).

```

CString CHEXEDlg::GetBasedString(long num, int base, BOOL bMitBasis)
{
    CString tmpStr;
    switch (base)
    {
        case 16:
            tmpStr.Format("%s%lX", bMitBasis ? "H ":"", num);
            break;
        case 10:

```

```

        tmpStr.Format("%s%d", bMitBasis ? "D ":"", num);
        break;
    case 8:
        tmpStr.Format("%s%lo", bMitBasis ? "O ":"", num);
        break;
    case 2:
    { // '{}' muß wegen 'int i' sein
        unsigned long unum = (unsigned long) num;
        tmpStr = bMitBasis ? "B ":"";
        char tmpbuf[40];
        int i = 31;
        tmpbuf[32]='\0';
        while (unum && i + 1)
        {
            tmpbuf[i--] = (int)(unum%2) + '0';
            unum /= 2;
        }
        if (i == 31)
            tmpbuf[i--] = '0';
        tmpStr += tmpbuf + i + 1;
        break;
    }
    default:
        tmpStr.Format("Basis %d nicht unterstützt", base);
    }
    return tmpStr;
}

```

Listing: Aufbereitung der Ausgabezeichenfolge je nach Zahlenformat

Nun verbinden Sie die BN_CLICKED-Nachricht der Zifferschnittflächen mit der OnNumber-Funktion. Es werden alle Zifferschnittflächen mit derselben Funktion verbunden werden. Das hat den Vorteil, daß der Programmcode übersichtlich bleibt.

```

BEGIN_MESSAGE_MAP(CHEXEDlg, CDialog)
//{{AFX_MSG_MAP(CHEXEDlg)
//...
ON_BN_CLICKED(IDD_BUTTON_0, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_1, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_2, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_3, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_4, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_5, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_6, OnNumber)

```

```

ON_BN_CLICKED(IDD_BUTTON_7, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_8, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_9, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_A, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_B, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_C, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_D, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_E, OnNumber)
ON_BN_CLICKED(IDD_BUTTON_F, OnNumber)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

```

Listing: Ziffernschaltflächen mit der Funktion verknüpfen

Für die Operatoren-Schaltflächen, die beide Operatoren miteinander verbinden, wird die BN_CLICKED-Nachricht mit der OnBinaryOperator-Funktion verbunden. Die beiden Operatoren-Schaltflächen, die sich nur auf das Ergebnis bzw. nur auf einen Operator beziehen, werden mit der OnUnaryOperator-Funktion verbunden.

```

BEGIN_MESSAGE_MAP(CHEXEDlg, CDialog)
//{{AFX_MSG_MAP(CHEXEDlg)
//...
ON_BN_CLICKED(IDD_BUTTON_PLUS, OnBinaryOperator)
ON_BN_CLICKED(IDD_BUTTON_MINUS, OnBinaryOperator)
ON_BN_CLICKED(IDD_BUTTON_TIMES, OnBinaryOperator)
ON_BN_CLICKED(IDD_BUTTON_DIV, OnBinaryOperator)
ON_BN_CLICKED(IDD_BUTTON_MOD, OnBinaryOperator)
ON_BN_CLICKED(IDD_BUTTON_AND, OnBinaryOperator)
ON_BN_CLICKED(IDD_BUTTON_OR, OnBinaryOperator)
ON_BN_CLICKED(IDD_BUTTON_EXOR, OnBinaryOperator)
ON_BN_CLICKED(IDD_BUTTON_SHIFLEFT, OnBinaryOperator)
ON_BN_CLICKED(IDD_BUTTON_SHIFRIGHT, OnBinaryOperator)
ON_BN_CLICKED(IDD_BUTTON_ONESCOMP, OnUnaryOperator)
ON_BN_CLICKED(IDD_BUTTON_CHANGESIGN, OnUnaryOperator)
//...
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

```

Listing: Operatorschaltflächen mit den Funktionen verbinden

Die übrigen Schaltflächen – =, BS, CL und HILFE – bekommen je eine eigene Funktion zugeordnet.

```

BEGIN_MESSAGE_MAP(CHEXEDlg, CDialog)
//{{AFX_MSG_MAP(CHEXEDlg)

```

```
//...
ON_BN_CLICKED(IDD_BUTTON_EQUAL, OnEqual)
ON_BN_CLICKED(IDD_BUTTON_CL, OnClear)
ON_BN_CLICKED(IDD_BUTTON_BS, OnBackspace)
ON_BN_CLICKED(IDD_HILFE, OnHilfe)
//...
}}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Listing: Übrige Schaltflächen mit den Funktionen verbinden

Jetzt fehlen nur noch einige Funktionen, um den Taschenrechner zum Leben zu erwecken. Zur Steuerung der Eingabe über die Bildlaufleiste stellt die Klasse *CWnd* die Nachricht *WM_VSCROLL* zur Verfügung. Diese Nachricht ruft die *OnVScroll*-Funktion auf

```
CWnd::OnVScroll
afx_msg void OnVScroll(UINT nSBCode, UINT nPos,
                      CScrollBar* pScrollBar);
```

Mit Hilfe des Klassen-Assistenten wird diese Nachricht mit der Funktion *OnVScroll* der Klasse *CHEXEDlg* verbunden. Diese Nachricht wird jedesmal gesendet, wenn der Bediener mit der Maus auf die Bildlaufleiste klickt. Diese Funktion hat unter anderem einen Parameter, mit dem die Laufrichtung und Schrittgröße bestimmt werden kann. Damit läßt sich die Eingabe eines Operators gut steuern. Es wird auch hier daraufhin überprüft, ob der Wertebereich verlassen wurde, und das Ergebnis wird ggf. korrigiert.

```
void CHEXEDlg::OnVScroll(UINT nSBCode, UINT nPos,
                        CScrollBar* pScrollBar)
{
    switch (nSBCode)
    {
        case SB_LINEDOWN : --areg;
                           break;
        case SB_PAGEDOWN : areg -= nbase;
                           break;
        case SB_LINEUP : ++areg;
                           break;
        case SB_PAGEUP : areg += nbase;
                           break;
        case SB_THUMBPOSITION :
        case SB_THUMBTRACK : areg = 255-nPos;
                           break;
```

```

}
// Positionierung prüfen
if (areg < 0)
    areg = 0;
if (areg > 255)
    areg = 255;
// Anzeige aktualisieren
UpdateCalculatorDisplay();
// Basisklasse aufrufen
CDialog::OnVScroll(nSBCode, nPos, pScrollBar);
}

```

Listing: Auswertung der Nachrichten der Bildlaufleiste

Für die Auswahl eines alten Ergebnisses durch den Bediener aus dem Kombinationsfeld steht die Nachricht CBN_SELCHANGE des *Kombinationsfeld*-Fensters zur Verfügung. Verbinden Sie diese Nachricht auch mit Hilfe des Klassen-Assistenten mit der Funktion `OnOldResultSelected`. Diese Nachricht wird jedesmal gesendet, wenn ein neues Listenfeld ausgewählt wird. In der Funktion wird dann der Datenbereich des aktiven Feldes ausgelesen und in das Register übernommen. Diese Daten werden bei jedem neuen Eintrag mit-gefüllt (siehe Funktion `OnEqual`). Anschließend wird der Eingabe-Fokus auf die Schaltfläche mit dem Gleichheitszeichen gesetzt und die Anzeige des Taschenrechners aktualisiert.

```

void CHEXEDlg::OnOldResultSelected()
{
    areg = m_cbListe.GetItemData(m_cbListe.GetCurSel());
    new_number = TRUE;
    GetDlgItem(IDD_BUTTON_EQUAL)->SetFocus();
    UpdateCalculatorDisplay();
}

```

Listing: Auswertung der Änderungen im Kombinationsfeld

Für das Umschalten zwischen den Zahlenbereichen wird eine Registerkarte verwendet. Diese sendet bei jeder Änderung der aktiven Karte eine TCN_SELCHANGE-Nachricht an das Dialogfenster. Verbinden Sie diese mit der Funktion `OnSelchangeTabAusgabe`. In der Funktion selbst wird die aktive Karte ermittelt und der Zahlenbereich entsprechend geändert. Zum Schluß wird die Anzeige wieder aktualisiert.

```

void CHEXEDlg::OnSelchangeTabAusgabe(NMHDR* pNMHDR, LRESULT*
pResult)
{
    switch (m_TabAnzeige.GetCurSel())

```

```

{
    case 0 : nbase = 2;
            break;
    case 1 : nbase = 8;
            break;
    case 2 : nbase = 10;
            break;
    case 3 : nbase = 16;
            break;
}
new_number = TRUE;
UpdateCalculatorDisplay();
*pResult = 0;
}

```

Listing: Auswertung der Registerkarte für die Zahlenbereiche

Nun fehlt nur noch eine Funktion, um den Taschenrechner zu kompletieren. Eine der Aufgaben besteht darin, die Anzeige bei negativem Ergebnis in einer anderen Farbe darzustellen. Dazu dient uns die Nachricht `WM_CTLCOLOR` der Klasse `CWnd`, die die Funktion `OnCtlColor` aufruft.

```

class Cwnd:
    afx_msg HBRUSH OnCtlColor(CDC* pDC, CWnd* pWnd,
                              UINT nCtlColor);

```



In früheren Versionen gab es nur eine `WM_CTLCOLOR`-Nachricht mit verschiedenen Unter-codes. In Win32 wurde diese durch die `WM_CTLCOLORxxx` abgelöst. So gibt es jetzt zum Beispiel eine `WM_CTLCOLOREDIT`-Nachricht speziell für Eingabefelder und eine `WM_CTLCOLORBTN`-Nachricht für Schaltflächen. Da die MFC intern alles auf die `WM_CTLCOLOR`-Nachricht zurückführt, bleiben wir in unserem Beispiel auch bei dieser Nachricht.

Leiten Sie die Funktion `OnCtlColor` für die Klasse `CHEXEDlg` ab. In der Funktion wird dann das Ausgabefeld (Eingabefeld) aufgrund seines Symbolnamens herausgefiltert, und die Zeichenfarbe für Vorder- und Hintergrund wird geändert. Auch die Pinselfarbe muß geändert werden, damit das Ausgabefeld keinen Rand behält.

```

HBRUSH CHEXEDlg::OnCtlColor(CDC* pDC, CWnd* pWnd, UINT nCtlColor)
{
    HBRUSH hbr = CDialog::OnCtlColor(pDC, pWnd, nCtlColor);
    // Attribute des Gerätekontexts ändern,
    // wenn richtiges Fenster dran ist
    if (pWnd->GetDlgCtrlID() == IDD_OUTPUT_ERG)

```



```

{
    // weisser Hintergrund
    pDC->SetBkColor(RGB(255,255,255));
    // pos. Ergebnis blau
    if(areg >= 0)
        pDC->SetTextColor(RGB(0,0,255));
    // neg. Ergebnis rot
    else
        pDC->SetTextColor(RGB(255, 0, 0));
    // Pinsel muß auch auf weiss geändert werden
    return (HBRUSH)GetStockObject(WHITE_BRUSH);
}
return hbr;
}

```

Listing: Ändern der Farbe für das Ausgabefeld

Ausprogrammieren der Funktionen

Jetzt geht es daran, die Rechen- und Hilfsfunktionen zu verwirklichen. Die Erläuterung diesbezüglich soll an dieser Stelle eingespart werden, denn wie man die vier Grundrechenarten programmiert, sollte wohl bekannt sein. Außerdem befindet sich der Programmcode auf der beiliegenden CD.

Auf eine Besonderheit soll trotzdem hingewiesen werden: In der Funktion `OnOldResultSelected` wird auf den Datenbereich des Kombinationsfeldes zugegriffen. Das Füllen dieses Datenbereiches geschieht in der Funktion `OnEqual`, nachdem das Ergebnis ermittelt wurde; dazu wird `SetItemData` der Klasse `CComboBox` verwendet.

```

void CHEXEDlg::OnEqual()
{
    //...
    ///---Kombinationsfeld aktualisieren---
    m_cbListe.EnableWindow(TRUE);
    m_cbListe.DeleteString(MAXRESULT - 1);
    m_cbListe.InsertString(0, GetBasedString(areg, nbase, TRUE));
    m_cbListe.SetItemData(0, areg);
    m_cbListe.SetCurSel(0);
    UpdateCalculatorDisplay();
}

```

Listing: Füllen des Kombinationsfeldes mit dem Ergebnis

Erweitern von OnInitDialog

Bevor das fertige Projekt komplett übersetzt werden kann, müssen noch einige Initialisierungen vorgenommen werden. Was eignet sich besser dazu, als die Funktion `OnInitDialog` der Klasse `Cdialog`? Konkret ist dort noch folgendes zu machen:

1. Initialisieren der Bildlaufleiste für das Raster 0...255
2. Anlegen eines Font-Objektes für die Ausgabe des Ergebnisses mit einer **fetten** Schriftart
3. Die Registerkarten zur Registersteuerung hinzufügen und initialisieren
4. Taschenrechneranzeige aktualisieren

Im Programmcode sieht das dann so aus:

```
BOOL CDialog::OnInitDialog()
{
    //...
    m_scrZeichenWert.SetScrollRange(0, 255, FALSE);
    LOGFONT LogFont;
    m_edErg.GetFont()->GetLogFont(&LogFont);
    LogFont.lfWeight = FW_BOLD;
    m_ergFont.CreateFontIndirect(&LogFont);
    m_edErg.SetFont(&m_ergFont);
    TC_ITEM TabCtrlItem;
    TabCtrlItem.mask = TCIF_TEXT;
    TabCtrlItem.pszText = "&binär (2)";
    m_TabAnzeige.InsertItem( 0, &TabCtrlItem );
    TabCtrlItem.pszText = "&oktal (8)";
    m_TabAnzeige.InsertItem( 1, &TabCtrlItem );
    TabCtrlItem.pszText = "&dezimal (10)";
    m_TabAnzeige.InsertItem( 2, &TabCtrlItem );
    TabCtrlItem.pszText = "&hexadezimal (16)";
    m_TabAnzeige.InsertItem( 3, &TabCtrlItem );
    m_TabAnzeige.SetCurSel(2);
    UpdateCalculatorDisplay();
    return TRUE;
}
```

Listing: Weitere Initialisierungen der Steuerungen**Einfacher Hilfedialog**

Um den doch aus Anwendersicht einfachen Taschenrechner mit einer kurzen Hilfe zu versehen, muß man nicht gleich das Windows-Hilfesystem verwenden. Es genügt ein kleiner Dialog, der die Funktionalität mit

wenigen Worten erläutert. Dafür müssen Sie nicht einmal eine eigene Klasse von *CDialog* ableiten. Erzeugen Sie eine Dialogvorlage (IDD_HILFE) zur Anzeige des kurzen Hilfetextes für HEXE.



Abbildung 12.17: Hilfedialog ohne eigene Klasse

Erzeugen Sie wie in Abschnitt 12.3 beschrieben einen Dialog ohne eigene Klasse für die Methode *OnHilfe* die von der Schaltfläche HILFE aufgerufen wird.

Damit ist die erforderliche Funktionalität fertig. Übersetzen Sie das Programm. Achten Sie auf Fehler und Warnungen beim Kompilieren und Linken der Anwendung. Sie können das Programm noch verbessern und erweitern. Passen Sie den *Info über ...*-Dialog an; Oder Sie setzen eine Abfrage vor dem Beenden des Dialoges ein. Verbessern Sie die Hilfe, wenn Ihnen der Dialog nicht genügt. Dies sind nur einige Anregungen. Es ist nun an Ihnen, sich an dem Projekt zu üben.

12.6 Sonstiges

Es gibt noch eine ganze Reihe allgemeiner Funktionen und Methoden, die für die Dialogfeld-Programmierung nützlich sind. In diesem Abschnitt sollen zwei davon kurz vorgestellt werden, weil man sie ab und zu gut gebrauchen kann.

Weitere Informationen finden sich in der elektronischen SDK-Dokumentation unter *Dialog boxes*.

Es gibt mindestens drei Techniken, die unter die Rubrik »Fehlerbehandlung« fallen, die Methoden *FlashWindow* und *MessageBox* der Klasse *CWnd* und die globale Funktion *MessageBeep*.



```
void ::MessageBeep(UINT uType);
class CWnd:
    BOOL FlashWindow(BOOL bInvert);
class CWnd:
    int MessageBox(const char FAR* lpText,
                  const char FAR* lpCaption = NULL,
                  UINT nType = MB_OK);
```

MessageBox wurde bereits in Kapitel 5 erklärt. Mit Hilfe von *MessageBeep* können Sie den eingebauten Lautsprecher veranlassen, einen Ton von sich zu geben. Dabei kann der Parameter *uType*, wie bei *Message-Box* auch, verschiedene Werte haben. Während 0xFFFFFFFF (-1L) nur einen Standard-Piepton zu erzeugen vermag, kann mit Hilfe der symbolischen Konstanten *MB_ICONASTERISK*, *MB_ICONEXCLAMATION*, *MB_ICONHAND*, *MB_ICONQUESTION* und *MB_OK* ein entsprechender Signalton erzeugt werden. Falls eine Soundkarte und/oder ein geeigneter Treiber installiert sind, erzeugen diese Werte die jeweils zugehörigen Systemklänge, andernfalls bleibt es beim Piepsen.

Mit *FlashWindow* werden der Rahmen und die Titelleiste des Fensters zwischen inaktiv und aktiv umgeschaltet, je nachdem, welchen Zustand das Fenster bisher hatte. Ein erneuter Aufruf sorgt dafür, daß der alte Zustand wiederhergestellt wird. Ruft man diese Methode mit jeweils einer kleinen Verzögerung mehrmals kurz hintereinander auf, so entsteht der Eindruck eines blinkenden Rahmens. Mit dieser Methode machen Programme, die den Eingabefokus nicht haben, auf sich aufmerksam, wenn bei ihnen ein wichtiges Ereignis aufgetreten ist. Typischerweise wird *FlashWindow* zusammen mit einem Timer verwendet, um ein periodisches Blinken zu realisieren, bis das Fenster den Eingabefokus bekommen hat. Diese Eigenschaft wirkt auch, wenn sich das Fenster minimiert und in der Task-Leiste befindet.



Ein Beispiel für die Anwendung dieser Anregungen finden Sie auf der beiliegenden CD unter BEISPIELE / WEITERE / BLINKDIALOG.

12.7 Nicht-modale Dialoge

12.7.1 Unterschiede zu modalen Dialogfeldern

Bisher wurden nur modale Dialogfelder betrachtet. Diese haben die Eigenschaft, die Aktivitäten des Hauptfensters (genauer: des Elternfensters) so lange zu blockieren, bis der Dialog beendet ist; erst dann reagiert das Hauptfenster wieder auf Tastatur- oder Mauseingaben.

Nun ist es allerdings nicht unbedingt nötig, ein Dialogfeld *modal* zu programmieren – auch wenn es unter Verwendung der Klasse *CDialog* und ihrer Methode *DoModal* sehr bequem ist. Zumeist entspricht es zwar dem lo-

gischen Ablauf der Anwendung, einen Dialog zu beenden, bevor der nächste aufgerufen wird; sehr häufig ist es aber auch sinnvoll, Dialoge *nicht-modal* zu programmieren. In diesem Fall wird der Dialog parallel zum Hauptfenster geöffnet, und der Benutzer kann wahlweise mit dem Dialog oder dem Hauptfenster kommunizieren.

Viele Windows-Anwendungen arbeiten mit *nicht-modalen* Dialogen, beispielsweise das *Eigenschaften*-Fenster der Visual C++-Entwicklungsumgebung, aber auch Visual Basic oder Windows Write. Meist werden die nicht-modalen Dialoge automatisch mit dem Hauptfenster zusammen erzeugt und bleiben solange aktiv, bis der Benutzer sie explizit schließt. Dazu gibt es gewöhnlich im Menü einen Punkt, mit dem solche globalen *nicht-modalen* Dialoge gestartet oder beendet werden können. Nicht-modale Dialoge werden beispielsweise verwendet, um Steuerungs- oder Funktionsleisten darzustellen oder einen *Suchen- und Ersetzen*-Dialog zu realisieren.

Nicht-modale Dialoge verhalten sich in vielerlei Hinsicht wie normale Fenster, die Steuerungen enthalten. Bei der Programmierung haben nicht-modale Dialoge aber eine Reihe von Vorteilen. Erstens zeigen sie bei Tastatureingaben dasselbe Verhalten wie modale Dialogfelder, d.h. korrekte Interpretation der [TAB]-Taste, der Cursortasten und Verwaltung von Gruppen. Zweitens liegen sie gegenüber ihrem Elternfenster automatisch im Vordergrund und können von diesem nicht verdeckt werden.

Mit diesen Eigenschaften dienen *nicht-modale* Dialoge meist dazu, das Menü des Programms auf die eine oder andere Art zu erweitern, um auf wichtige Funktionen direkten Zugriff zu haben. Ein solches Menüfenster hat darüber hinaus den Vorteil, verschiebbar zu sein, und kann bei Bedarf vom Benutzer geschlossen werden. Der Trend zu nicht-modalen Dialogen, wie er in vielen neueren Applikationen zu beobachten ist, kommt also nicht von ungefähr. Wie sich zeigen wird, sind nicht-modale Dialoge fast genauso einfach zu programmieren wie modale. So gibt es eigentlich keinen Grund, sie im Bedarfsfall nicht auch in eigenen Anwendungen einzusetzen.

12.7.2 Entwurf der Dialogvorlage

Nicht-modale Dialogfelder werden genauso entworfen wie modale: Man legt Sie mit dem Dialog-Editor an, fügt Steuerungen ein und legt deren Eigenschaften fest. Bei der Dialogvorlage gibt es keinen Unterschied zwischen modalen und nicht-modalen Dialogen. Die einzige Besonderheit ist die Eigenschaft *Sichtbar* auf der Eigenschaftskarte WEITERE FORMATE. Wenn die Eigenschaft *WS_VISIBLE* gesetzt ist, erscheint das Dialogfeld selbständig. Ansonsten müssen Sie den Dialog erst mit der Methode *ShowWindow* sichtbar machen. Betrachtet man einige Programme mit nicht-modalen Dialogen, erkennt man jedoch einige Unterschiede zu modalen Dialogen, die sich im Laufe der Zeit eingebürgert haben.

Zunächst besitzt ein nicht-modaler Dialog oft eine dünnere Titelleiste als ein modaler. Üblicherweise wird nicht die Standardtitelleiste verwendet, sondern nur eine etwa 2/3 so große, was allerdings auf das Verhalten keinerlei Einfluß hat. Um dies im Dialog-Editor einzustellen, ist im *Eigenschaften*-Fenster der Dialogvorlage unter ERWEITERTE FORMATE das Kontrollkästchen *Tool-Fenster* zu aktivieren.



Abbildung 12.18: Der nicht-modale Ampeldialog mit schmaler Titelleiste

Es ist aber genauso gut möglich, einen Dialog ganz ohne Titelleiste und Schaltflächen zu gestalten. Ein solcher Dialog könnte z.B. zur Statusanzeige o.ä. verwendet werden.

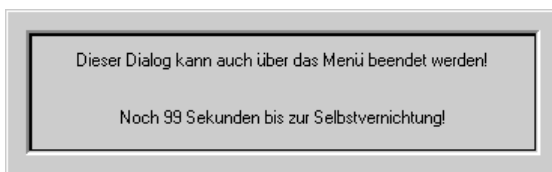


Abbildung 12.19: Der nicht-modale Dialog ohne Titelleiste und Schaltflächen

Optional kann ein nicht-modaler Dialog *Minimieren*- und *Maximieren*-Schaltflächen besitzen. Mit Hilfe dieser Schaltflächen könnte der Anwender nichtbenötigte Dialoge temporär als Symbol ablegen und bei Bedarf wieder aktivieren. Der Nachteil an der Sache ist, daß die Dialogboxklasse sich selbst um das Zeichnen des Icons kümmern muß, denn ein nicht-modaler Dialog besitzt kein Icon, das automatisch angezeigt würde. Statt dessen bleibt das Icon in »Symboldarstellung« leer, und es wird lediglich das Windows-Symbol verwendet. Um das Icon zu füllen, könnte beispielsweise die *OnPaint*-Methode der Dialogklasse überlagert werden (wie bei dialogbasierenden Anwendungen).

12.7.3 Programmierung

Im allgemeinen werden nicht-modale Dialoge nicht über Rahmenreservierung (Stack-Rahmen) angelegt. Hier bietet die MFC keine allzu komfortablen Methoden an. Sie sind selbst für das Anlegen, Anzeigen und Zerstö-

ren der Instanz verantwortlich. Das Objekt wird meist auf dem Heap angelegt. Sobald sich der Dialog auf dem Bildschirm befindet, kehrt die Steuerung zurück. Nachrichten und Eingaben des Anwenders gelangen nur durch eigene Botschaften nach außen.

Da die Methode *DoModal*, wie der Name bereits sagt, immer einen modalen Dialog hervorbringt, ist hier ein anderes Verfahren gefragt. Um zu einem nicht-modalen Dialog zu kommen, leiten Sie mit dem Klassen-Assistenten, wie auch bei den meisten modalen Dialogen, eine eigene Klasse von *CDialog* ab. Fügen Sie einen eigenen öffentlichen Konstruktor hinzu, mit dem dann die Klasse aufgerufen werden kann. Sie können bei dieser Gelegenheit dem Objekt auch einen Parameter übergeben.

```
class CDlgAmpel : public CDialog
{
// Konstruktion
public:
    CDlgAmpel(CWnd* pParent = NULL);    // Standardkonstruktor
    CDlgAmpel(int start);               // eigener Konstruktor
    //...
};
```

Listing: Eigener Konstruktor für nicht-modale Dialoge

Durch die Methode *CDialog::Create* des Dialogobjektes wird die Dialogvorlage geladen und dargestellt.

```
class Cdialog:
    BOOL Create(UINT nIDTemplate, CWnd* pParentWnd = NULL);
```

Dem Parameter *nIDTemplate* wird der Bezeichner der Dialogfeldvorlage übergeben und *pParentWnd* ein Zeiger auf das Elternfenster. Falls *pParentWnd* NULL ist oder nicht übergeben wird, ist das Hauptfenster der Anwendung das Elternfenster. Die Methode *CDialog::Create* kann sowohl im Konstruktor als auch danach aufgerufen werden.

Nun steht noch die Frage offen, wie man einen nicht-modalen Dialog wieder loswird. Das Beenden ist genauso leicht wie das Aufrufen, funktioniert aber etwas anders, als es bei modalen Dialogen der Fall ist. Anstelle der Methode *EndDialog* muß die Methode *DestroyWindow* der Klasse *CWnd* aufgerufen werden, um den Dialog zu zerstören:

```
class Cwnd:
    virtual BOOL DestroyWindow();
```

Dieser Aufruf kann sowohl im Rahmenprogramm erfolgen, etwa wenn das Programm beendet wird, oder auch im Dialog selbst, wenn er über entsprechende Schaltflächen verfügt.



```
void CMainWindow::OnClose()
{
    //...
    pModelessDialog->DestroyWindow();
    //...
}
```

Zu beachten ist allerdings, daß die Instanz der Klasse dann immer noch existiert. Da sie mit *new* angelegt wurde, muß sie mit *delete* auch wieder abgebaut werden. Dazu wird eine weitere Methode der Klasse *CWnd* überlagert:

```
void CDlgAmpel::PostNcDestroy()
{
    CDialog::PostNcDestroy();
    delete(this);
}
```

Listing: Auflösen eines nicht-modalen Dialogs

Der Dialog entfernt sich selbst aus dem Speicher. Daß sich eine Instanz so auflöst, ist bei anderen Klassen unzulässig, doch hier ist es ausnahmsweise möglich und auch nötig.

Die Klasse *CDialog* der MFC wertet standardmäßig u.a. auch die **ESC** Taste aus. Der nicht-modale Dialog wird dann durch die Methode *EndDialog* vom Bildschirm entfernt, bleibt aber weiterhin aktiv. Um dies zu verhindern, muß die Methode *OnCancel* überlagert werden. In dieser Funktion wird dann einfach nur *DestroyWindow* aufgerufen, um den Dialog zu beenden, oder die Funktion bleibt leer, wenn auf die Taste nicht reagiert werden soll.



```
void CModlesDialog::OnCancel()
{
    DestroyWindow();
}
```

12.7.4 Anwendungsbeispiele

Im Prinzip gib es in Bezug auf den Beendigungsvorgang nur zwei verschiedene Programmvarianten eines nicht-modalen Dialoges: Zum einen kümmert sich der Dialog selbst um seine Lebensdauer, oder er wird von außen gesteuert. Beide beruhen auf der gleichen bereits beschriebenen Funktionalität. Der programmtechnische Unterschied besteht darin, daß im Fall

der Steuerung von außen eine dauerhafte Variable existieren muß, die den Zeiger auf das Dialogobjekt aufnehmen kann. Im folgenden werden zwei kleine Beispiele die Varianten veranschaulichen und dabei die oben beschriebene Programmier Technik nicht-modaler Dialoge noch einmal veranschaulichen.

Der Ampeldialog

Zunächst soll ein nicht-modaler Dialog realisiert werden, der nicht von außen gesteuert werden kann. Damit ist er also selbst für seine Lebensdauer verantwortlich. Das Hauptprogramm hat nach dem Erzeugen keinen Einfluß mehr auf den Dialog. Es können mehrere Dialoge gleichzeitig dargestellt werden. Dazu folgendes Beispiel:

Es geht darum, einen nicht-modalen Dialog zu realisieren, der eine symbolisierte Ampel und darunter eine Schaltfläche zum zyklischen Umschalten zwischen den Zuständen ROT, ROT-GELB, GRÜN und GELB enthält. Außerdem ist ein Instanzenzähler vorgesehen, der protokolliert, wie oft der Dialog aufgerufen wurde. In Abbildung 12.18 ist der Dialog für die Rotphase dargestellt. Die vier Ampelphasen werden durch unterschiedliche Bitmaps dargestellt, die in ein Bild-Feld geladen werden. Für jedes Bitmap wird eine eigene Member-Variable angelegt.

Zur Anzeige des entsprechenden Ampel-Bitmaps implementieren Sie eine Methode *ShowState*, die je nach aktuellem Status die dazugehörige Ampel sichtbar macht. Der Status wird durch die Variable `m_nState` repräsentiert.

```
void CDlgAmpel::ShowState()
{
    m_Ampel1.ShowWindow(SW_HIDE);
    m_Ampel2.ShowWindow(SW_HIDE);
    m_Ampel3.ShowWindow(SW_HIDE);
    m_Ampel4.ShowWindow(SW_HIDE);
    switch (m_nState)
    {
        case 0 :
            m_Ampel1.ShowWindow(SW_SHOW);
            break;
        case 1 :
            m_Ampel2.ShowWindow(SW_SHOW);
            break;
        case 2 :
            m_Ampel3.ShowWindow(SW_SHOW);
            break;
```

```

        case 3 :
            m_Ampel4.ShowWindow(SW_SHOW);
            break;
        }
    }

```

Listing: Ampelsteuerung

Der Status wird über die Schaltflächen gesteuert. Verbinden Sie mit Hilfe des Klassen-Assistenten die Schaltfläche  mit der Funktion `OnWechsel`. Dies verändert die Variable `m_nStatus` und ruft ihrerseits die Funktion `ShowState` auf.

```

void CDlgAmpel::OnWechsel()
{
    ++m_nState %= 4;
    ShowState();
}

```

Listing: Funktion der Schaltfläche

Zur Realisierung des Instanzenzählers muß der Dialog eine statische Variable erhalten. Diese wird mit dem Wert »0« Modulglobal initialisiert.

```

private:
    static int m_nAnzahl;
// Initialisierung
int CDlgAmpel::m_nAnzahl = 0;

```

Als nächstes kann der Konstruktor überlagert werden. Er erhält nur einen Parameter mit dem Startwert der Ampelschaltung. Innerhalb des Konstruktors wird die Methode `CDialog::Create` aufgerufen, um den Dialog sofort anzuzeigen. Außerdem ist der Instanzenzähler `m_nAnzahl` zu erhöhen.

```

CDlgAmpel::CDlgAmpel(int start)
{
    state = start;
    m_nAnzahl++;
    if (CDialog::Create(IDD))
    {
        ShowState();
        CString strAnzahl;
        strAnzahl.Format("%d. Instanz", m_nAnzahl);
        m_stInstanz.SetWindowText(strAnzahl);
    }
}

```

Listing: Überlagerter Konstruktor

Zuletzt muß noch das Aufräumen erledigt werden. Dazu überlagern Sie die bereits erwähnte Methode *PostNcDestroy* von *CWnd*. Hier löst sich die Instanz selbst auf, nicht ohne vorher den *static*-Zähler zu erniedrigen.

```
void CDlgAmpel::PostNcDestroy()
{
    m_nAnzahl--;
    CDialog::PostNcDestroy();
    delete (this);
}
```

Wenn Sie das Programm jetzt übersetzen und starten, wird der Dialog zwar funktionieren, jedoch erniedrigt sich der Instanzenzähler nicht wie vorgesehen nach dem Beenden. Die Methode *PostNcDestroy* wird also nicht wie eigentlich erwartet aufgerufen, das heißt, unser Dialog lebt noch weiter. Erst beim Beenden des Hauptprogramms werden alle Fenster zerstört. Dafür sorgt Windows, denn es sendet an aller Kindfenster des Hauptfensters die *PostNcDestroy*-Nachricht. Die Ursache für unser Phänomen liegt in dem Verhalten der Basisklasse *CWnd*. Die *ON_CLOSE*-Nachricht schließt zwar das Fenster, zerstört es aber nicht. Deshalb muß bei nicht-modalen Dialogen, die auf eine *ON_CLOSE*-Nachricht reagieren sollen die entsprechende Methode *OnClose* auch noch überlagert werden. In Ihr müssen Sie dann nur noch *DestroyWindow* aufrufen, um den Dialog wie eigentlich beabsichtigt auch aus dem Speicher zu entfernen.

```
void CDlgAmpel::OnClose()
{
    DestroyWindow();
}
```



Nachdem Sie diese Funktion eingefügt haben, erfüllt der Dialog (fast) alle Erwartungen.

Außensteuerung

In dem zweiten Beispiel geht es darum, einen nicht-modalen Dialog zu realisieren, der auch von außen beendet werden kann. Außerdem soll es maximal einen Dialog auf dem Bildschirm geben. Dazu ist es erforderlich, daß das Hauptprogramm bzw. das Hauptfenster einen Zeiger auf das Objekt hat; im folgenden ein Beispiel hierzu:

In einem nicht-modalen Dialog ohne Titelleiste und Schaltfläche wird nur ein Text ausgegeben. Außerdem wird ein Timer installiert, der nach Ablauf einer bestimmten Zeit nicht nur den Dialog beendet, sondern das Hauptfenster schließt und damit das Programm sogar beendet, wenn er nicht vorher vom Programm selbst wieder geschlossen wird. In Abbildung 12.19 ist die Dialogvorlage zu sehen.

Die Realisierung ist relativ einfach. Dieses mal überlagern Sie zuerst den Konstruktor. Er hat zwei Parameter: Erstens die Zeitdauer in Sekunden bis zum Beenden der Anwendung und zweitens einen Zeiger auf das Elternfenster wie auch im Standardkonstruktor. Wie bereits bekannt, wird dann die Methode *Create* von *CDialog* aufgerufen.

```
CDlgZaehler::CDlgZaehler(int dauer, CWnd* pParent)
{
    m_nZaehler = dauer;
    m_pParent = pParent;
    if (CDialog::Create(IDD))
        m_nTimerID = SetTimer(1, 1000, NULL);
}
```

Listing: Überlagerter Konstruktor mit Timer

Außerdem wird ein Timer mit einer Periode von einer Sekunde installiert. Dieser zählt dann die verbleibende Zeit bis zum Schließen des Hauptfensters. Außerdem wird jedesmal der Text im Dialog geändert und ein akustisches Signal gegeben. Wenn der Zähler bei Null angekommen ist, bevor der Dialog von außen beendet wurde, vollendet er seinen Auftrag und beendet nicht nur sich selbst.

```
void CDlgZaehler::OnTimer(UINT nIDEvent)
{
    if (nIDEvent == m_nTimerID)
    {
        CString strText;
        strText.Format("Noch %d Sekunden bis zur  
Selbstvernichtung.", m_nZaehler);
        m_stZaehler.SetWindowText(strText);
        m_nZaehler--;
        MessageBeep(0xFFFFFFFF);
        if (m_nZaehler == 0)
        {
            KillTimer(m_nTimerID);
            m_pParent->PostMessage(WM_CLOSE);
            DestroyWindow();
            return;
        }
    }
    CDialog::OnTimer(nIDEvent);
}
```

Listing: Timergesteuertes Programmbeenden

Jetzt fehlt nur noch das Überlagern der Methode *PostNcDestroy* für das Auflösen der Instanz, und der Dialog ist fertig. Für dieses Beispiel muß nun aber auch noch einiges im Hauptprogramm erweitert werden, damit die Steuerung von außen auch funktioniert. Zuerst bekommt das Hauptfenster eine Variable `CDlgZaehler* pDlgZaehler` als Zeiger auf den nicht-modalen Dialog. Dieser wird mit `NULL` im Konstruktor initialisiert.

```
CMainFrame::CMainFrame()
{
    pDlgZaehler = NULL;
}
```

Listing: Konstruktor des Hauptfensters

Im Menü des Hauptfensters legen Sie zwei Menüpunkte an: Einen zur Anzeige und den zweiten zum Beenden des Dialoges. Außerdem sollen die Menüpunkte über die `ON_UPDATE_COMMAND_UI`-Nachricht gesteuert werden, um einen Mehrfachaufruf zu vermeiden. Das Ganze sieht im Programmcode dann etwa so aus:

```
// Menüsteuerung für Zähler an
void CMainFrame::OnUpdateBeispieleZaehleran(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(pDlgZaehler == NULL);
}

// Zählerdialog anlegen
void CMainFrame::OnBeispieleZaehleran()
{
    pDlgZaehler = new CDlgZaehler(15, this);
}

// Menüsteuerung für Zähler aus
void CMainFrame::OnUpdateBeispieleZaehleraus(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(pDlgZaehler != NULL);
}

// Zählerdialog beenden
void CMainFrame::OnBeispieleZaehleraus()
{
    pDlgZaehler->DestroyWindow();
    pDlgZaehler = NULL;
}
```

Listing: Dialogaufruf und Menüsteuerung

Damit ist auch das zweite Beispielprogramm fertig. Übersetzen und testen Sie es. Um auch dieses Programm sicherer zu machen, sollten Sie auch noch die Methode *OnClose* überlagern. Zwar besitzt der nicht-modale

Dialog kein Systemmenü, über das er eine WM_CLOSE-Nachricht erhalten könnte, wenn der Dialog aber den Eingabefocus besitzt, und der Bediener die Tastenkombination `Alt+F4` drückt, hat das dieselbe Wirkung. Auch die ESC-Taste müssen Sie noch abfangen, indem Sie die Methode *OnCancel* überlagern und in dieser nichts tun, auch nicht die Basisklasse aufrufen.

Wie Sie sehen, ist doch einiges zu beachten bei der Entwicklung nicht-modaler Dialoge. Doch das sollte niemanden davon abhalten, diese in seinen Programmen zu verwenden. Denn es gibt oft Anforderungen, die nur durch den Einsatz nicht-modaler Dialoge sinnvoll realisiert werden können. Das komplette Beispielprogramm mit beiden Dialogvarianten finden Sie auf der Begleit-CD unter BEISPIELE \ WEITERE \ NICHTMODAL.



Dokument-Ansichten- Architektur

13

Kapitelübersicht

13.1	Dokument-Ansicht-Paradigma der MFC	496
13.2	Die Klasse CDocument	497
13.2.1	Basis aller MFC-Dokumente	497
13.2.2	Zugriff auf die Ansichtsklassen	498
13.2.3	Verwaltung der Daten	499
13.2.4	Die Registrierung der Dokumente	505
13.3	Die Klasse CView	507
13.3.1	Basis alle MFC-Ansichten	507
13.3.2	Zugriff auf das Dokument	508
13.3.3	Initialisierung	509
13.3.4	Ausgabemöglichkeiten	510
13.3.5	Nachrichtenbearbeitung mit Klassen-Assistenten	516
13.4	Die Klasse CDocTemplate	517
13.5	SDI-basierende Anwendung - ein Beispiel	519
13.5.1	Besonderheiten einer SDI-Anwendung	519
13.5.2	Aufgabe	520
13.5.3	Bedienung	521
13.5.4	Implementierung	522
13.5.5	Projektanlage mit dem MFC-Anwendungs-Assistenten	523
13.5.6	Anpassung der Ressourcen	524
13.5.7	Zusätzlicher Programmcode	528
13.6	Die Klasse CFormView	549
13.6.1	Verwendung	549
13.6.2	Formularvorlagen formatierter Ansichten	549
13.6.3	Der Einsatz in der Anwendung	550
13.7	Die Klasse CSplitterWnd	551
13.8	MDI-basierende Anwendung - ein Beispiel	555
13.8.1	Besonderheiten einer MDI-Anwendung	555
13.8.2	Aufgabe	557
13.8.3	Implementation	558
13.8.4	Projektanlage mit dem MFC-Anwendungs-Assistenten	558
13.8.5	Erweitern und Anpassen der Ressourcen	561
13.8.6	Zusätzlicher Programmcode	564

13.1 Dokument-Ansicht-Paradigma der MFC

Wenn Sie bis hierher gekommen sind, beherrschen Sie bereits die Grundlagen der MFC-Programmierung und den Umgang mit der Entwicklungsumgebung. Das Wissen reicht aus, um eine MFC-basierende Anwendung zu schreiben. Doch die Beschreibung der bedeutendsten Möglichkeiten steht noch aus.

Nachdem Sie bereits eine dialogbasierende Anwendung komplett mit Anwendungs-Assistenten verwirklicht haben und mit dem Klassen-Assistenten die Nachrichten und Klassen verwaltet und angelegt haben, geht es in diesem Teil des Buches um Dokument-Ansicht-basierende Anwendungen und die erweiterten Möglichkeiten der Assistenten.

Die nachfolgenden Abschnitte geben eine Einführung in das Arbeiten mit dem *Dokument-Ansicht-Paradigma* der MFC mit der Trennung von Daten und Ansichten. Dabei geht es um das Anlegen und die Verwendung von Dokument- und Ansicht-Klasse. Das Dokument ist Datenspeicher der Anwendung. Es sorgt für die Verwaltung der Datenstrukturen und stellt Methoden zur Verfügung, mit denen andere Klassen auf die Daten zugreifen können. Die Ansicht dagegen ist das visuelle Interface der Daten. Sie ist in der Lage, die Daten des Dokuments auf dem Bildschirm anzuzeigen, und reagiert auf Nachrichten, die durch Aktionen des Bedieners ausgelöst werden.



Ein stark vereinfachtes, aber anschauliches Modell ist die Darstellung von Datum und Uhrzeit. Die Daten sind das aktuelle Datum und die aktuelle Uhrzeit und stehen somit für das Dokument. Nun gibt es verschiedene Möglichkeiten der Darstellung: Die Uhrzeit als digitale Zahlen und die Monate ebenfalls mit Zahlen; oder die analoge Darstellung der Uhrzeit und die Monate mit ihrem Namen.

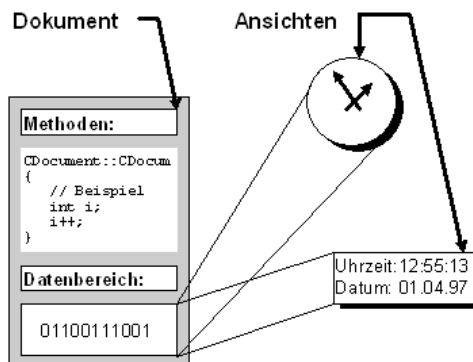


Abbildung 13.1: Beispiel »Dokument und Ansichten«

MFC-Programme verwenden in der Regel ein Softwaremodell, das die Daten des Programms von deren Ansicht trennt. In diesem Modell liest und schreibt ein Dokumentenobjekt die Daten. Das Dokument kann auch eine Schnittstelle zu Daten in einer Datenbank herstellen. Ein gesondertes Ansichtenobjekt verwaltet die Darstellung der Daten in einem Fenster und ermöglicht deren Bearbeitung durch den Benutzer. Die Ansicht erhält die anzuzeigenden Daten vom Dokument und teilt dem Dokument wiederum mit, wenn Änderungen an den Daten erfolgen.

Die Trennung von Dokument und Ansicht ist nicht in jedem Fall sinnvoll und leicht zu umgehen. Dafür sind Beispiele am Anfang dieses Buches ausgeführt. Für einfache Anwendungen ist es nicht unbedingt erforderlich, dem Paradigma zu folgen, denn ein gewisser Initialaufwand ist nötig, um mit dem Paradigma zu arbeiten. Den unterschiedlichen Anforderung wird auch der Anwendungs-Assistent jetzt gerecht. Mit der aktuellen Version unterstützt er auch Anwendungen, die nicht dem Paradigma folgen. Es gibt jedoch auch viele Gründe, diesem Modell normalerweise zu folgen. Einer der wichtigsten tritt dann ein, wenn Sie für ein Dokument mehrere Ansichten benötigen. Das Dokument-Ansicht-Modell ermöglicht ein separates Ansichtenobjekt für verschiedene Ansichten der Daten. Das Dokument übernimmt auch die Aufgabe der Aktualisierung aller Ansichten, wenn sich die Daten ändern.

Die Dokument-Ansichten-Architektur der MFC unterstützt mehrere Ansichten, mehrerer Dokumenttypen und unterteilte Fenster. Die ganze Entwicklungsumgebung, das durch den Anwendungs-Assistenten generierte Programmgerüst und die Basisklassen der MFC unterstützen die Dokument-Ansicht-Architektur hervorragend. Auch der Klassen-Assistent hilft Ihnen bei der Erstellung von entsprechenden Anwendungen. Die Klassen *CDocument* und *CView* zeigen, wie man alles integriert, um daraus ein leistungsfähiges Programm zu machen. Aus Platzgründen können auch in diesem Teil des Buches nicht alle Aspekte vollständig behandelt werden. Der gebotene Einblick reicht aber für den Einstieg aus und ermöglicht es Ihnen, mit Hilfe der Dokumentationen und der Online-Hilfe weiterzukommen.

13.2 Die Klasse *CDocument*

13.2.1 Basis aller MFC-Dokumente

Die Klasse *CDocument* ist die Basisklasse alle benutzerdefinierten Dokumentenklassen der MFC. Außer einigen wenigen Methoden der Basisklasse *CCmdTarget* enthält sie auch alle Methoden der Klasse *CObject* und viele eigene Methoden zur Behandlung von Daten. Dabei ist ein Teil für die Navigation und Kommunikation mit den Ansichtsklassen verantwor-

lich. Ein anderer Teil der Methoden dient der Daten-Manipulation bzw. -Verarbeitung, wie z. B. dem Öffnen und Speichern von Dateien. Ein weiterer Teil ermöglicht das Versenden der Daten des Dokumentes via Mail.

Darüber hinaus besitzen die MFC spezielle Klassen – von *CDocument* abgeleitet –, mit denen über OLE ein Datenaustausch mit anderen Dokumenten oder Anwendungen erfolgen kann.

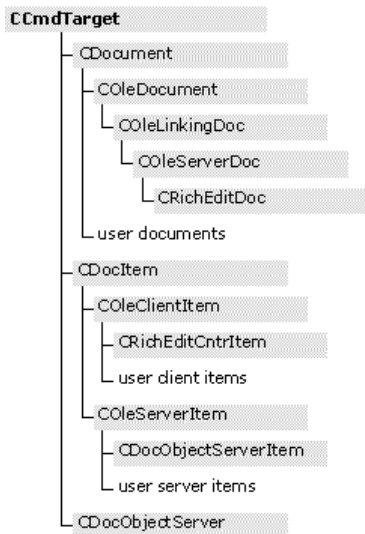


Abbildung 13.2: Klassenhierarchie von *CDocument*

Die Klasse *CDocument* ist also von Hause aus mit den wichtigsten Funktionen ausgestattet. Für Sie bleibt dann nur noch die Aufgabe, Ihre Datenstruktur zu implementieren und die entsprechenden Zugriffsmethoden zu programmieren. Im Folgenden werden die wichtigsten Funktionen erläutert.

13.2.2 Zugriff auf die Ansichtsklassen

Wie bereits erwähnt, müssen Dokument und Ansicht miteinander in Verbindung treten können. Das ist z. B. notwendig, wenn das Dokument Daten verändert und der Ansicht oder den Ansichten mitteilen will, daß ihre Bildschirmausschnitte neu gezeichnet werden müssen.

Zur Lösung dieses Problems wurde die Methode *UpdateAllViews* geschaffen, mit der es einem Dokument möglich ist, alle zugeordneten Ansichten von einer Datenänderung in Kenntnis zu setzen. *UpdateAllViews* macht nichts weiter, als für jede Ansicht die Methode *CView::OnUpdate* aufzuru-

fen. In der Standard-Implementierung führt *OnUpdate* dabei einen Aufruf von *Invalidate(TRUE)* durch, um die komplette Ansicht inklusive ihres Hintergrunds für ungültig zu erklären.

Da es nicht bei jeder Datenänderung sinnvoll ist, den kompletten Bildschirm inklusive Hintergrund neu zu zeichnen, besitzt *UpdateAllViews* einige Parameter, mit denen der zu restaurierende Bereich differenziert werden kann:

```
class Cdocument:
    void UpdateAllViews(CView *pSender, LPARAM lHint = 0L,
                      COBJECT *pHint = NULL);
```

Mit *pSender* benennt *UpdateAllViews* die Ansicht, die für die Änderung der Daten verantwortlich war, und erlaubt es *OnUpdate*, diese Ansicht von der Änderung auszunehmen (für den Fall, daß sie das schon selbst erledigt hat). Zusätzlich können in den Parametern *lHint* und *pHint* Hinweise übergeben werden, die in kodierter Form übermitteln, welcher Bereich des Bildschirms aktualisiert werden muß. Auf welche Weise die Kodierung dieser Hinweise erfolgt, bleibt allerdings den beiden Methoden selbst überlassen.

Mit den Methoden *AddView*, *RemoveView*, *GetFirstViewPosition* und *GetNextView* können Ansichtsklassen dem Dokument hinzugefügt bzw. bereits vorhandene durchsucht werden.

```
class Cdocument:
    void AddView(CView* pView);
    void RemoveView(CView* pView);
    virtual POSITION GetFirstViewPosition( ) const;
    virtual CView* GetNextView(POSITION& rPosition) const;
```

13.2.3 Verwaltung der Daten

Serialisieren der Daten

Eine der Hauptaufgaben der Dokumentenklasse besteht in der Verwaltung der Daten. Da Daten meist nicht nur einmalig zur Laufzeit des Programmes wichtig sind, sondern eine längere Lebensdauer als das Programm haben, müssen sie langfristiger gespeichert und wieder eingelesen werden können. Auch die Ergebnisse der Bearbeitung durch das Programm müssen gesichert, vielleicht anderen Programmen zur Verfügung gestellt werden können. Heute sind dazu Datenträger wie Diskette oder Festplatte vorhanden, auf denen die Daten in Dateien geschrieben werden.

Die MFC stellt mehrere Möglichkeiten zur Ein- und Ausgabe von Daten zur Verfügung. Eine davon ist die datenbankorientierte Ein- und Ausgabe, die z.B. ODBC-Datenquellen verwendet. Eine einfachere und doch ge-

niale Methode ist die Serialisation der Daten und Objekte, d.h. schreiben und lesen in einem meist unstrukturierten Datenstrom. Dabei werden nicht nur die Daten, sondern auch die Objektstruktur der Klassen, die die Daten aufnehmen, gespeichert. Dieses Verfahren wird von der MFC voll unterstützt und auch in der Klasse *CDocument* eingesetzt. Die Fähigkeit, sich in einen Datenstrom zu »verwandeln«, erben alle Klassen, die von *CObject* abgeleitet sind. Darüber hinaus ist diese Funktionalität auch Klassen wie *CString* implementiert, *auch wenn sie nicht von CObject abgeleitet sind*.

```
CObject::Serialize
virtual void Serialize(CArchive& ar);
    throw(CMemoryException);
    throw(CArchiveException);
    throw( CFileException );
```

Für *CString* und andere Klassen sind entsprechende Operatoren (>> und <<) vorgesehen, die das Objekt in ein Archiv schreiben und auch aus einem solchen lesen können.

```
friend CArchive& operator <<(CArchive& ar, const CString& string);
    throw(CArchiveException);
friend CArchive& operator >>( CArchive& ar, CString& string);
    throw(CArchiveException);
friend CDumpContext& operator <<(CDumpContext& dc,
                                const CString& string);
```

Serialize wird von der Klasse *CDocument* immer dann automatisch aufgerufen, wenn das aktuelle Dokument gespeichert oder ein neues geladen werden soll – also als Reaktion auf die Menüpunkte ÖFFNEN, SPEICHERN oder SPEICHERN UNTER des DATEI-Menüs.

Interessanterweise erfolgt sowohl das Laden als auch das Speichern in einer Methode, denn *Serialize* bekommt ein Objekt der Klasse *CArchive* übergeben, das für beide Operationen verwendet werden kann. Ob im konkreten Fall gerade die eine oder andere Aktion erwünscht ist, kann mit Hilfe der Methode *IsStoring* herausgefunden werden: ist deren Rückgabewert *TRUE*, soll gespeichert, andernfalls geladen werden.

```
void CEvolutionDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        // ZU ERLEDIGEN: Hier Code zum Speichern einfügen
    }
}
```

```
else
{
    // ZU ERLEDIGEN: Hier Code zum Laden einfügen
}
}
```

Listing: Methode zum Lesen und Schreiben von Objekten in eine Datei

Ohne auf weitere Details einzugehen, sei an dieser Stelle die Bedeutung der Klasse *CArchiv* kurz zusammengefaßt. Ein Objekt der Klasse *CArchiv* ist ein Hilfsmittel, um Objekte persistent zu machen, d.h., sie auf einem externen Medium zu speichern bzw. von diesem zu lesen. Abstrakt kann man sich ein solches Archiv als Datei vorstellen, die bereits im Lese- oder Schreibmodus geöffnet wurde und in der Lage ist, Daten in binärer Form mit dem Objekt auszutauschen.

```
class Carchive:
    UINT Read(void *lpBuf, UINT nMax);
    void Write(const void *lpBuf, UINT nMax);
```

Normalerweise wird man für verschiedene Objektarten, die in einem Archiv gespeichert oder aus einem solchen geladen werden sollen, jeweils die Operatoren << und >> überlagern. So hat man später eine bequeme Möglichkeit, auf das Archiv zuzugreifen.

Read liest aus der geöffneten Datei, die dem Archiv zugeordnet ist, *nMax* Bytes und legt sie an der Stelle ab, auf die der Zeiger *lpBuf* zeigt. Der Rückgabewert der Funktion ist die Anzahl der Bytes, die tatsächlich gelesen werden konnten, also möglicherweise kleiner als *nMax*, wenn die Datei nicht (mehr) lang genug war. Mit *Write* hingegen werden die ersten *nMax* Bytes aus dem Puffer *lpBuf* in die Datei geschrieben, die dem Archiv zugeordnet ist. In beiden Fällen werden die Daten in keiner Weise interpretiert, sondern vollkommen unverändert weitergegeben.

Voraussetzung für dieses Verfahren zur Datenein- und -ausgabe ist die Implementierung der Klassen mit den Makros `DECLARE_SERIAL` und `IMPLEMENT_SERIAL`.

Dokumente öffnen und schließen

Um Dateien zu öffnen oder zu schließen, stehen der Klasse *CDocument* die Methoden *OnOpenDocument*, *OnNewDocument*, *OnSaveDocument* und *OnCloseDocument* zur Verfügung.

```
class Cdocument:
    virtual BOOL OnOpenDocument(const char* pszPathName);
    virtual BOOL OnNewDocument();
    virtual void OnCloseDocument();
    virtual BOOL OnSaveDocument(const char* pszPathName);
```

Die folgende Tabelle zeigt, wann die Methoden jeweils aufgerufen werden und welche Aufgabe ihnen zukommt:

Methode	Wird aufgerufen, wenn...	Sollte erledigen...
<i>OnOpenDocument</i>	mit DATEI/ÖFFNEN ein bestehendes Dokument geöffnet wird.	Initialisierung
<i>OnNewDocument</i>	mit DATEI/NEU ein Dokument neu angelegt wird, außerdem beim Starten der Anwendung.	Initialisierung
<i>OnCloseDocument</i>	mit DATEI/SCHLIESSEN ein Dokument geschlossen wird (nur MDI).	Endebehandlung
<i>OnSaveDocument</i>	mit DATEI/SPEICHERN [UNTER] ein Dokument gespeichert wird.	Braucht sich in der Regel weder um Initialisierung noch um Endebehandlung zu kümmern.

Tabelle 13.1: Einige virtuelle Funktionen von CDocument

Falls irgendeine dieser Methoden überlagert wird, ist es erforderlich, darin die gleichnamige Methode der Basisklasse aufzurufen. Diese realisiert neben der speziellen Initialisierung und Endebehandlung der Datenstruktur auch die eigentliche Funktion dieser Methode: das Laden, Neuanlegen oder Schließen eines Dokuments. Normalerweise benutzen diese Funktionen zur Auswahl der Datei durch den Bediener die Standarddialoge für DATEI ÖFFNEN und DATEI SPEICHERN (siehe Kapitel 5). Sie können aber auch eigene Abfragedialoge einsetzen.

Weitere Methoden der Klasse *CDocument* unterstützen die komfortable Anwendung des Verfahrens zum Lesen und Schreiben der Daten. Mit *SetModifiedFlag*, *IsModified* und *SaveModified* können Informationen über den Zustand der Daten seit dem letzten Sichern verwaltet werden.

```
class Cdocument:
    void SetModifiedFlag(BOOL bModified = TRUE);
    BOOL IsModified();
    virtual BOOL SaveModified();
```

Mit *SetModifiedFlag* können Sie festhalten, ob sich der Inhalt der Daten-Member des Dokumentes geändert hat oder nicht. *IsModified* gibt Auskunft über den aktuellen Status, und *SaveModified* wird vom Programmgerüst vor dem Schließen des Dokumentes aufgerufen. Diese Funktionen werten den Status aus und bringen gegebenenfalls eine Benutzerabfrage zum Sichern der Daten.

Datenaktualisierung

Für die Aktualisierung der Daten innerhalb des Dokuments steht die virtuelle Methode *DeleteContents* zur Verfügung. Sie wird immer dann aufgerufen, wenn die Daten des Dokuments zu initialisieren sind. Da dies in den MFC nicht nur der Fall ist, wenn ein Dokument geöffnet oder neu angelegt wird, sondern auch, wenn es geschlossen wird (bei SDI-Anwendungen insbesondere, wenn das Programm beendet wird), kann man möglicherweise einen Teil der Initialisierungs- und Endbehandlung hier ausführen lassen.

Im Einzelfall muß man jedoch prüfen, ob es möglich (und sinnvoll) ist, Initialisierung und Endbehandlung auf diese Weise zu vermischen. Um dies zu entscheiden, kann man anhand von Abbildung 13.3 erkennen, wann *DeleteContents* in den einzelnen Methoden tatsächlich aufgerufen wird.

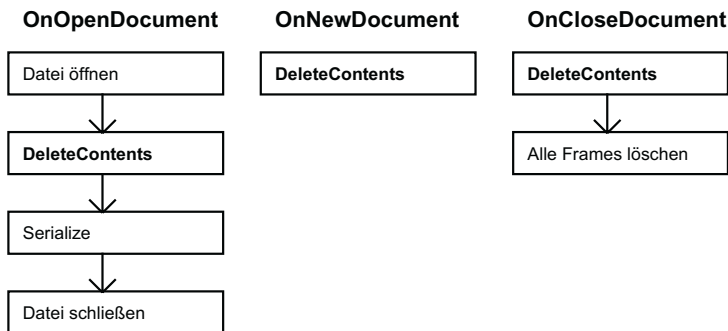


Abbildung 13.3: Aufruf von *DeleteContents*

Mit Hilfe der Variablen *m_bAutoDelete* können Sie entscheiden, ob das Dokument aufgelöst werden soll oder nicht, wenn die letzte zugeordnete Ansicht geschlossen wird.



Versenden der Dokumente

Mit der Klasse *CDocument* bietet die MFC auch eine Unterstützung der MAPI-Schnittstelle (Microsoft Messaging Application Programming Interface). MAPI beinhaltet eine Sammlung von Funktionen, die das Erstellen,

Bearbeiten, Übertragen und Speichern von E-Mail-Nachrichten aus einer Anwendung heraus ermöglichen. *CDocument* unterstützt nur einen bescheidenen Teil dieser Funktionalität. Mit ihr ist es möglich, den Inhalt eines Dokumentes mit Hilfe des installierten Mailsystems als Anlage zu versenden. Dazu steht die Methode *OnFileSendMail* zur Verfügung.

```
class CDocument : public CCmdTarget
{
    DECLARE_DYNAMIC(CDocument)
    // ...
protected:
    // mail enabling
    afx_msg void OnFileSendMail();
    afx_msg void OnUpdateFileSendMail(CCmdUI* pCmdUI);
    DECLARE_MESSAGE_MAP()
};
```

Sie wird über die Nachricht `ID_FILE_SEND_MAIL` aktiviert und versendet den Dokumenteninhalt. Zum Bereitstellen der Daten wird intern der Befehl *Speichern* aufgerufen, der dann über die Serialisation einen Datenstrom erzeugt. Die so entstandene Nachricht muß nur noch adressiert werden, dann ist sie bereits versandfertig. Voraussetzung ist die Installation einer Mail-Unterstützung auf dem System. Die MFC benutzt für die Implementierung die Programmbibliothek `MAPI32.DLL`.

```
void CDocument::OnFileSendMail()
{
    // ...
    _AFX_MAIL_STATE* pMailState = _afxMailState;
    if (pMailState->m_hInstMail == NULL)
        pMailState->m_hInstMail = ::LoadLibraryA("MAPI32.DLL");
    // ...
}
```

Wenn auf dem PC ein Mailsystem nicht zur Verfügung steht, wird das über die Methode *OnUpdateFileSendMail* ermittelt. Die MFC entfernt dann die Menüpunkte und deaktiviert den Zugriff.

Um eine Anwendung mit der Mail-Funktionalität auszurüsten, haben Sie zwei Möglichkeiten. Die erste Möglichkeit ist, daß Sie das Menü um einen Punkt mit dem Symbolnamen `ID_FILE_SEND_MAIL` erweitern. Anschließend fügen Sie folgende Zeilen in die Nachrichtentabelle des betreffenden Dokumentes ein:


```
ON_COMMAND(ID_FILE_SEND_MAIL, OnFileSendMail)
ON_UPDATE_COMMAND_UI(ID_FILE_SEND_MAIL, OnUpdateFileSendMail)
```

Achten Sie darauf, daß dies außerhalb der Kommentare // {{AFX_MSG_MAP... erfolgt, da sonst der Klassen-Assistent beim nächsten Aufruf seinen Dienst für diese Klasse versagt. Da diese Member-Funktionen in der Basisklasse implementiert sind, müssen Sie die Anwendung nur noch neu erstellen.

Die zweite Möglichkeit ist, diese Aufgabe dem Anwendungs-Assistenten zu überlassen, indem Sie im Schritt 4 die MAPI-Unterstützung (Messaging API) aktivieren. Der Assistent fügt den Menüpunkt NACHRICHT SENDEN ... unter DATEI ein. Auch die Nachrichtentabelle wird dann automatisch ergänzt.

Damit der Empfänger mit den so übermittelten Daten auch etwas anfangen kann, muß er die gleiche Anwendung oder mindestens eine Anwendung mit gleichem Dokumenten-Typ installiert haben. Wichtig ist auch, daß diese Dokumente registriert sind, sonst können sie nicht automatisch geöffnet werden (siehe auch nächstes Kapitel).

Mehr Informationen zur MAPI, vor allem zur Nutzung des vollen Leistungsumfangs, finden Sie bei der SDK-Unterstützung der Win32-API-Funktionen.



13.2.4 Die Registrierung der Dokumente

Das Programmgerüst ermöglicht es, die Dokumente der Anwendung in die Registrierdatenbank von Windows einzutragen. In Schritt 4 der Anlage des Projektes mit dem Anwendungs-Assistenten kann über die Schaltfläche WEITERE OPTIONEN ... folgender Dialog aufgerufen werden:

In der Anwendung wird damit nicht nur der Dialog *Datei öffnen* mit der Dokument-Endung (hier .CD) vorbelegt, sondern durch das Aufnehmen der entsprechenden Methoden in *InitInstance* der Eintrag der Endung in die Registrierung beim ersten Programmstart vorgenommen. Das erledigt die Methode *RegisterShellFileTypes*.

```
CWinApp::RegisterShellFileTypes
void RegisterShellFileTypes(BOOL bCompat = FALSE);
```

Wichtig ist, daß zuvor alle zu registrierenden Dokumente in der Dokumentvorlage angelegt und in die Liste der Dokumentvorlagen durch *Add-DocTemplate* eingetragen wurden (siehe Kapitel »Dokumentvorlage«). Alle Dokumente der Vorlage werden dann registriert.

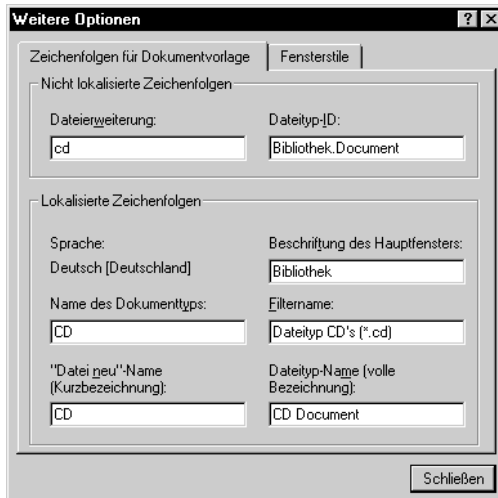


Abbildung 13.4: Erweiterte Dokument-Optionen



```

BOOL CMyApp::InitInstance()
{
    //...
    AddDocTemplate(pDocTemplate);
    // Haupt-MDI-Rahmenfenster erzeugen
    CMainFrame* pMainFrame = new CMainFrame;
    if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
        return FALSE;
    m_pMainWnd = pMainFrame;
    // Öffnen per DragDrop aktivieren
    m_pMainWnd->DragAcceptFiles();
    // DDE-Execute-Open aktivieren
    EnableShellOpen();
    RegisterShellFileTypes(TRUE);
    //...
}

```

Die Dateierweiterung `.CD` wird unter `ARBEITSPLATZ\HKEY_CLASS\ROOT` eingetragen und mit `BIBLIOTHEK.DOCUMENT` verknüpft. Der erzeugte Eintrag der Dokumentenklasse in der Registerdatenbank sieht dann so aus:

Dadurch stehen insgesamt drei Möglichkeiten zum Öffnen eines Dokuments mit der registrierten Erweiterung zur Verfügung:

1. Starten der Anwendung durch Doppelklick auf das Dokumentsymbol und anschließendes Öffnen des gewählten Dokuments (eingebetteter Start) über DDE.

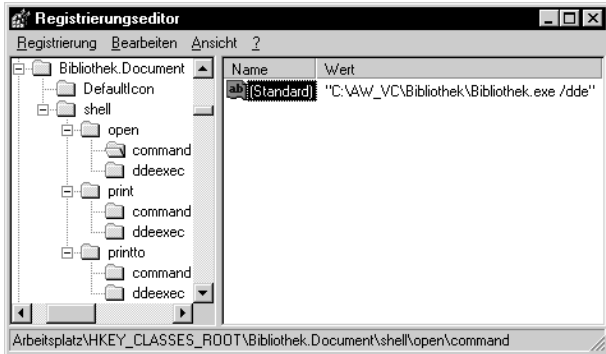


Abbildung 13.5: Eintrag in der Registerdatenbank

2. Ziehen des Dokumentsymbols mit Hilfe der Maus auf das Hauptfenster der laufenden Anwendung (*Drag and Drop*).
3. Öffnen eines Dokuments durch Kommandozeilen-Argument der Anwendung (ausgewertet durch *ParseCommandLine* und *ProcessShell-Command*).

13.3 Die Klasse *CView*

13.3.1 Basis alle MFC-Ansichten

Nachdem im vorangegangenen Teil der Dokument-Bestandteil der Dokument-Ansicht-Architektur erklärt wurde, geht es in diesem Abschnitt um die Bedeutung und die Konstruktion der Ansichten. Die Basis aller Ansichtsklassen der MFC ist *CView*. Im Gegensatz zum Dokument ist die Ansichtsklasse von *CWnd* angeleitet. Sie erbt damit bereits alle wichtigen Fensterfunktionen, die man unter Windows braucht. Darüber hinaus stellen die MFC eine ganze Reihe spezialisierter Versionen dieser Basisklasse zur Verfügung.

Dabei gibt es spezielle Klassen zur reinen Textanzeige, zur Listenausgabe bis hin zur formatierten Form mit einer Dialogvorlage und Datenbankansichten. Die Klasse *CView* besitzt bereits viele wichtige Methoden, die sich auf die Darstellung und Aktualisierung der Daten auf Bildschirm und Drucker beziehen.

Beim Anlegen eines dokumentbasierenden Programmgerüsts mit dem Anwendungs-Assistenten wird neben dem Dokument auch eine Ansicht (View) angelegt. Diese hat die Aufgabe, die Daten des Programms visuell darzustellen und auf Benutzereingaben zu reagieren. Die Ansicht ist somit die Schnittstelle zwischen dem Benutzer des Programms und den internen Daten.

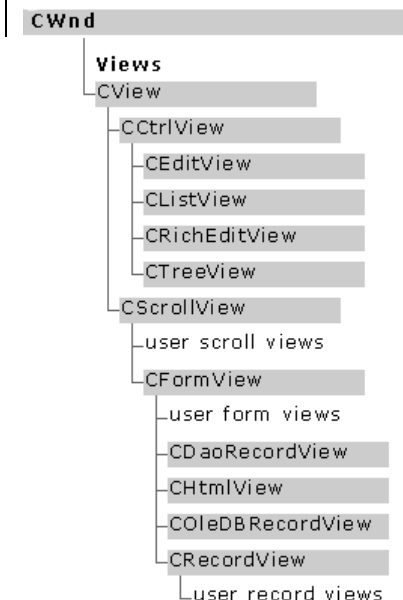


Abbildung 13.6: Klassenhierarchie von CView

13.3.2 Zugriff auf das Dokument

Im Gegensatz zum Dokument, welches mehrere Ansichten mit Daten versorgen kann, ist die Ansichtsklasse nur in der Lage, mit einem Dokument zusammenzuarbeiten. Die Klasse *CView* besitzt die Methode *GetDocument*, um sich einen Zeiger auf das zugehörige Dokument zu beschaffen.

```
class CView:
    CDocument* GetDocument() const;
```

Durch die entsprechende Typkonvertierung kann auf alle öffentlichen Methoden und Variablen des Dokuments zugegriffen werden.

Der Anwendungs-Assistent überlagert diese Methode der Klasse *CView* bereits in der abgeleiteten Ansichtsklasse. Es steht eine Debug- und eine Release-Version dieser Funktion zur Verfügung. In der Release-Version ist sie als Inline implementiert.

```
#ifndef _DEBUG // Testversion in EvolutionView.cpp
inline CEvolutionDoc* CEvolutionView::GetDocument()
{ return (CEvolutionDoc*)m_pDocument; }
#endif
```

Listing: Inline-Version der Methode GetDocument

```
#ifdef _DEBUG
CEvolutionDoc* CEvolutionView::GetDocument() // Die endgültige
// (nicht zur Fehlersuche kompilierte) Version ist Inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CEvolutionDoc)));
    return (CEvolutionDoc*)m_pDocument;
}
#endif // _DEBUG
```

Listing: Debug-Version der Methode GetDocument

Die wichtigste Schnittstelle zwischen Dokument und Ansicht stellen die Methoden *CDocument::UpdateAllViews* und *CView::OnUpdate* dar (siehe Kapitel 13.2.2).

```
class CView:
    virtual void OnUpdate(CView *pSender, LPARAM lHint = 0L,
                        COBJECT *pHint = NULL);
```

In der Methode *OnUpdate* sollten dann alle Aktualisierungsmaßnahmen erfolgen. Dabei können die übergebenen Parameter ausgewertet werden.

Folgende Abbildung illustriert noch einmal die Beziehung zwischen Dokument und Ansicht.

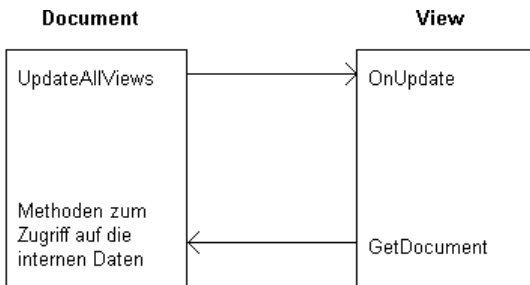


Abbildung 13.7: Die Beziehung zwischen Dokument und Ansicht

13.3.3 Initialisierung

Unmittelbar nach dem Anlegen des Ansicht-Objekts, aber noch vor dem ersten Anzeigen der Daten, wird die virtuelle Methode *OnInitialUpdate* vom Programmgerüst aufgerufen:

```
class CView:
    virtual void OnInitialUpdate();
```

Falls die abgeleitete Klasse spezielle Initialisierungen vornehmen will, sollten Sie *OnInitialUpdate* überlagern und ihre Initialisierungen in dieser Methode erledigen. Anders als im Konstruktor ist hier bereits ein Zugriff auf Ausgabefunktionen sowie Steuerungen möglich. Innerhalb der eigenen Funktion muß *OnInitialUpdate* Basisklasse aufgerufen werden. *CView::OnInitialUpdate* ruft nun ihrerseits die bereits beschriebene virtuelle Methode *OnUpdate* auf, die dann die eigentliche Anzeige aktualisieren sollte.

13.3.4 Ausgabemöglichkeiten

Die Methode *OnDraw*

Wenn es erforderlich ist, den Bildschirm neu zu zeichnen, wird die Methode *CView::OnDraw* aufgerufen. Sie hat damit für eine Ansichts-klasse in etwa die Bedeutung wie *OnPaint* für ein normales Fenster. Im Gegensatz zu *OnPaint* benötigt sie aber keinen Eintrag in der Nachrichtenschleife, sondern kann als virtuelle Funktion ohne weitere Maßnahmen in der abgeleiteten Klasse überlagert werden. Ein weiterer Unterschied zu *OnPaint* besteht darin, daß *OnDraw* einen Zeiger auf einen gültigen Device-Kontext – in Form eines Zeigers auf eine CDC-Klasse – bereits als Parameter übergeben bekommt und sich diesen nicht erst beschaffen muß (siehe Kapitel »*OnPaint/OnDraw*-Routinen für Textausgabe«):

```
class Cview:
    virtual void OnDraw(CDC *pDC);
```

Damit innerhalb von *OnDraw* auf die Daten des Dokuments zugegriffen werden kann, wird die Methode *GetDocument* benutzt. Sie liefert, wie bereits beschrieben, einen Zeiger auf das zugehörige Dokument, um dessen Methoden aufrufen zu können. Der Rest von *OnDraw* ist normale GDI-Programmierung unter den MFC und kann so ausgeführt werden, wie es auch in einer *OnPaint*-Funktion möglich wäre:

```
void CEvolutionView::OnDraw(CDC* pDC)
{
    CEvolutionDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    CRect rect;
    int    cx, cy, x, y;
    CPen   pen, *pOldPen;
    CBrush bluebrush, redbrush;

    GetClientRect(rect);
    cx = rect.right / CELLSIZE;
    cy = rect.bottom / CELLSIZE;
```

```

//Zuerst das Gitter zeichnen
pen.CreatePen(PS_SOLID, 1, RGB(0, 128, 0));
pOldPen = pDC->SelectObject(&pen);
for (x = 0; x <= CELLSIZE; ++x)
{
    pDC->MoveTo(x * cx, 0);
    pDC->LineTo(x * cx, CELLSIZE * cy);
}
for (y = 0; y <= CELLSIZE; ++y)
{
    pDC->MoveTo(0, y * cy);
    pDC->LineTo(CELLSIZE * cx, y * cy);
}
pDC->SelectObject(pOldPen);
//Nun die Zellen zeichnen
greenbrush.CreateSolidBrush(RGB(0, 255, 0));
redbrush.CreateSolidBrush(RGB(255, 0, 64));
for (y = 0; y < CELLSIZE; ++y)
{
    for (x = 0; x < CELLSIZE; ++x)
    {
        if (pDoc->IsAlive(x, y))
        {
            pDC->FillRect(CRect(x * cx + 1, y * cy + 1,
                (x + 1) * cx, (y + 1) * cy), &redbrush);
        }
        else
        {
            pDC->FillRect(CRect(x * cx + 1, y * cy + 1,
                (x + 1) * cx, (y + 1) * cy), &greenbrush);
        }
    }
}
}

```

Listing: Fensterausgabe mit der Methode »OnDraw«

Druckausgabe und Seitenansicht

Ein großer Vorteil für den Anwender von Windows ist die relativ einfache Handhabung von Druckern. Windows stellt eine ganze Reihe von Druckertreibern zur Verfügung, die dann für den entsprechenden Drucker installiert werden können. Sogar eine einheitliche Benutzeroberfläche für die Konfiguration stellt Windows zur Verfügung. Doch was bedeutete das

bisher für den Programmierer, dessen Anwendung unter Windows drucken soll? Wer es schon einmal versucht hat, weiß, daß das zu den schwierigsten Aufgaben gehört.

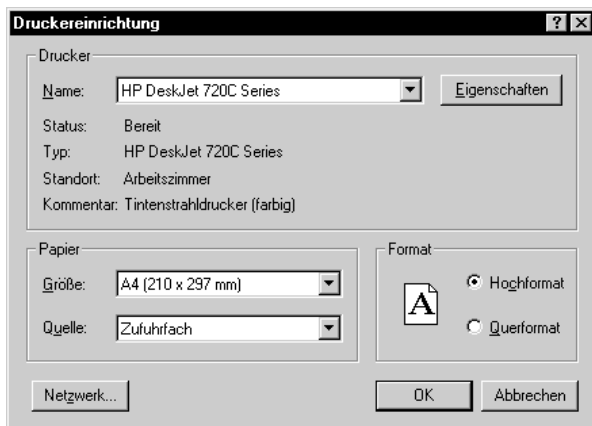


Abbildung 13.8: Dialog »Druckereinstellung«

Das MFC-Programmgerüst und Visual C++ erleichtern diese Arbeit beachtlich. Zum einen können die Standard-Dialoge (Abbildung 13.8) des Betriebssystems benutzt werden. Mit ihnen kann die Einstellung des aktiven Windows-Druckers verändert oder der zu druckende Bereich interaktiv bestimmt werden. In Abbildung 13.9 ist die Bedeutung von Seitenzahlen zu erkennen.

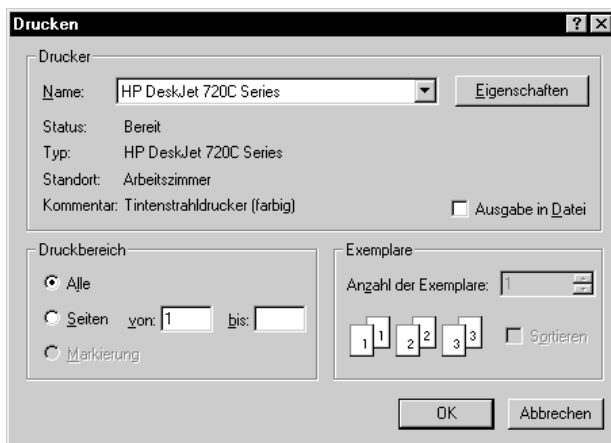


Abbildung 13.9: Standard-Dialog »Drucken«

Zum anderen enthalten die MFC eine Seitenansichtsfunktion ähnlich der von Winword oder Excel. Sie zeigt die Seiten in der Druckvorschau genauso an, wie sie auch beim Ausdruck mit dem aktuellen Drucker aussehen würden.

Dazu untersuchen die Funktionen der MFC jedes einzelne Zeichen und bestimmen Form, Größe und Position aufgrund des Drucker-Gerätekontextes. Die Schrift mag bei kleinen Größen etwas ungewohnt aussehen, doch die Zeichen stehen in der Druckvorschau in der Regel an der richtigen Stelle. Für die grafischen Elemente wird die Funktionalität der oben beschriebenen Methode *OnDraw* der Ansichts-Klasse zum Drucken benutzt (Abbildung 13.10).

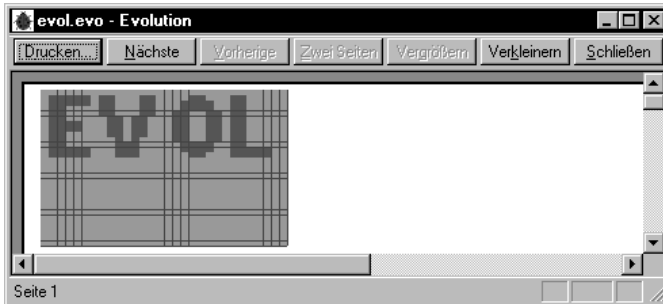


Abbildung 13.10: MFC-Druckvorschau (Seitenansicht)

Außerdem zeigt während des Druckens die Anwendung den Status des Druckvorgangs in einem kleinen Dialog an. Dieser ermöglicht auch den Abbruch des Druckauftrags.



Abbildung 13.11: Druckerstatus

Um diese Funktionalität in der Anwendung zur Verfügung zu haben, müssen Sie nur beim Anlegen des Programmgerüsts mit dem Anwendungs-Assistenten die notwendige Option einstellen. Dazu ist in Schritt 4 die Option *Drucken und Seitenansicht* zu aktivieren.

Diese beschriebene Funktionalität ist nur in dokumentbasierenden Anwendungen möglich. In der Dokument-Ansicht-Architektur ist der Ausdruck bzw. die Druckvorschau eine Sicht auf die Daten des Dokuments. Damit wird klar, daß die Druckfunktionen in der Klasse *CView* enthalten sind. Für die Druckerausgabe verwendet die MFC ein Gerätekontext-Objekt der Klasse *CDC*. Dieses wird vom Programmgerüst erzeugt und der *OnDraw*-Funktion der Klasse *CView* als Parameter übergeben. Dieses Gerätekontext-Objekt besitzt u.a. eine Methode *IsPrinting*, die es ermöglicht, festzustellen, ob man sich im Druck- oder Bildschirmmodus befindet.

```
CDC::IsPrinting
    BOOL IsPrinting() const;
```

So könnte ein Rumpf der überlagerten Funktion *OnDraw* erweitert werden, um für das Drucken eine andere Darstellung zu implementieren als zur Bildschirmausgabe. Diese Funktion wird für die Bildschirmausgabe von *OnPaint* und für die Druckerausgabe von *OnPrint* aufgerufen. Mit Hilfe der Klasse *CDC* sind Sie in der Lage, zwischen Bildschirm- und Druckerausgabe zu unterscheiden.



```
void CMYView::OnDraw(CDC* pDC)
{
    ASSERT_VALID(pDC);

    if (pDC->IsPrinting())
    {
        //...
        return;
    }
    //...
}
```

Wenn in der Anwendung ein Druckauftrag ausgelöst wird, startet das Programmgerüst automatisch die *CView*-Methode *OnPreparePrinting*.

```
CView::OnPreparePrinting
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
```

Dies geschieht noch vor der Anzeige des Dialogs *Drucken* (Abbildung 13.9). An dieser Stelle kann die minimale und maximale Seitenanzahl gesetzt werden, die dann im Dialog angezeigt und geändert werden kann.



```
BOOL CMYView::OnPreparePrinting(CPrintInfo* pInfo)
{
    pInfo->SetMinPage(1);
```

```

pInfo->SetMaxPage(5);
//...
return DoPreparePrinting(pInfo);
}

```

Wenn der Dialog *Drucken* geschlossen wird, erfolgt der Aufruf von *OnBeginPrinting*. Diese Funktion kann überlagert werden, um GDI-Objekte – z.B. eine Schriftart – für den gesamten Ausdruck zu erzeugen.

```

CView::OnBeginPrinting
virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);

```

Dann wird für jede Seite *OnPrint* aufgerufen, die, wie oben beschrieben, *OnDraw* der Ansichtsklasse ausführt. Hier können Änderungen oder Anpassungen an den GDI-Objekten vorgenommen werden, die dann für die jeweilige Seite gelten.

```

CDC::SetBkColor
virtual COLORREF SetBkColor(COLORREF crColor);
CDC::SetBkMode
int SetBkMode(int nBkMode);

```

Dabei kann Ihnen die bereits mehrfach erwähnte Klasse *CDC* gute Dienste leisten. Diese Klasse bietet Ihnen eine Vielzahl von Member-Funktionen, die Ihnen einen einfachen Umgang mit dem Device-Kontext ermöglichen. Mit ihr kann nicht nur die Hintergrundfarbe über *SetBkColor* geändert werden, auch der Hintergrundmodus kann mit *SetBkMode* auf Transparent umgeschaltet werden.

```

void CMyView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
    ASSERT_VALID(this);
    ASSERT_VALID(pDC);
    ASSERT(pInfo != NULL);
    //...
    pDC->SetBkMode(TRANSPARENT);
    //...
}

```

Als letztes, wenn der Druckauftrag erledigt ist, wird die Funktion *OnEndPrinting* aufgerufen. Diese Funktion muß überladen werden, um die GDI-Objekte, die in *OnBeginPrinting* angelegt wurden, wieder zu löschen.

```

CView::OnEndPrinting
virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);

```

13.3.5 Nachrichtenbearbeitung mit Klassen-Assistenten

Der Klassen-Assistent unterstützt auch die Nachrichtenbearbeitung der Klasse *CView*. Dabei spielt es keine Rolle, ob es sich um Nachrichten des Programmgerüsts bzw. der Basisklassen handelt, oder ob es sich um Menüpunkte, Nachrichten von Steuerungen, Symbolleisten und Zugriffstasten handelt. Wenn Sie ein Menü für Ihre Dokumentvorlage und damit auch der Ansichtsklasse hinzugefügt haben, können Sie die entsprechenden Kommandos mit dem Klassen-Assistenten bequem mit dazugehörigen Funktionen der Ansichtsklasse verbinden.

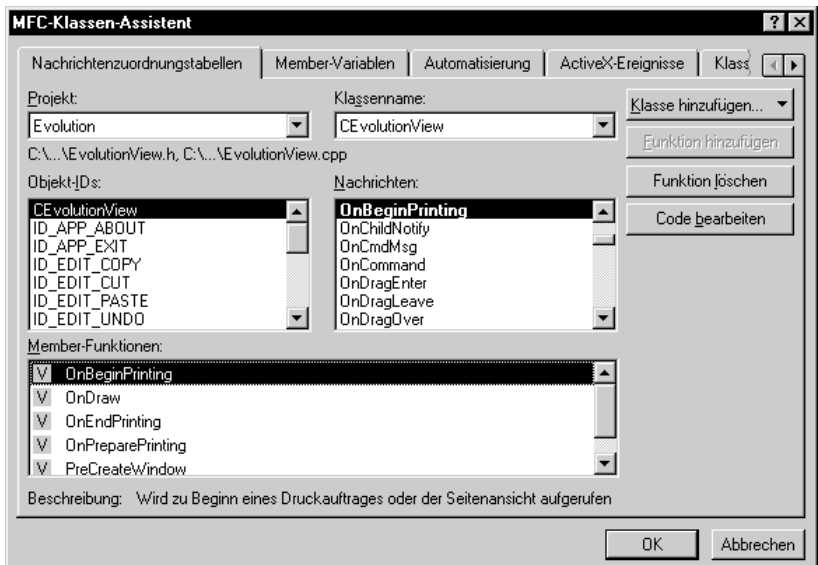


Abbildung 13.12: Nachrichtenbearbeitung mit dem Klassen-Assistenten



Sobald Sie eine Nachricht auswählen, wird eine kurze Beschreibung am unteren Rand des Dialogfelds des Assistenten angezeigt.

Das Kombinationsfeld *Objekt-ID's* der Nachrichtenzuordnungstabelle des Klassen-Assistenten enthält neben der Ansichtsklasse die Symbolnamen der Menüpunkte der Anwendung sowie die Nachrichten des Programmgerüsts. Im Kombinationsfeld *Nachrichten* werden die dazugehörigen virtuellen Member-Funktionen der Basisklassen bzw. die Windows-Nachrichten zur Auswahl angezeigt. Über die Schaltfläche **FUNKTION HINZUFÜGEN** wird die Member-Funktion der Klasse hinzugefügt. Dabei kann der vorgeschlagene Name der Funktion geändert werden.

Bereits vorhandene Funktionen werden im Kombinationsfeld *Nachrichten* fett dargestellt. Zu diesen Funktionen kann dann direkt in den Programmcode verzweigt werden. Auch ein Löschen der hervorgehobenen Funktionen ist möglich, jedoch entfernt der Klassen-Assistent nicht den Funktionsrumpf.

Die Nutzung des Klassen-Assistenten ist nur für MFC Benutzeroberflächenklassen möglich, die von *CCmdTarget* abgeleitet sind und Nachrichten behandeln oder Dialogfeld-Steuerungen verwalten.



13.4 Die Klasse *CDocTemplate*

Das Konzept der Dokument-Ansicht-Architektur der MFC findet ihren krönenden Abschluß in der Klasse der Dokumentvorlagen. Die Dokumente, Ansichten und Hauptfenster der Anwendung werden in Dokumentvorlagen zusammengefaßt. Die Klasse *CDocTemplate* der MFC ist die Basisklasse aller Dokumentvorlagen. Für Anwendungen mit nur einem aktiven Dokument verwendet man die abgeleitete Klasse *CSingleDocTemplate* (wie im Programm EVOLUTION), für Anwendungen mit mehreren Dokumenten die Klasse *CMultiDocTemplate* (wie im Programm BIBLIOTHEK). Jede dokumentbasierende Anwendung besitzt eine Liste *m_templateList* aller aktiven Dokumentvorlagen. Mit der Methode *AddDocTemplate* der Klasse *CWinApp* werden die Dokumentvorlagen an die Liste angefügt. *CWinApp* ruft dann ihrerseits die Methode *AddDocTemplate* von *CDocManager* auf.

```
void CDocManager::AddDocTemplate(CDocTemplate* pTemplate)
{
    if (pTemplate == NULL)
    {
        if (pStaticList != NULL)
        {
            POSITION pos = pStaticList->GetHeadPosition();
            while (pos != NULL)
            {
                CDocTemplate* pTemplate =
                    (CDocTemplate*)pStaticList->GetNext(pos);
                AddDocTemplate(pTemplate);
            }
            delete pStaticList;
            pStaticList = NULL;
        }
        bStaticInit = FALSE;
    }
    else
    {
```

```
ASSERT_VALID(pTemplate);  
ASSERT(m_templateList.Find(pTemplate, NULL) == NULL);  
// must not be in list  
pTemplate->LoadTemplate();  
m_templateList.AddTail(pTemplate);  
}  
}
```

Beim Aufruf von *ProcessShellCommand* oder *OnFileNew* der Anwendung wird die Liste der Dokumentvorlagen vom Programmgerüst ausgewertet. Ist nur eine Vorlage enthalten, wird diese durch *OpenDocumentFile* sofort aktiviert. Befinden sich bereits mehrere Vorlagen in der Liste, erscheint ein Dialog zur Auswahl der Vorlage (siehe Abbildung 13.13). *OpenDocumentFile* erzeugt das Dokument mit der Methode *CreateNewDocument* und das Hauptfenster der Vorlage mit *CreateNewFrame*. Als letztes wird in *OpenDocumentFile* die Methode *InitialUpdateFrame* von *CDocTemplate* aufgerufen, in der dann die Ansicht erzeugt wird.

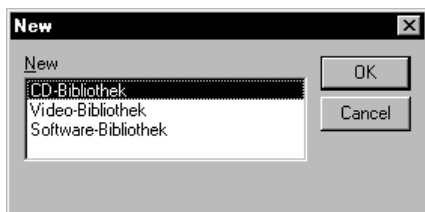


Abbildung 13.13: Mehrere Dokumentvorlagen

Das komplexe Zusammenspiel von Klassen, wie es in der Dokumentvorlage der MFC realisiert wurde, ist nur durch die Verwendung der Programmiersprache C++ und der Klassenhierarchie der MFC möglich. Durch das Benutzen der Makros `DECLARE_DYNCREATE` und `IMPLEMENT_DYNCREATE` bei der Deklaration und Implementierung der Klassen ist die Klassenbibliothek in der Lage, Objekte bestimmter Klassen dynamisch anzulegen und zu verwalten.

Ein Objekt der Dokumentvorlage hat vier Parameter. Der erste Parameter ist ein Symbolname und hat mehrere Aufgaben bei der Anlage der Dokumentvorlage durch das Programmgerüst:

1. Zeichenfolge in der Stringtabelle für Fenstertitel, Namenswurzel, Dokumententyp; Beschreibung der Filter für *FileOpen*; Typkennung für Registrierung und deren Beschreibung (Parameter durch `\n` getrennt).
2. Symbolname der Anwendung bzw. des Dokuments (Icon).

3. Menü des aktiven Fensters.

4. Zugriffstasten.

Der zweite Parameter ist das Dokument, der dritte ist das Hauptfenster der Vorlage, und der vierte Parameter ist die Ansichts-Klasse. Die Parameter zwei bis vier werden als *CRuntimeClass* übergeben. Dies ermöglicht die oben beschriebene Variabilität der Dokumentvorlage.

```
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(
    IDR_MYTYPE,
    RUNTIME_CLASS(CMyDoc),
    // Benutzerspezifischer MDI-Child-Rahmen
    RUNTIME_CLASS(CChildFrame),
    RUNTIME_CLASS(CMyView));
AddDocTemplate(pDocTemplate);
```



Der Anwendungs-Assistent generiert diese Implementation in *InitInstance* sowohl für MDI- als auch für SDI-Anwendungen. Die Unterschiede werden in den entsprechenden Abschnitten erläutert. Für beide gilt jedoch: Wenn der Benutzer einen Befehl auswählt, der ein Dokument erstellt wie DATEI ÖFFNEN, ruft das Gerüst die Dokumentvorlage auf, um das Dokumentobjekt, dessen Ansicht und das Rahmenfenster zu erstellen.

13.5 SDI-basierende Anwendung – ein Beispiel

13.5.1 Besonderheiten einer SDI-Anwendung

SDI-Anwendungen (Single Document Interface), wie das Beispiel EVOLUTION, sind Programme mit nur einem Dokument zur Datenhaltung und einer oder mehrerer Ansichten auf die Daten des Dokuments. Bei SDI-Anwendungen ist das Hauptfenster der Dokumentvorlage auch gleichzeitig das Hauptfenster der Anwendung. Die Dokumentvorlage der SDI-Anwendung ist die Klasse *CSingleDocTemplate*. Sie wird vom Anwendungs-Assistenten in der Methode *InitInstance* der Anwendung angelegt.

```
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
    IDR_MAINFRAME,
    RUNTIME_CLASS(CEvolutionDoc),
    RUNTIME_CLASS(CMainFrame),    // SDI
    RUNTIME_CLASS(CEvolutionView));
AddDocTemplate(pDocTemplate);
```

Listing: Anlage der SDI-Dokumentvorlage

Eine SDI-Anwendung unterstützt zwar nur ein Dokument, kann aber mehrere Dokument-Typen verwalten. So können also mehrere Vorlagen vom Typ *CSingleDocTemplate* in die Liste der Dokumentvorlagen aufgenommen werden. Es erscheint dann bei jedem *OnFileNew* der Auswahldialog für die entsprechende Vorlage. Für jede Vorlage wird auch ein neues Hauptfenster der Anwendung erzeugt, was dann dem eigentlichen Sinn einer SDI-Anwendung nicht ganz entspricht und auch meist keine sinnvolle Anwendung darstellt.

Wenn es erforderlich ist, können mehrere Ansichten auf ein Dokument einer SDI-Anwendung realisiert werden. Das Programmgerüst unterstützt bereits eine zweite Ansicht, die Druckvorschau. Für verschiedene gleichzeitige Ansichten auf die Daten des Dokuments ist der Einsatz von geteilten Fenstern möglich. Dafür stellt die MFC eine spezielle Ansichtsklasse zur Verfügung: *CSplitterWnd* (siehe auch Kapitel 13.7).

13.5.2 Aufgabe

Das Programm *EVOLUTION* basiert auf der berühmt gewordenen Simulation *Game Of Life* des amerikanischen Mathematikers J. H. Conway. Die Simulation läuft auf einem »unendlich großen« zweidimensionalen Spielplan ab, der in rechteckige Felder unterteilt ist. Jedes Feld besitzt zu jeder Zeit einen von zwei Zuständen, es enthält entweder eine lebende Zelle, oder es ist leer. Der Anfangszustand des Spielplans (d.h. die Platzierung lebender Zellen) wird vom Anwender vorgegeben, alle weiteren Zustände (die Nachfolge-Generationen) werden mit zwei sehr einfachen Regeln berechnet:

1. Besitzt ein leeres Feld genau drei lebende Zellen als Nachbarn, so wird auf diesem Feld eine neue Zelle angelegt (Geburt einer Zelle), andernfalls bleibt das Feld leer.
2. Besitzt eine lebende Zelle weniger als zwei Nachbarn, so stirbt sie durch Vereinsamung, bei mehr als drei Nachbarn durch Überfüllung (Tod einer Zelle). Nur, wenn sie genau zwei oder drei Nachbarn besitzt, bleibt sie am Leben.

Der Generationswechsel erfolgt synchron, d.h. um die Nachfolgegeneration zu ermitteln, werden alle Felder simultan betrachtet und der jeweilige Nachfolgezustand errechnet. Erst nachdem die Folgezustände aller Felder errechnet sind, werden sie tatsächlich umgesetzt und bilden damit eine neue Generation.

Jede Zelle hat acht Nachbarn, jeweils links, rechts, oben, unten und auf den Diagonalen. Um dies auch bei den Randzellen zu gewährleisten, werden gegenüberliegende Ränder als nebeneinanderliegend betrachtet, so daß ein quasi unendlicher Spielplan simuliert wird.

Berühmt wurde *Game of Life* vor allem deshalb, weil es aus einem definierten Anfangszustand und zwei einfachen Regeln sehr komplexe Muster erzeugen kann. Betrachtet man die Generationswechsel in schneller Folge, so entsteht der Eindruck, daß es sich tatsächlich um ein echtes, chaotisch anmutendes Zellwachstum handeln könnte.

Abbildung 13.14 zeigt einige einfache Muster, die als Ausgangspunkt für eigene Experimente verwendet werden können. Der *Läufer* ist ein zyklisches Muster mit einer Periodenlänge von 4, der sich auf dem Spielfeld von links oben nach rechts unten bewegt. Der *Blinker* hat eine Periode von zwei und wechselt immer zwischen waagerechter und senkrechter Darstellung. Die konstanten Muster verändern ihr Aussehen überhaupt nicht. Und die *Treppe* schließlich ist ein Beispiel für ein Muster, das sich zunächst sehr stark ausbreitet und, aber nach 63 Generationen in vier Quadranten endet, die sich nicht mehr verändern.

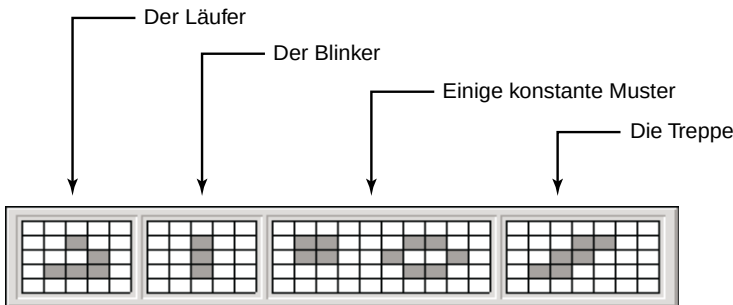


Abbildung 13.14: Einige Zellmuster

Um dem Spiel etwas Eigendynamik zu geben, wird die Simulation um einen Effekt erweitert. Pro Durchgang soll eine Anzahl von »Mutationen« eingestreut werden, die an beliebiger Stelle auftreten können. Das hat zur Folge, daß an den betreffenden Stellen eine Zelle quasi aus dem Nichts entsteht bzw. wenn an der betreffenden Stelle bereits eine Zelle existiert, stirbt diese ab. Mit dieser Erweiterung können von selbst neue Muster und Abfolgen entstehen. Die Entwicklung ist nicht mehr vorauszuberechnen.

13.5.3 Bedienung

EVOLUTION ist sehr einfach zu bedienen. Der dokumentorientierte Ansatz äußert sich darin, daß es möglich ist, den aktuellen Zustand des Spielplans in einer Datei zu speichern und ihn später wieder zu laden. Darüber hinaus bietet das Menü *DATEI* Funktionen zum Neuanlegen, zur Seitenansicht und zum Drucken des aktuellen Musters und erlaubt es, die vier zuletzt bearbeiteten Dateien sofort wieder aufzurufen. Das Menü *BEARBEITEN* ist

stillgelegt, die Funktionen zum Bearbeiten der Zwischenablage sind also nicht aktiviert. Mit Hilfe des Menüs *ANSICHT* kann sowohl die Statuszeile als auch die Symbolleiste an- oder ausgeschaltet werden. Außerdem ist die Anzeige- oder besser die Berechnungsgeschwindigkeit veränderbar, um bei schnellen Computern die Entwicklung noch mit dem Auge verfolgen zu können. Zusätzlich gibt es noch das Menü *HILFE*, das es ermöglicht, den Dialog *Info über ...* anzuzeigen.

Neben den Menüs, die vom Anwendungs-Assistenten generiert wurden, gibt es noch das Menü *GENERATION*, mit dem die nächsten Generationen berechnet werden können. Hier besteht die Möglichkeit, eine, zehn oder unendlich viele Nachfolgenerationen zu berechnen. Wurde die letzte Möglichkeit gewählt, berechnet das Programm die jeweils nächste Generation so lange, bis auf dem Spielfeld die rechte Maustaste gedrückt wird. Außerdem kann in diesem Menü die Anzahl der Mutationen festgelegt werden.

Die wichtigsten Funktionen stehen auch in der Symbolleiste zur Verfügung. Ganz links gibt es die Möglichkeit, Dateien neu anzulegen, zu öffnen oder zu schließen. Die danebenliegenden Schaltflächen bedienen die Zwischenablage und sind – wie die dazugehörigen Menüpunkte – deaktiviert. Rechts daneben befindet sich die Schaltfläche zum Drucken. Noch weiter rechts sitzen die drei selbstdefinierten Schaltflächen, mit denen die Nachfolgenerationen berechnet werden, und ganz am rechten Ende befindet sich die Schaltfläche zum Aufruf des Dialogs *Info über*

Mit Hilfe der linken Maustaste kann das Spielfeld manipuliert werden. Jeder Druck auf diese Taste schaltet den Zustand der Zelle unter dem Mauszeiger um. In Zusammenarbeit mit dem Menüpunkt *DATEI|NEU* oder der entsprechenden Schaltfläche kann so der Anfangszustand des Spielfeldes festgelegt werden. Während der Berechnung der Nachfolgeneration ist es nicht möglich, den Zustand einer Zelle auf diese Weise zu ändern.

13.5.4 Implementierung

Das Spielfeld ist eine rechteckige Matrix fester Größe, die durch ein zweidimensionales logisches Array dargestellt wird. Enthält ein Element des Arrays den Wert *WAHR*, so repräsentiert es eine lebende Zelle, andernfalls eine tote. Ein zweites, identisch aufgebautes Array dient dazu, die Folgezustände temporär zu speichern.

Um die Nachfolgeneration zu berechnen, muß für jedes Feld die Anzahl der lebenden Nachbarn ermittelt werden. Dazu wird ein Nachbarschaftsarray verwendet, das für jeden der acht Nachbarn den x- und y-Offset zur aktuellen Zelle enthält. Unter Berücksichtigung der Spielfeldränder können somit relativ schnell die Nachbarschaftsfelder untersucht und die Anzahl der lebenden Nachbarn gezählt werden.

Nun wird aus dem aktuellen Zustand und der Anzahl der Nachbarn der Folgezustand bestimmt und in dem zweiten Array gespeichert. Nachdem alle Folgezustände ermittelt wurden, wird der Inhalt des zweiten Arrays in das erste übertragen und der Bildschirm auf den neuesten Stand gebracht.

Zum Schluß werden die Mutationen in das Spielfeld durch einen Zufalls-generator eingestreut.

13.5.5 Projektanlage mit dem MFC-Anwendungs-Assistenten

Inzwischen haben Sie bereits Erfahrung bei der Projektanlage durch den Anwendungs-Assistenten. Das Programm EVOLUTION soll eine SDI-Anwendung werden, also müssen Sie im Schritt 1 des Assistenten die Option *Einzelnes Dokument (SDI)* wählen. Bei allen weiteren Schritten folgen Sie den Vorschlägen des Anwendungs-Assistenten. Welche Dateien hat der Anwendungs-Assistent für das Projekt EVOLUTION erzeugt? Sie finden die Dateien in dem Projektverzeichnis EVOLUTION (Step 1) und dessen Unterverzeichnissen (siehe Tabelle 13.2).

Dateiname	Bedeutung, Inhalt
Evolution.h Evolution.cpp	Implementierungs- und Schnittstellendatei für die Anwendung Evolution, in diesen Dateien ist die von <i>CWinApp</i> abgeleitete Applikationsklasse enthalten, zusätzlich ist noch die <i>Info über ...</i> -Dialogfeldklasse <i>CaboutDlg</i> integriert.
MainFrm.h MainFrm.cpp	In diesen Dateien ist die Klasse <i>CMainFrame</i> des Haupt-SDI-Rahmenfensters enthalten. Sie ist von <i>CFrameWnd</i> abgeleitet.
EvolutionView.cpp EvolutionView.h	Hier befindet sich die Ansichtsklasse <i>CEvolutionView</i> der Anwendung.
EvolutionDoc.cpp EvolutionDoc.h	Diese Dateien enthalten das Dokument <i>CEvolutionDoc</i> der Anwendung.
Evolution.rc Resource.h	Die Ressource-Datei zum Projekt mit allen bereits wichtigen Ressourcen wie Icon, <i>Info über ...</i> -Dialog, Toolbar usw. In <i>Resource.h</i> sind die zugehörigen symbolischen Konstanten definiert. Weitere Ressourcen sind im Unterverzeichnis RES abgelegt.
StdAfx.h StdAfx.cpp	Enthält die Anweisungen zum Einbinden der Schnittstellendatei der MFC. Aus diesen wird die vorkompilierte Header-Datei <i>Evolution.pch</i> erzeugt, um den Erstellungs-Vorgang zu beschleunigen.
Evolution.clw	Binärdatei mit Informationen für den Klassen-Assistenten.
ReadMe.txt	Eine Textdatei mit Informationen zu den angelegten Dateien, ähnlich wie in dieser Tabelle beschrieben.

Tabelle 13.2: Dateien des Projektes EVOLUTION

Eine Übersicht der Dateien ist auch im *Dateien*-Fenster des Arbeitsbereiches vorhanden (siehe Abbildung 13.15). Das Projekt ist bezüglich der Dateien fast komplett. Zusätzlich müssen Sie später nur noch einen Dialog

implementieren, der die Anzahl der Mutationen steuert. Sie können die Anwendung bereits erstellen, doch hat sie natürlich noch keine besondere Funktionalität.

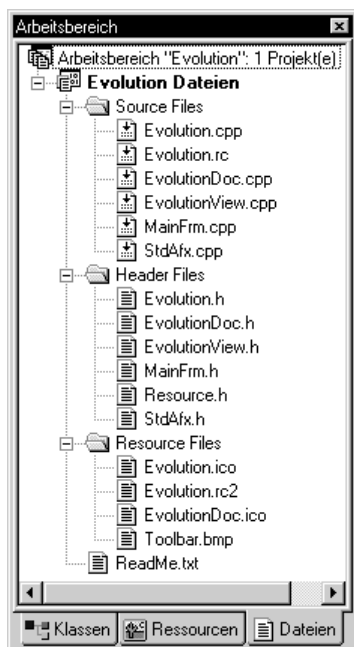


Abbildung 13.15: Ansicht im Dateien-Arbeitsbereich

13.5.6 Anpassung der Ressourcen

Gestaltung der Symbolleiste

Wie bereits beschrieben, soll die Symbolleiste um drei Schaltflächen erweitert werden. Dazu starten Sie den Symbolleisten-Editor durch einen Doppel-Klick auf die Toolbar IDR_MAINFRAME im Ressourcen-Fenster des Arbeitsbereiches. Der Editor startet automatisch im Symbolleistenmodus. Das heißt, daß der Editor das Bitmap in eine Symbolleiste konvertiert. Gleichzeitig wird am Ende der Leiste eine leere Schaltfläche eingefügt, die durch ein gestricheltes Quadrat hervorgehoben wird. Diese können Sie durch Anklicken aktivieren und bearbeiten. Mit der Maus können Sie die Schaltflächen im oberen Teil des Fensters durch Verschieben neu positionieren.

Über das Menü BILD des Developer Studios können Sie in den Grafikmodus des Editors umschalten. Damit können Sie zwar die Symbolleiste im Ganzen bearbeiten, jedoch geht die Aufteilung der Schaltflächen in der Anzeige verloren, und Sie müssen die Pixel einzeln auszählen.

Ergänzen Sie die drei Schaltflächen für eine, zehn und unendlich viele Generationen. Positionieren Sie diese zwischen dem Drucker-Symbol und dem Fragezeichen, und geben Sie ihnen die Symbolnamen `ID_GEN_WEITER`, `ID_GEN_WEITER10` und `ID_GEN_UNENDLICH`. Ihre Symbolleiste sollte dann etwa wie in Abbildung 13.16 aussehen.



Abbildung 13.16: Symbolleiste für Evolution

Entwerfen des Icons

Ihre Anwendung soll natürlich ein eigenes Symbol (Icon) erhalten. Standardmäßig generiert der Anwendungs-Assistent das MFC-Symbol für jede Anwendung und das Dokument-Symbol für die Dokumente. Das MFC-Symbol hat den Symbolnamen `IDR_MAINFRAME`. Um das Icon im Symbol-Editor zu bearbeiten, doppelklicken Sie auf das Icon im *ResourceView*-Fenster. Das Icon hat standardmäßig eine Größe von 32x32 Punkten mit maximal 256 Farben. Diese Fläche erscheint zwar auf den ersten Blick sehr klein, doch mit etwas gutem Willen kann man doch beachtliche Ergebnisse erzielen. Versuchen Sie sich als Künstler. In Abbildung 13.17 sehen Sie Ihr Ziel.

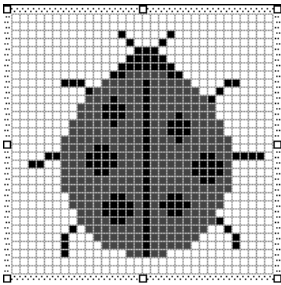


Abbildung 13.17: Programm-Symbol für Evolution

Im Symbol-Editor können Sie die rechte Maustaste **nicht** für das Kontextmenü verwenden, da diese für die Bildbearbeitung verwendet wird.

Es ist schwierig, eine Dokument-Ansicht-Anwendung, die mit dem Visual Studio erzeugt wird, davon zu überzeugen, ein anderes Programm-Symbol auch wirklich anzunehmen. Das blaue MFC-Symbol scheint sehr hartnäckig zu sein. Das liegt daran, daß das Developer Studio die Windows-Ressourcen der Anwendung zusätzlich in einer vorkompilierten Form hält (*.aps). Um Ihre Anwendung mit einem eigenen Programm-Symbol zu versehen, gehen Sie wie folgt vor:





Legen Sie ein neues Programm-Symbol mit dem Ressourcen-Editor an und speichern Sie es. Löschen Sie es gleich wieder im *Ressource*-Fenster des Arbeitsbereiches. Die Datei mit Ihrem Programm-Symbol bleibt erhalten. Schließen Sie die Entwicklungsumgebung, und gehen Sie mit dem Windows-Explorer in das *res*-Verzeichnis Ihrer Anwendung. Löschen Sie dort die alte Symboldatei (*Anwendung.ico*) und benennen die Datei mit dem neuen Symbol um, so daß diese jetzt den Namen der Anwendung trägt. Wechseln Sie in das Stammverzeichnis Ihres Projektes, und löschen Sie dort die Datei mit der Endung *.aps*. Nun können Sie die Entwicklungsumgebung neu starten und Ihr Projekt laden. Jetzt sollte Ihre Anwendung das gewünschte Symbol haben.

Zusätzliche Dialogfeldvorlage

Anders als bei einer dialogbasierenden Anwendung gibt es hier kein Dialogfeld als Hauptfenster der Anwendung. Beim Dokument-Ansicht-Konzept bildet die Ansichtsklasse mit dem Rahmenfenster das Hauptfenster der Anwendung. Der Anwendungs-Assistent erzeugt standardmäßig auch nur eine Dialogvorlage mit dem Symbolnamen `IDD_ABOUTBOX` für den *Info über ...* Dialog. Diesen können Sie zwar auch ändern und erweitern, doch für diese Beispiel-Anwendung benötigen Sie noch ein zusätzliches Dialogfeld zur Steuerung des Zufallsgenerators der Mutation.

Die Dialogvorlage hat natürlich bei weitem nicht die Komplexität wie der Rechnerdialog vom Beispielprogramm *HEXE*. Der Dialog dient lediglich der Eingabe der Anzahl von Mutationen auf dem Spielfeld. Erzeugen Sie also ein neues Dialogfeld mit dem Symbolnamen `IDD_MUTATION` und ein Eingabefeld (`IDC_MUTATIONEN`), ein Kontrollkästchen (`IDC_SICHTBAR`) und ein Textfeld. Entfernen Sie das Systemmenü über das Eigenschaftsfenster, beschriften Sie das Dialogfeld mit *Eingabe*, und formatieren Sie das Ganze, wie im Beispielprogramm *HEXE* beschrieben, bis es etwa wie in Abbildung 13.18 aussieht.

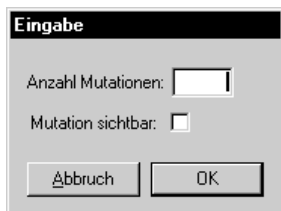


Abbildung 13.18: Dialogvorlage für die Eingabe der Anzahl von Mutationen



Wenn Sie dem Eingabefeld im Eigenschaftsfenster auf der Karte *Format* die Eigenschaft *Numerisch* geben, sparen Sie sich die Überprüfung der Zulässigkeit der Eingabe.

Erweiterung des Hauptmenüs

Als letzte Ressource müssen Sie nun noch das Hauptmenü erweitern. Zunächst erweitern Sie den Menüpunkt ANSICHT. Dazu doppelklicken Sie auf das Menü IDR_MAINFRAME im *ResourceView*-Fenster. Im Menü-Editor des Developer Studios erscheint das Hauptmenü im Bearbeitungsmodus. Klappen Sie das Menü ANSICHT auf, erweitern es zuerst durch einen Trennstrich und dann durch ein neues Popup-Menü. Geben Sie ihm die Bezeichnung GESCHWINDIGKEIT. Klappen Sie dieses Menü ebenfalls auf, und fügen Sie die Menübefehle NIEDRIG (ID_ANSICHT_NIEDRIG), NORMAL (ID_ANSICHT_NORMAL) und HOCH (ID_ANSICHT_HOCH) ein (siehe Abbildung 13.19).

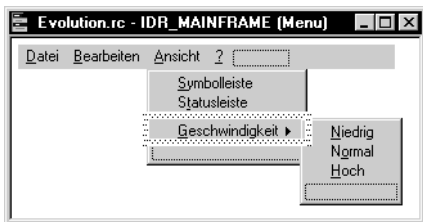


Abbildung 13.19: Erweiterung des Menüpunktes »Ansicht«

Fügen Sie anschließend ein Popup-Menü in das Hauptmenü zwischen ANSICHT und FRAGEZEICHEN ein. Beschriften Sie es mit GENERATIONEN. Fügen Sie die Menübefehle NÄCHSTE, NÄCHSTEN 10 und UNBEGRENZT ein, und geben Sie Ihnen die selben Symbolnamen wie den Schaltflächen der Symbolleiste mit gleicher Bedeutung. Nach einer Trennlinie fügen Sie den Menübefehl ANZAHL MUTATIONEN mit dem Symbolnamen ID_MUTATION ein. Dieser Menüpunkt dient später dem Aufruf der Dialogvorlage IDD_MUTATION zur Steuerung der Mutationen auf dem Spielfeld. Das Menü sollte nun wie in Abbildung 13.20 aussehen.

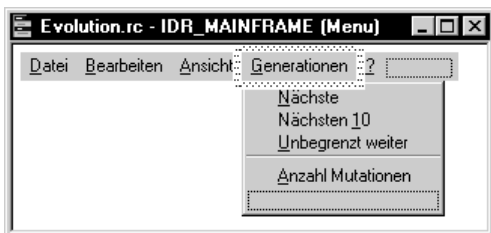


Abbildung 13.20: Das Popup-Menü »Generationen«

Erstellen Sie die Anwendung neu (Step 2), und sehen Sie sich das Ergebnis der Ressourcen-Anpassung an.

Im nächsten Schritt geht es an das Ausprogrammieren des Datenmodells, der Anzeigeroutinen, der Funktionalität zum Laden und Speichern der Daten und der Implementierung der Steuerung mit dem dazugehörigen Programmcode.

13.5.7 Zusätzlicher Programmcode

Datenmodell und Zugriffsmethoden

Zur Implementierung von EVOLUTION ist es erforderlich, eine Datenstruktur für den Spielplan zu entwerfen. Die naheliegendste – wenn auch nicht beste – Lösung, einen rechteckigen Spielplan mit lebenden oder toten Zellen darzustellen, besteht darin, ein logisches Array derselben Größe zu verwenden.

Diese Darstellung ist natürlich weder im Hinblick auf ihr Laufzeit- noch auf ihr Speicherverhalten effizient, doch geht es bei EVOLUTION einzig und allein um die Windows-Programmierung, so daß mit ruhigem Gewissen die einfachste Datenstruktur verwendet werden kann. Fügen Sie also folgende Variablen und Funktionen mit dem Klassen-Assistenten zur Klasse *CEvolutionDoc* hinzu:

```
#define CELLSIZE 30;

class CEvolutionDoc : public CDocument
{
//...
public:
    BOOL arCell[CELLSIZE][CELLSIZE];
    BOOL arCellNext[CELLSIZE][CELLSIZE];

    void NextGeneration();
    void ToggleCell(int x, int y);
    BOOL IsAlive(int x, int y);
//...
public:
    virtual void DeleteContents();
//...
}
```

Listing: Erweiterung der Dokumentenklasse

Der Spielplan ist rechteckig und hat eine fixe Größe von 30 x 30 Zellen. Dieser Wert wird durch die Konstante *CELLSIZE* in *EvolutionDoc.h* festgelegt. Wenn Sie die Größe des Spielfeldes verändern wollen, müssen Sie nur die Konstante anpassen. Daneben besitzt die Klasse *CEVOLUTIONDOC* zwei boolesche Arrays, *arCell* und *arCellNext*, mit der Größe 30 x 30. In *arCell* wird der aktuelle Zustand des Spielplans festgehalten. Ist ein Wert gesetzt,

so entspricht dieser einer lebenden Zelle an der korrespondierenden Position, ist er nicht gesetzt, so lebt an dieser Stelle keine Zelle. Das zweite Array dient als temporärer Zwischenspeicher bei der Berechnung der Nachfolgegeneration. Mit diesen beiden Arrays ist die Datenstruktur von *CEvolutionDoc* bereits beendet, weitere Daten-Member sind wegen der Einfachheit der Aufgabenstellung nicht erforderlich.

In *EVOLUTION* muß auf die Daten des Dokuments auf unterschiedliche Arten zugegriffen werden:

1. Der gesamte Spielplan muß initialisiert werden.
2. Der Zustand einer Zelle muß zwischen »lebend« und »tot« umgeschaltet werden können.
3. Der aktuelle Zustand einer Zelle muß abgefragt werden können.
4. Die Nachfolgegeneration des aktuellen Spielplans muß berechnet werden können.
5. Die Zell-Mutation muß implementiert werden.

Diese Zugriffe werden allesamt durch spezielle Methoden realisiert, ein direkter Zugriff auf die Daten erfolgt nur innerhalb dieser Methoden. Sie werden natürlich nicht von der Basisklasse *CDocument* zur Verfügung gestellt, sondern müssen von Ihnen selbst implementiert werden. Der Grund dafür ist, daß *CDocument* lediglich das Gerüst für eine abstrakte Datenklasse liefert, von der tatsächlichen Darstellung der Daten und der Art des Zugriffs darauf aber nichts weiß.

Am besten, Sie machen sich bei der Implementierung des Dokuments zunächst völlig frei von irgendwelchen Gedanken an die Windows- oder MFC-Programmierung und verfahren so, als würde man eine abstrakte Datenklasse in C++ realisieren.

Zur Initialisierung des Spielplans dient die Methode *DeleteContents*, wie im Kapitel »Datenaktualisierung« von *CDocument* beschrieben.

```
void CEvolutionDoc::DeleteContents()
{
    for (int y = 0; y < CELLSIZE; ++y)
    {
        for (int x = 0; x < CELLSIZE; ++x)
        {
            arCell[y][x] = FALSE;
        }
    }
    CDocument::DeleteContents();
}
```

Listing: Initialisierung des Spielfeldes in der Methode DeleteContents

DeleteContents macht nichts weiter, als jedes Element von *arCell* auf FALSE zu setzen und erzeugt damit einen völlig leeren Spielplan. Im nächsten Abschnitt wird deutlich, daß *DeleteContents* bei gewissen Gelegenheiten automatisch aufgerufen wird, um das Dokument in einen definierten Anfangszustand zu versetzen – beispielsweise vor dem Laden einer Datei oder beim Neuanlegen eines Dokuments. Genaugenommen lautet die bestimmungsgemäße Aufgabe von *DeleteContents*: Lösche alle Daten und initialisiere die Datenstruktur, so daß es möglich ist, das Dokument-Objekt erneut zu benutzen. Es ist zu beachten, daß es sich hier um eine virtuelle Methode handelt, die bereits in der Basisklasse *CDocument* implementiert ist.

Das Umschalten des aktuellen Zustands einer Zelle erledigen Sie mit Hilfe der Methode *ToggleCell*:

```
void CEvolutionDoc::ToggleCell(int x, int y)
{
    arCell[y][x] = ! arCell[y][x];
    SetModifiedFlag();
    UpdateAllViews(NULL, LPARAM(1));
}
```

Listing: Negation des Zelleninhalts in *ToggleCell*

ToggleCell negiert zunächst den booleschen Inhalt des korrespondierenden Array-Elements und ruft dann die beiden Methoden *SetModifiedFlag* und *UpdateAllViews* auf, um bekanntzumachen, daß sich das Aussehen des Spielfeldes geändert hat.

Mit Hilfe von *SetModifiedFlag* wird in *CEvolutionDoc* ein internes Merkmal gesetzt, mit dem sich die Klasse merkt, daß ihre Daten seit dem letzten Speichern verändert wurden. Ist dieses Merkmal gesetzt, so wird der Anwender beim Schließen des Dokuments gefragt, ob er seine Änderungen noch sichern will. Der konsequente Aufruf dieser Methode nach jeder Datenänderung verringert die Gefahr von versehentlichen Datenverlusten des Anwenders.

Nachdem die Daten als geändert markiert wurden, muß möglicherweise die Darstellung des Spielplans auf dem Bildschirm verändert werden. Da das Dokument weder weiß, wie viele Ansichten geöffnet sind, noch welchen Teil der Daten diese jeweils anzeigen, teilt es durch Aufruf von *UpdateAllViews* allen Ansichten mit, daß sich die Daten geändert haben. Der Aufruf dieser Methode führt in allen betroffenen Ansichts-Klassen zum Aufruf der Methode *OnUpdate*, die dafür verantwortlich ist, das Neuzeichnen der Ansicht einzuleiten. In unserem Fall bekommt nur die Klasse *CEvolutionView* diese Update-Nachricht.

Die Funktionsweise und die Parameter von *UpdateAllViews* wurden zwar bereits erläutert, zum besseren Verständnis soll aber hier am Beispiel der Mechanismus noch einmal vertieft werden. Der Grundgedanke dabei ist, daß es möglich sein soll, nur den Teil der Ansichten zu aktualisieren, der tatsächlich durch die Datenänderung betroffen wurde.

In *pSender* benennt der Aufrufer die Ansicht, die für die Änderung der Daten verantwortlich war, und schließt sie dadurch von der Benachrichtigung aus. Wird *NULL* angegeben, so geht die Nachricht an alle Ansichten. Wenn nur eine kleine Datenänderung vorgenommen wurde, ist es nicht immer sinnvoll, die komplette Ansicht neu zu zeichnen. Um den betroffenen Bereich einzugrenzen, können der Methode *OnUpdate* in den Parametern *lHint* und *pHint* einige Informationen übergeben werden. Auf welche Weise die Kodierung dieser Informationen erfolgt, bleibt dem Programm überlassen. Mit *lHint* kann zunächst ein *LPARAM* übergeben werden, also eine 32-Bit-Ganzzahl. Wenn das nicht ausreicht, kann eine (von *CObject* abgeleitete) eigene Klasse zur Verwaltung beliebig komplexer Informationen entworfen und ein Zeiger auf ein solches Objekt übergeben werden. *EVOLUTION* macht nur Gebrauch von *lHint* und übergibt 0, wenn der Fensterhintergrund neu gezeichnet werden soll, und 1, wenn dies nicht erforderlich ist.

```
void CEvolutionView::OnUpdate(CView *pSender, LPARAM lHint,
                             CObject *pHint)
{
    if (lHint)
    {
        Invalidate(FALSE);
    }
    else
    {
        Invalidate(TRUE);
    }
}
```

Listing: Wie die Ansichtsklasse auf die Änderung von Daten im Dokument reagiert

Falls *lHint* gleich 0 ist, wird das Fenster inklusive Hintergrund als ungültig markiert, andernfalls bleibt der Bildhintergrund unverändert. Durch differenzierte Aufrufe von *UpdateAllViews* könnte das Dokument also bereits in dieser sehr einfachen Form deutlichen Einfluß auf den Neuaufbau des Fensters nehmen. (Tatsächlich übergibt *CDocument* in *lHint* immer 1; 0 wird nur einmal während der Initialisierungsphase von *OnInitialUpdate* übergeben).

Soll herausgefunden werden, ob eine bestimmte Zelle am Leben ist oder nicht, kommt die Methode `IsAlive` zum Einsatz:

```
BOOL CEvolutionDoc::IsAlive(int x, int y)
{
    return arCell[y][x];
}
```

Listing: Zustandsabfrage einer Zelle (gekapselte Daten)

Sie liefert `TRUE`, wenn die Zelle lebt, andernfalls `FALSE`.

Die bei weitem aufwendigste Aufgabe wird von der Methode *NextGeneration* erledigt. Sie berechnet die Nachfolgeneration des Spielplans.

```
void CEvolutionDoc::NextGeneration()
{
    // Vererbungsregel
    static int neighbourpos[8][2] =
    {
        {-1,-1}, { 0,-1}, { 1,-1}, {-1, 0},
        { 1, 0}, {-1, 1}, { 0, 1}, { 1, 1},
    };
    int neighbours,nx,ny,x,y;
    //hier kommt später die Mutationsregel hin
    for (y = 0; y < CELLSIZE; ++y)
    {
        for (x = 0; x < CELLSIZE; ++x)
        {
            neighbours = 0;
            for (int i = 0; i < 8; ++i)
            {
                nx = x + neighbourpos[i][0];
                ny = y + neighbourpos[i][1];
                if (nx < 0) nx = CELLSIZE-1;
                if (ny < 0) ny = CELLSIZE-1;
                if (nx >= CELLSIZE) nx = 0;
                if (ny >= CELLSIZE) ny = 0;
                if (arCell[ny][nx]) ++neighbours;
            }
            if (arCell[y][x] && !(neighbours==2 || neighbours==3))
                arCellNext[y][x] = FALSE;
            else
                if (!arCell[y][x] && neighbours == 3)
                    arCellNext[y][x] = TRUE;
                else

```

```

        arCellNext[y][x] = arCell[y][x];
    }
}
for (y = 0; y < CELLSIZE; ++y)
    for (x = 0; x < CELLSIZE; ++x)
        arCell[y][x] = arCellNext[y][x];
//.hier kommt später die zweite Mutationsregel hin
SetModifiedFlag();
UpdateAllViews(NULL,LPARAM(1));
}

```

Listing: Berechnung der nächsten Generation

Aufgrund der Definition des Spiels ist es möglich (und nötig), den Folgezustand jeder Zelle separat zu berechnen. *NextGeneration* durchläuft dazu alle Zellen und bestimmt jeweils die Anzahl ihrer lebenden Nachbarn. Um deren Position möglichst schnell und ohne große Fallunterscheidung herauszufinden, verwendet die Methode das Nachbarschaftsarray *neighbourpos*, in dem für alle acht Nachbarn einer Zelle die x- und y- Koordinatenoffsets gespeichert werden. Durch einfaches Addieren der Offsets zu den aktuellen Zellkoordinaten ergeben sich damit die Koordinaten der Nachbarn, deren Zustand von Interesse ist. Eine kleine Sonderbehandlung ist erforderlich, um jeden der vier Ränder des Spielfeldes mit dem gegenüberliegenden Rand zu verbinden.

Nachdem die Anzahl der lebenden Nachbarn ermittelt wurde, kann entschieden werden, wie der Folgezustand der Zelle ist, um ihn anschließend an derselben Position in das Array *arCellNext* einzutragen. Nachdem auf diese Weise alle Zellen abgearbeitet wurden, enthält *arCellNext* die Nachfolgegeneration von *arCell*, und durch einfaches Kopieren der Arrays wird diese wieder nach *arCell* gebracht. Mit Hilfe von *SetModifiedFlag* und *UpdateAllViews* werden dann sowohl die eigene Klasse als auch die zugehörigen Ansichten von der Änderung unterrichtet.

Nachdem das Datenmodell und die Zugriffsmethoden in die Klasse *CEvolutionDoc* implementiert wurden, muß untersucht werden, welche Objekte von *CEvolutionDoc* erzeugt werden und wann dies geschieht. Um hierauf die korrekte Antwort zu finden, ist zwischen MDI- und SDI-Anwendungen zu unterscheiden. In einer SDI-Anwendung (wie z.B. EVOLUTION) gibt es lediglich ein einziges Objekt der Klasse *CDocument*. Es wird während der Initialisierungsphase des Programms angelegt und dann für alle Dokumente, die während der Laufzeit des Programms geladen oder neu angelegt werden, wiederverwendet. Anders sieht es bei MDI-Anwendungen aus. Hier wird bei jedem Neuanlegen oder Laden eines Dokuments ein neues Objekt angelegt und beim Schließen desselben wieder zerstört.

Diese Vorgehensweise macht deutlich, daß es – wenigstens bei SDI-Anwendungen – keinen Sinn hat, die Initialisierung der Datenstruktur im Konstruktor der Dokument-Klasse zu erledigen. Während der Konstruktor nur ein einziges Mal aufgerufen wird, ist es in einer SDI-Anwendung sehr viel häufiger nötig, die Daten zu initialisieren – nämlich immer dann, wenn eines der Kommandos NEU, ÖFFNEN oder SCHLIESSEN aus dem Menü DATEI aufgerufen wurde. Dieselben Überlegungen gelten für den Destruktor der Dokument-Klasse, der aus diesen Gründen ebenfalls ungeeignet ist, eine Endebehandlung aufzunehmen.

Statt dessen sollten Initialisierung und Endebehandlung durch Überlagerung spezieller virtueller Methoden der Klasse *CDocument* erledigt werden, die nach verschiedenen Benutzeraktivitäten aufgerufen werden.

Falls irgendeine dieser Methoden überlagert wird, ist es erforderlich, darin die gleichnamige Methode der Basisklasse aufzurufen. Diese realisiert neben der speziellen Initialisierung und Endebehandlung der Datenstruktur auch die eigentliche Funktion dieser Methode: das Laden, Neuanlegen oder Schließen eines Dokuments.

Im Beispielprogramm *EVOLUTION* geschieht die Initialisierung, wie oben bereits beschrieben, durch den Aufruf der virtuellen Methode *DeleteContents*. Dadurch ist es sogar möglich, vollkommen auf das Überlagern von *OnOpenDocument*, *OnNewDocument* und *OnCloseDocument* zu verzichten, da auch sämtliche Aufräumarbeiten von *DeleteContents* erledigt werden können. Aus diesem Grund können Sie die vom Anwendungs-Assistenten eingefügte Methode *OnNewDocument* sowohl aus *EvolutionDoc.h* wie auch aus *EvolutionDoc.cpp* entfernen.

Bedienung mit der Maus

Um das Spielfeld editieren zu können, soll die linke Maustaste verwendet werden, das heißt, daß bei jedem Drücken der linken Maustaste die Zelle unter dem Mauscursor zwischen »lebend« und »tot« umgeschaltet werden soll. Ähnlich wie alle Windows-Ereignisse löst auch das Drücken der Maustaste eine ganz bestimmte Nachricht aus, in diesem Fall *WM_LBUTTONDOWN*.

Um auf eine Nachricht zu reagieren, definiert das Programmgerüst üblicherweise eine Methode in der Klasse, die die Nachricht empfangen soll, und verbindet diese mit Hilfe eines Eintrags in der Nachrichtenbehandlungsroutine mit der Nachricht. Dabei wird die Nachricht *WM_LBUTTONDOWN* über den Eintrag *ON_WM_LBUTTONDOWN* abgefangen und zum Aufruf der Methode *OnLButtonDown* umgesetzt:

```
ON_WM_LBUTTONDOWN:  
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
```

OnLButtonDown erwartet zwei Parameter. In *nFlags* wird eine Kombination von Flags übergeben, die anzeigt, welche Sondertasten derzeit gedrückt sind:

nFlags	Sondertaste
MK_CONTROL	Strg-Taste
MK_LBUTTON	Linke Maustaste
MK_MBUTTON	Mittlere Maustaste
MK_RBUTTON	Rechte Maustaste
MK_SHIFT	Umschalt-Taste

Tabelle 13.3: Sondertastenflags

In *point* wird die aktuelle Position des Mausursors relativ zur linken oberen Ecke des Fensters übergeben.

Um die Methode der Ansichtsklasse hinzuzufügen, aktivieren Sie den Klassen-Assistenten für das Projekt EVOLUTION. Wählen Sie unter *Objekt-IDs* die Ansichtsklasse *CEvolutionView* aus. Suchen Sie im Kombinationsfeld *Nachrichten* nach *WM_LBUTTONDOWN* und fügen Sie über die Schaltfläche *FUNKTION HINZUFÜGEN* die Funktion *OnLButtonDown* ein (siehe Abbildung 13.21).

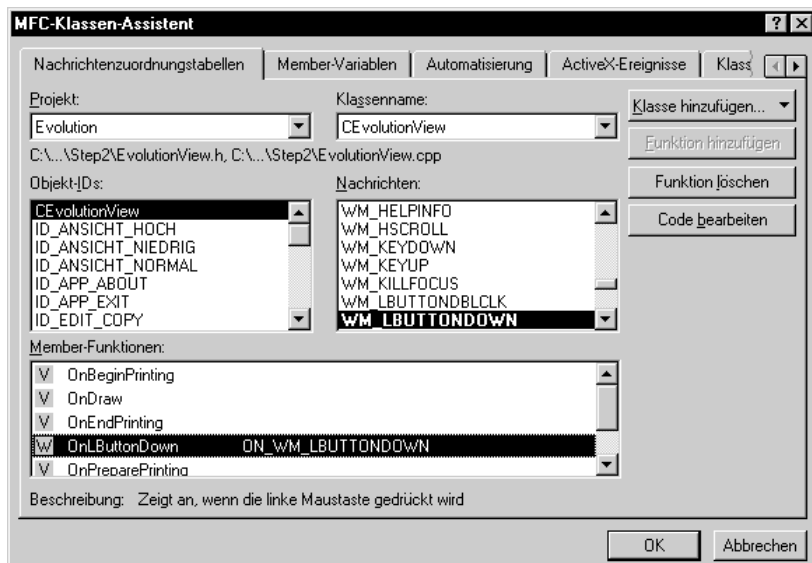


Abbildung 13.21: Einfügen der Funktion OnLButtonDown

Nun können Sie die Methode durch Drücken von *Code bearbeiten* ausprogrammieren. Sie sollte wie folgt aussehen:

```
void CEvolutionView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CRect rect;
    int cx, cy;

    GetClientRect(rect);
    cx = rect.right / CELLSIZE;
    cy = rect.bottom / CELLSIZE;
    GetDocument()->ToggleCell(point.x / cx, point.y / cy);

    CView::OnLButtonDown(nFlags, point);
}
```

Listing: Methode, die beim Drücken der linken Maustaste ausgeführt wird

Zunächst wird wieder die Größe des Arbeitsbereichs bestimmt, um daraus die x- und y- Ausdehnung der Zelle zu errechnen. Teilen Sie die Koordinaten des Mauszeigers durch die jeweilige Ausdehnung einer Zelle, so erhalten Sie die x- und y- Position der Zelle, die sich unter dem Mauszeiger befindet. Diese wird an die Methode *ToggleCell* des Dokuments übergeben, welche die Datenstruktur ändert und durch Aufruf der Funktion *UpdateAllViews* die Neudarstellung der Ansicht veranlaßt.

Berechnen der nächsten Generationen

Auf die gleiche Art wie bei der Nachricht *WM_LBUTTONDOWN* können Sie mit dem Klassen-Assistenten die drei *WM_COMMAND*-Nachrichten aus dem Menüpunkt *GENERATION* verwenden, um die entsprechenden Funktionen anzulegen. Der einzige Unterschied bei der Bedienung besteht darin, daß in dem Kombinationsfeld *Objekt IDs* nicht der Klassenname der Ansichtsklasse angegeben wird, sondern der Bezeichner des jeweiligen Menüpunkts ausgewählt wird. Die Bezeichner sind in der folgenden Tabelle aufgelistet:

Menüpunkt	Bezeichner	Funktionsname
Nächste	ID_GEN_WEITER	<i>OnGenWeiter</i>
Nächsten 10	ID_GEN_WEITER10	<i>OnGenWeiter10</i>
Unbegrenzt weiter	ID_GEN_UNENDLICH	<i>OnGenUnendlich</i>

Tabelle 13.4: Menüpunkte »Generation« und ihre Bezeichner

Nach *Funktion hinzufügen* öffnet sich jeweils ein Fenster, in dem der vorgeschlagene Name der Methode verändert werden kann, was normalerweise nicht nötig ist. Editieren Sie die Funktionen wie folgt:

```
void CEvolutionView::OnGenWeiter()
{
    CEvolutionDoc* pDoc = GetDocument();
    pDoc->NextGeneration();
}

void CEvolutionView::OnGenWeiter10()
{
    CEvolutionDoc* pDoc = GetDocument();
    for (int i = 1; i<=10; ++i)
    {
        pDoc->NextGeneration();
        UpdateWindow();
    }
}

void CEvolutionView::OnGenUnendlich()
{
    CEvolutionDoc* pDoc = GetDocument();
    MSG msg;
    while (1)
    {
        pDoc->NextGeneration();
        UpdateWindow();
        if (PeekMessage(&msg, m_hWnd, WM_RBUTTONDOWN,
                        WM_RBUTTONDOWN, PM_NOYIELD | PM_NOREMOVE))
        {
            break;
        }
    }
}
```

Listing: Die drei Methoden der Ansichtsklasse zum Berechnen der nächsten Generation

Die Methoden *OnGenWeiter* und *OnGenWeiter10* sind dabei nicht sehr spektakulär, denn sie haben nicht mehr zu tun, als die Methode *NextGeneration* des Dokuments jeweils ein- bzw. zehnmal aufzurufen und dafür zu sorgen, daß die Ergebnisse auf dem Bildschirm angezeigt werden.

Etwas mehr Aufwand müssen Sie betreiben, um die fortlaufende Berechnung und Darstellung neuer Generationen zu ermöglichen. Hier wird in einer Endlosschleife immer wieder folgendes getan:

1. Berechnen der nächsten Generation.
2. Aktualisieren des Bildschirms und Anzeige der neuen Generation.
3. Überprüfen, ob die Nachricht WM_RBUTTONDOWNDOWN eingetroffen ist; wenn ja, beenden der Schleife.

Die Verfügbarkeit einer WM_RBUTTONDOWNDOWN-Nachricht, also der Anzeige, daß die rechte Maustaste gedrückt wurde, wird mit Hilfe der globalen Funktion *PeekMessage* überprüft:

```
BOOL PeekMessage(LPMSG *lpMsg, HWND hWnd, UINT uFilterFirst,  
                INT uFilterLast, UINT fuRemove);
```

PeekMessage hat die Aufgabe, in der Nachrichtenschlange der Anwendung nachzusehen, ob eine Nachricht vorhanden ist und diese in der Struktur *lpMsg* zur Verfügung zu stellen. Eine besondere Eigenschaft von *PeekMessage* ist es, daß die Funktion auch dann terminiert, wenn keine Nachricht zur Verfügung steht, so daß die Anwendung unmittelbar nach dem Aufruf fortfahren kann. Der Rückgabewert zeigt an, ob eine Nachricht gefunden wurde (dann ist er ungleich 0) oder nicht (dann ist er gleich 0).

Der erste Parameter von *PeekMessage* ist die Adresse einer MSG-Struktur, um eine eventuell eingetretene Nachricht speichern zu können. Damit nicht alle Fenster durchsucht werden müssen, wird in *hWnd* der Handle des Fensters angegeben, dessen Warteschlange durchsucht werden soll. Um hier das aktuelle Fenster anzugeben, können Sie einfach den Daten-Member *m_hWnd* des aktuellen Fensterobjekts übergeben.

Die beiden Parameter *uFilterFirst* und *uFilterLast* dienen dazu, den Bereich der erwünschten Fensternachrichten einzuschränken. Dabei gibt *uFilterFirst* die erste und *uFilterLast* die letzte Nachricht des Bereichs an, der durchsucht werden soll. In *EVOLUTION* ist nur WM_RBUTTONDOWNDOWN interessant, so daß dieser Bezeichner in beiden Parametern übergeben wird.

Der letzte Parameter, *fuRemove*, erlaubt es, einen gewissen Einfluß auf das Verhalten der Funktion zu nehmen. Hier kann eine der symbolischen Konstanten *PN_REMOVE* oder *PM_NOREMOVE* angegeben werden, um zu entscheiden, ob die gefundene Nachricht aus der Queue entfernt wird oder ob sie darin erhalten bleibt. Zusätzlich ist es möglich, diesen Wert durch ein logisches Oder mit *PM_NOYIELD* zu verknüpfen, um zu verhindern, daß die aktuelle Anwendung unterbrochen und mit einer anderen Anwendung fortgeföhren wird.

Mutation

Um dem Spiel eine gewisse Eigendynamik zu geben, soll ein Zufallsgenerator auf Wunsch das Spielfeld beeinflussen. Die Steuerung erfolgt über den Dialog IDD_MUTATION. Legen Sie eine Dialogfeldklasse für diesen Symbolnamen an, geben Sie ihr den Namen *CDlgMutation* und speichern ihn in den Dateien *DlgMutation.cpp* und *DlgMutation.h*.

Für das Eingabefeld und das Kontrollkästchen der Dialogvorlage benötigen Sie je eine Variable, um die Eingaben zu speichern. Aktivieren Sie den Klassen-Assistenten, um die Variablen anzulegen und mit den Steuerungen zu verbinden. In der Registerkarte *Member-Variablen* des Assistenten sehen Sie die Symbolnamen unter *Steuerelement-IDs*.

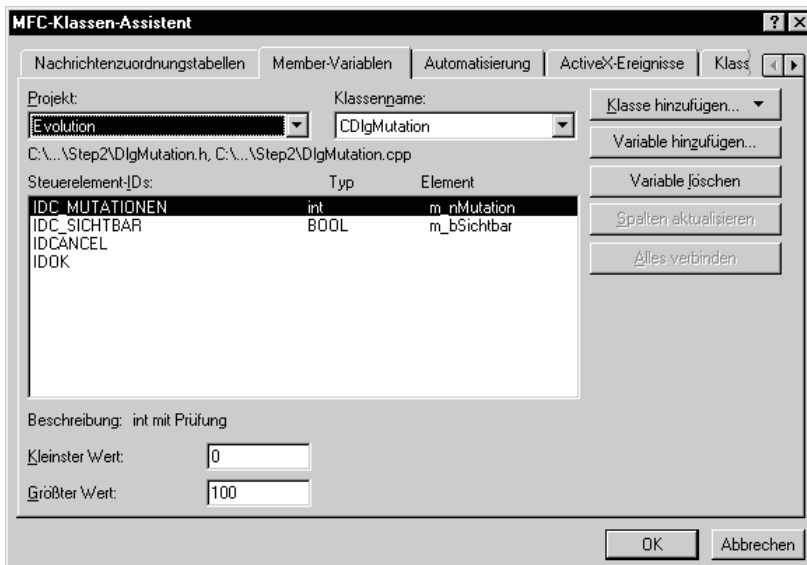


Abbildung 13.22: Variablen anlegen mit dem Klassen-Assistenten

Über die Schaltfläche VARIABLE HINZUFÜGEN legen Sie die Member-Variablen an und verbinden sie gleichzeitig mit dem Symbolnamen. Dabei dient ein Eingabedialog zur Auswahl des Datentyps und zur Namensvergabe. Je nach Art der Steuerung sind die Kombinationsfelder mit anderen Werten gefüllt. Die Unterscheidung nach *Kategorie* entscheidet, ob eine eigene Klasse für das Steuerelement angelegt werden soll, oder, ob nur der Wert interessiert.

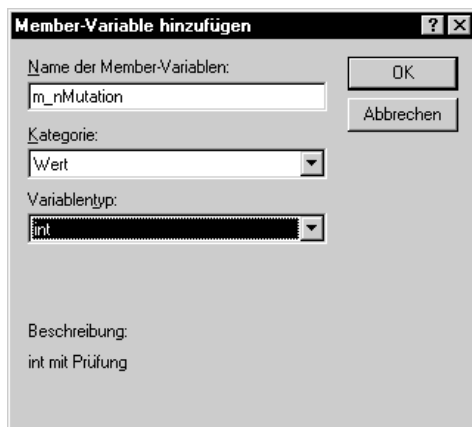


Abbildung 13.23: Member-Variablen hinzufügen

In der Nachrichtenbehandlungsroutine erzeugt der Klassen-Assistent folgende Einträge:

```
void CDlgMutation::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CDlgMutation)
    DDX_Text(pDX, IDC_MUTATIONEN, m_nMutation);
    DDV_MinMaxInt(pDX, m_nMutation, 0, 100);
    DDX_Check(pDX, IDC_SICHTBAR, m_bSichtbar);
    //}}AFX_DATA_MAP
}
```

Listing: Nachrichtenbehandlung in CDlgMutation

Im Konstruktor der Klasse *CDlgMutation* werden die Variablen automatisch durch den Assistenten initialisiert:

```
CDlgMutation::CDlgMutation(CWnd* pParent /*=NULL*/)
: CDialog(CDlgMutation::IDD, pParent)
{
    //{{AFX_DATA_INIT(CDlgMutation)
    m_nMutation = 0;
    m_bSichtbar = FALSE;
    //}}AFX_DATA_INIT
}
```

Listing: Konstruktor von CDlgMutation

Damit ist der Dialog zur Eingabe der Mutations-Parameter bereits fertig. Für den Austausch zwischen Steuerung und der dazugehörigen Variablen sorgt das Programmgerüst der MFC. In den Member-Funktionen *InitDialog* und *OnOk* der Klasse *CDialog* wird die Methode *UpdateData* der Klasse *CWnd* aufgerufen, die dann ihrerseits *DoDataExchange* aufruft und für den Datenaustausch sorgt.

```
class CWnd::UpdateData
BOOL UpdateData(BOOL bSaveAndValidate = TRUE);
```

Der Parameter *bSaveAndValidate* entscheidet, ob die Daten zur Steuerung hin oder von der Steuerung zurück ausgetauscht werden sollen.

Jetzt muß der Aufruf des Dialogfeldes noch realisiert werden. Dazu legen Sie mit dem Klassen-Assistenten in bewährter Art und Weise für die Menü-Nachricht *GENERATIONEN|ANZAHL MUTATIONEN* eine Funktion an.

```
void CEvolutionView::OnMutation()
{
    CEvolutionDoc* pDoc = GetDocument();
    CDlgMutation DlgMutation(this);
    DlgMutation.m_nMutationen = pDoc->m_nMutationen;
    DlgMutation.m_bSichtbar = pDoc->m_bSichtbar;
    if (DlgMutation.DoModal() == IDOK)
    {
        pDoc->m_nMutationen = DlgMutation.m_nMutationen;
        pDoc->m_bSichtbar = DlgMutation.m_bSichtbar;
    }
}
```

Listing: Aufruf des Mutationsdialogs

Die Variablen der Dialogfeldklasse werden vor dem Aufruf von *DoModal* mit den Werten aus dem Dokument vorbelegt. Wenn der Dialog mit der OK-Schaltfläche verlassen wurde, übernehmen Sie deren Werte zurück in das Dokument.

Um in der Datei *EvolutionView.cpp* auf die Dialogfeldklasse zugreifen zu können, müssen Sie die Schnittstellendatei hinzufügen (*#include »DlgMutation.h«*).



Für die Implementierung des Zufallsgenerators sind zwei Stellen vorgesehen: eine vor der Neuberechnung der nächsten Generation und danach. Darüber entscheidet der Wert von *m_bSichtbar*. Nebenbei können dadurch auch verschiedene optische Resultate erzielt werden.

```
void CLIFEDoc::NextGeneration()
{
    //...
    if (!m_bSichtbar)
    {
        for (int i = 0; i < m_nMutationen; ++i)
        {
            int y = rand() * CELLSIZE / RAND_MAX;
            int x = rand() * CELLSIZE / RAND_MAX;
            arCell[y][x] = !arCell[y][x];
        }
    }
    //...
}
```

Listing: Erweiterung der Methode NexGeneration

Es wird eine zufällige x- und y- Position innerhalb des Arrays ermittelt. An dieser Stelle wird der Zelleninhalt einfach negiert. Diese Schleife wird so oft durchlaufen, wie es in der Variablen `m_nMutation` steht.

Bildschirmausgabe

Die Ausgabe des Spielfeldes erfolgt im Fenster der Ansichtsklasse. Dabei soll der Spielplan nach Möglichkeiten das gesamte Fenster ausfüllen. Dazu ist es wichtig, die Größe einer einzelnen Zelle zu erfahren. Diese ergibt sich – getrennt für die x- und y- Richtung – aus der Ausdehnung des Arbeitsbereichs geteilt durch die Anzahl der Zellen. Das Ergebnis der Divisionen gibt die Größe einer Zelle in ganzen Pixeln an und wird in den lokalen Variablen `cx` und `cy` abgelegt.

Multiplizieren Sie beide Werte wieder mit `CELLSIZE`, kommt wegen der ganzzahligen Division in der Regel ein Rechteck zustande, das etwas kleiner ist als der ursprüngliche Arbeitsbereich, und neben dem Spielfeld bleibt ein schmaler Rand übrig.

Zum Zeichnen des Gitters wird ein dunkelgrüner Stift von einem Pixel Breite benötigt. Um diesen zu erstellen, legen Sie die Variable `pen` der Klasse `CPen` an. Durch den Aufruf von `CreatePen` mit den angegebenen Parametern legen Sie die Eigenschaften des Stiftes fest:

1. er soll eine durchgehende Linie zeichnen (`PS_SOLID`)
2. die Breite soll ein Pixel betragen
3. die Farbe soll dunkelgrün sein (`RGB(0, 128, 0)`).

```
class CPen:
    CPen();
    BOOL CreatePen(int PenStyle, int nWidth, COLORREF crColor);
```

Neben dem Default-Konstruktor besitzt *CPen* noch einen Konstruktor, der dieselben Parameter akzeptiert wie *CreatePen*. Auf diese Weise können Sie die Eigenschaften eines Stiftes bereits beim Anlegen festlegen.

Nachdem ein Stift erzeugt wurde, führt dies noch nicht automatisch dazu, daß er bei allen zukünftigen Zeichenoperationen verwendet wird. Die Zeichenfunktionen verwenden grundsätzlich den Stift, der in den Device-Kontext selektiert wurde. Um einen Stift (und auch andere Zeichenobjekte) in den Device-Kontext zu selektieren, kann die Methode *SelectObject* aufgerufen werden:

```
class CDC:
    CPen *SelectObject(CPen *pPen);
```

SelectObject erwartet einen Zeiger auf ein Grafikobjekt, das in den Device-Kontext selektiert werden soll, und liefert einen Zeiger auf das bisherige Objekt desselben Typs zurück. Wird *SelectObject* das erste Mal für ein bestimmtes Zeichenobjekt – also beispielsweise einen Stift – aufgerufen, muß der Rückgabewert gesichert werden, um später wieder in den Device-Kontext geladen werden zu können. Hierzu dient in *OnDraw* die Variable *pOldPen*, die einen Zeiger auf den ursprünglichen Stift des Device-Kontextes erhält und diesen am Ende der Methode durch einen erneuten Aufruf von *SelectObject* wiederherstellt.

Es ist übrigens nicht nötig, ein *CPen*-Objekt nach Gebrauch separat zu löschen, wenn es als lokale Variable angelegt wurde (bei der SDK-Programmierung ist dazu ein separater Aufruf von *DeleteObject* erforderlich). *CPen* ist ein Abkömmling der Klasse *CGdiObject* und erbt von dieser den Destruktor, der beim Löschen des C++-Objekts automatisch einen Aufruf von *DeleteObject* durchführt.

Das Windows-GDI bietet zum Zeichnen von Linien zwei Funktionen an, die mit den Funktionen einer Turtle-Grafik verwandt sind. Eine Turtle-Grafik zeichnet sich durch folgende Eigenschaften aus:

1. Das Grafiksystem merkt sich ein Koordinatenpaar, das die aktuelle Stiftposition repräsentiert.
2. Es gibt eine Funktion, um den Stift von der aktuellen Position zu einer beliebigen anderen Position innerhalb des Zeichenbereichs zu bewegen und dabei eine Linie zu ziehen.
3. Es gibt darüber hinaus eine Funktion, um den Stift an eine neue Position zu bewegen, ohne eine Linie zu ziehen.

Der Vorteil der Turtle-Grafik ist, daß es einfacher ist, Linienzüge zu zeichnen, da jeweils das Ende eines Liniensegments mit dem Anfang des nächsten übereinstimmt. Der Nachteil ist, daß zum Zeichnen einer einzelnen Linie immer zwei Funktionsaufrufe erforderlich sind: Der erste, um den Stift an die Ausgangsposition zu bewegen, und der zweite, um von dort eine Linie zur Zielposition zu ziehen. Diese beiden Funktionsaufrufe stecken in den Methoden *MoveTo* und *LineTo*.

```
class CDC:
    CPoint MoveTo(int x, int y);
    CPoint MoveTo(POINT point);
    BOOL LineTo(int x, int y);
    BOOL LineTo(POINT point);
```

MoveTo bewegt den Stift an die Position (x, y) , ohne dabei eine Linie zu zeichnen, und liefert den vorigen Punkt in Form eines *CPoint*-Objekts zurück. *LineTo* bewegt den Stift von der aktuellen Position nach (x, y) und zieht dabei eine Linie. Zum Zeichnen der Linie wird dabei immer der aktuelle Stift des Device-Kontextes verwendet.

Bei der Verwendung von *LineTo* müssen Sie aufpassen, denn die Funktion hat eine (beabsichtigte) Eigenschaft, die sehr leicht zu Programmierfehlern führen kann: Sie bewegt zwar den Stift bis zum angegebenen Endpunkt, zeichnet diesen Punkt aber nicht mehr mit! Wenn Sie das nicht beachten, zeichnen Sie Linien, die einen Pixel zu kurz sind.

Diese beiden Linienfunktionen verwenden Sie in *OnDraw* der Ansichts-klasse, um das Gitter zu zeichnen. Dabei werden zuerst alle vertikalen und dann alle horizontalen Gitterlinien gezeichnet. Nachdem dies erledigt ist, wird wieder der anfängliche Stift in den Device-Kontext selektiert.

```
void CEvolutionView::OnDraw(CDC* pDC)
{
    //...
    CRect rect;
    int cx,cy,x,y;
    CPen pen,*pOldPen;

    GetClientRect(rect);
    cx = rect.right / CELLSIZE;
    cy = rect.bottom / CELLSIZE;
    //...
    // Zuerst das Gitter zeichnen
    pen.CreatePen(PS_SOLID, 1, RGB(0, 128, 0));
    pOldPen = pDC->SelectObject(&pen);
```



```

for (x = 0; x <= CELLSIZE; ++x)
{
    pDC->MoveTo(x * cx, 0);
    pDC->LineTo(x * cx, CELLSIZE * cy);
}
for (y = 0; y <= CELLSIZE; ++y)
{
    pDC->MoveTo(0, y * cy);
    pDC->LineTo(CELLSIZE * cx, y * cy);
}
pDC->SelectObject(pOldPen);
//...
}

```

Listing: Zeichnen des Gitters im Spielfeld

Zum Zeichnen der Zellen werden Pinsel gebraucht. Pinsel dienen in der Regel dazu, zusammenhängende Flächen in einer bestimmten Farbe oder mit einem bestimmten Muster einzufärben. *OnDraw* benötigt einen grünen Pinsel zum Zeichnen eines Feldes, das keine Zelle enthält, und einen roten Pinsel für ein Feld mit einer lebenden Zelle.

Um die Pinsel zu definieren, müssen Sie zunächst zwei Pinselvariablen *greenbrush* und *redbrush* der Klasse *CBrush* anlegen. Ähnlich wie *CPen*-Objekte besitzen auch *CBrush*-Objekte, die mit dem Default-Konstruktor erzeugt wurden, zunächst keine sinnvollen Eigenschaften, sondern erfordern den Aufruf einer *Create*-Funktion. Im Gegensatz zu Stiften gibt es für Pinsel eine ganze Reihe dieser Funktionen, die es unter anderem ermöglichen, gemusterte Pinsel anzulegen oder Pinsel, die mit einem Bitmap zeichnen. Das Anlegen der vollfarbigen Pinsel in *EVOLUTION* geschieht mit der Methode *CreateSolidBrush*:

```

class Cbrush:
    BOOL CreateSolidBrush(COLORREF crColor);

```

CreateSolidBrush erwartet als Parameter eine *COLORREF*-Struktur, die Sie mit dem RGB-Makro erzeugen können. Der Parameter wird benutzt, um den Pinsel in der gewünschten Farbe einzufärben. Der Rückgabewert ist *TRUE*, wenn alles gut ging, andernfalls *FALSE*.

Nach dem Erstellen der beiden Pinsel werden diese nicht nacheinander in den Device-Kontext selektiert – wie dies beim Stift der Fall war –, sondern als Parameter in der Funktion *FillRect* verwendet. *FillRect* erwartet als ersten Parameter ein Rechteck, das den Bereich angibt, der eingefärbt werden soll, und als zweiten Parameter einen Pinsel, der die Farbe der Füllung angibt.

Verwenden Sie in *OnDraw* die Funktion *FillRect* für jedes Feld des Spielplans. Mit Hilfe der Methode *IsAlive* der Dokument-Klasse, auf die über *GetDocument* zugegriffen werden kann, bestimmen Sie in *OnDraw* für jedes Feld, ob sich dort eine lebende Zelle befindet oder nicht, und färben das Feld dementsprechend rot oder grün ein.

```
class CDC:
    void FillRect(LPRECT lpRect, CBrush* pBrush);
```

An dieser Stelle von *OnDraw* stellen Sie zum ersten Mal eine Verbindung zwischen der Ansicht und dem Dokument her. Obwohl es einfach ist, soll die Vorgehensweise wegen ihrer Wichtigkeit noch einmal zusammengefaßt werden:

1. In der Ansichts-Klasse gibt es die Methode *GetDocument*, die einen Zeiger auf das zugeordnete Dokument liefert.
2. Mit Hilfe dieses Zeigers kann auf die Methoden des Dokuments zugegriffen und so der Zustand der Daten abgefragt oder manipuliert werden.

In *OnDraw* sieht das Zeichnen der Rechtecke dann folgendermaßen aus:

```
void CEvolutionView::OnDraw(CDC* pDC)
{
    CEvolutionDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    CRect rect;
    int cx,cy,x,y;
    CPen pen,*pOldPen;
    CBrush greenbrush,redbrush;

    GetClientRect(rect);
    cx = rect.right / CELLSIZE;
    cy = rect.bottom / CELLSIZE;
    //...
    // Nun die Zellen zeichnen
    greenbrush.CreateSolidBrush(RGB(0, 255, 0));
    redbrush.CreateSolidBrush(RGB(255, 0, 64));
    for (y = 0; y < CELLSIZE; ++y)
    {
        for (x = 0; x < CELLSIZE; ++x)
        {
            if (pDoc->IsAlive(x, y))
            {
```

```

        pDC->FillRect(CRect(x * cx + 1, y * cy + 1,
                           (x + 1) * cx, (y + 1) * cy), &redbrush);
    }
    else
    {
        pDC->FillRect(CRect(x * cx + 1, y * cy + 1,
                           (x + 1) * cx, (y + 1) * cy), &greenbrush);
    }
}
}

```

Listing: Zeichnen der Rechtecke im Spielfeld

Geschwindigkeitssteuerung

Um die Ausgabegeschwindigkeit steuern zu können, wird eine Pausen-Funktion in die Berechnungsroutine *NextGeneration* des Dokumentes eingefügt. Windows stellt dafür die Funktion *Sleep* zur Verfügung.

```

VOID Sleep(DWORD dwMilliseconds
           // sleep time in milliseconds);

```

Die Funktion versetzt Ihren Programmfaden für die übergebene Zeitdauer in eine Art Schlafzustand. Während dieser Zeit empfängt das Programm auch keinerlei Nachrichten.

Damit die Berechnungsgeschwindigkeit veränderlich gestaltet werden kann, wird die Zeitdauer über eine Member-Variable `m_nPause` des Dokumentes gesteuert. Im Konstruktor wird sie mit Null vorbelegt, was eine hohe Geschwindigkeit bedeutet. Diese Variable soll nun über das Menü ANSICHT|GESCHWINDIGKEIT geändert werden können. Außerdem soll die aktuelle Einstellung im Menü ersichtlich sein. Wie man Menü-Nachrichten mit einer Member-Funktion verbindet, wissen Sie bereits. Legen Sie also die folgenden drei Funktionen mit dem Klassen-Assistenten an:

```

void CEvolutionView::OnAnsichtNiedrig()
{
    GetDocument()->m_nPause = 200;
}

void CEvolutionView::OnAnsichtNormal()
{
    GetDocument()->m_nPause = 100;
}

void CEvolutionView::OnAnsichtHoch()
{
    GetDocument()->m_nPause = 0;
}

```

Listing: Steuerung der Anzeige(Rechen-)geschwindigkeit

Die Pausenzeiten für *Niedrig* und *Normal* können Sie auch variieren. Für die Zustandsanzeige innerhalb des Menüs verwenden Sie die Menüfeld-Nachricht `UPDATE_COMMAND_UI`, die in der Nachrichtenbehandlungs-routine durch das Makro `ON_UPDATE_COMMAND_UI` realisiert wird. Diese Nachricht wird zur Aktualisierung des Menüs verwendet. Fügen Sie also auch diese drei Funktionen mit dem Klassen-Assistenten hinzu:

```
void CEvolutionView::OnUpdateAnsichtNiedrig(CCmdUI* pCmdUI)
{
    pCmdUI->SetRadio(GetDocument()->m_nPause == 200);
}
void CEvolutionView::OnUpdateAnsichtNormal(CCmdUI* pCmdUI)
{
    pCmdUI->SetRadio(GetDocument()->m_nPause == 100);
}
void CEvolutionView::OnUpdateAnsichtHoch(CCmdUI* pCmdUI)
{
    pCmdUI->SetRadio(GetDocument()->m_nPause == 0);
}
```

Listing: Menüsteuerung über `UPDATE_COMMAND_UI`

Die so erzeugten Member-Funktionen bekommen einen Zeiger auf ein `CCmdUI`-Objekt übergeben. Diese Klasse enthält vier Methoden zur Manipulation des betreffenden Menüpunktes.

```
class CCmdUI:
    virtual void Enable(BOOL bOn = TRUE);
    virtual void SetCheck(int nCheck = 1);
    virtual void SetRadio(BOOL bOn = TRUE);
    virtual void SetText(LPCTSTR lpszText);
```

Die Methode *Enable* aktiviert bzw. deaktiviert den Menüpunkt. Mit *SetCheck* können Sie den Menüpunkt mit einem Haken versehen und mit *SetRadio* mit einem Punkt. Über *SetText* ist der Menütext änderbar.

Im Beispielprogramm *EVOLUTION* soll ein Punkt wie bei einem Optionsfeld vor dem Menütext sein. Verwenden Sie deshalb die Methode *SetRadio* innerhalb der Member-Funktion `UPDATE_COMMAND_UI`. Für die Steuerung wird der aktuelle Wert von `m_nPause` ausgewertet.



Damit ist das Beispiel fertig. Auf der Begleit-CD finden Sie den kompletten Quellcode und auch einige gespeicherte Muster. Auch dieses Beispiel kann von Ihnen noch ausgebaut werden, so können Sie z. B. die Serialisation erweitern und die Werte für die Geschwindigkeit und Mutation mitspeichern und auch wieder einlesen.

13.6 Die Klasse *CFormView*

13.6.1 Verwendung

CFormView ist eine Spezialform von *CScrollView*, die ihrerseits von *CView* abgeleitet ist. Objekte dieser Klasse erhalten ähnlich wie Dialoge eine Dialogvorlage. Auch in ihrem Verhalten erinnern sie an einen nicht-modalen Dialog, doch besitzt die Klasse alle Eigenschaften von *CScrollView* bzw. *CView*. Sie eignet sich besonders für Formulare, da hier alle Steuerelemente verwendet werden können. Außer dem Konstruktor besitzt die Klasse *CFormView* keine eigene öffentliche Methode.

Wie ein Dialog erhält eine formatierte Ansicht Nachrichten von den Steuerelementen. Der Datenaustausch mit DDX- und die Datenüberprüfung mit DDV-Methoden sind ebenfalls möglich. Auch vom Dialog bekannte *CWnd*-Methoden wie *GotoDlgCtrl* können verwendet werden. Darüber hinaus sind Objekte der Klasse *CFormView* auch in der Lage, Botschaften vom Menü oder von der Symbolleiste zu erhalten und entsprechend darauf zu reagieren.

Da die Klasse *CFormView* für die Bildschirmanzeige gedacht ist, liefert der Einsatz der Druckfunktionalität von *CView* nicht die erhofften Ergebnisse. Die eingebetteten Steuerelemente können nicht automatisch mitgedruckt werden. Vielmehr beschränkt sich die Funktionalität auf die von *CView* ererbten Methoden wie *OnDraw*.

Im MDI-Beispielprogramm BIBLIOTHEK werden zwei formatierte Ansichten benutzt. Eine enthält eine Liste mit der Übersicht aller Titel, die zweite dient der Eingabe und gleichzeitig der Anzeige aller Daten zu einem Titel.

13.6.2 Formularvorlagen formatierter Ansichten

Für das Anlegen der Formularvorlage mit dem Dialogeditor wird eine Dialogfeldvorlage benutzt. Dabei müssen folgende Einstellungen im Dialogfeld *Eigenschaften Formate* beachtet werden, damit das Fenster zur Laufzeit ordnungsgemäß aufgebaut werden kann:



Abbildung 13.24: Dialogfeld »Eigenschaften« Formate für formatierte Ansichten

Im Gegensatz zum Dialogfeld sind die Formate *Titelleiste* und *Systemmenü* nicht aktiv. Außerdem müssen *Format* auf *Untergeordnet* und *Rand* auf *Kein* stehen, sonst gibt es Probleme beim Laden der Ressource. Alle bekannten Steuerelemente für Dialogvorlagen können Sie auch in einer formatierten Ansicht verwenden. Der Dialogeditor behandelt die Ressource analog zu einem Dialogfeld.

13.6.3 Der Einsatz in der Anwendung

Der Einsatz der Klasse *CView* wurde bereits im vorangegangenen Teil dieses Buches beschrieben. Die Besonderheiten der Klasse *CFormView* werden in diesem Abschnitt erläutert.

Beim Anlegen einer Anwendung auf Basis von Dokumentvorlagen (SDI oder MDI) mit dem Anwendungs-Assistenten muß der Typ der Ansicht in Schritt 6 der Projektanlage bereits festgelegt werden (siehe Abbildung 13.25). Hier können dann auch gegebenenfalls die Dateinamen für Deklaration und Implementierung der Klassen angepaßt werden.

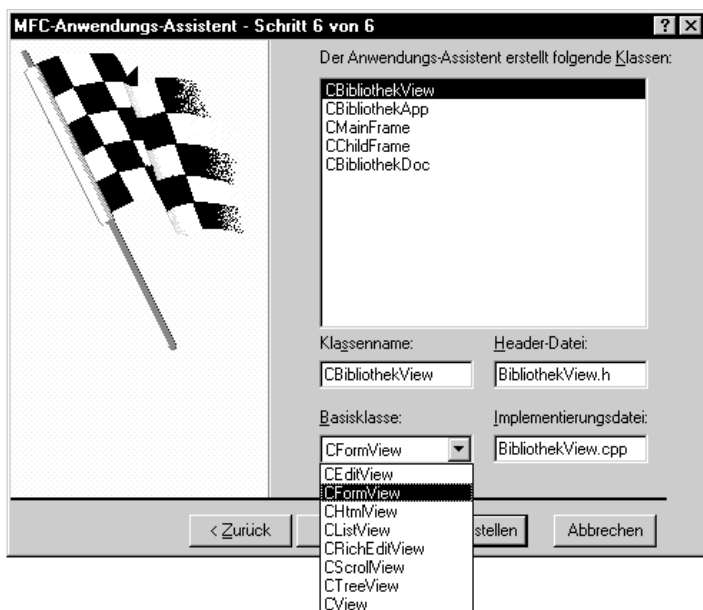


Abbildung 13.25: Auswahl der Basisklasse *CFormView* für die Ansichtsklasse

Wie bereits beschrieben hat die Klasse *CFormView* einen eigenen Symbolnamen: denjenigen einer Dialogfeldvorlage. Damit hat sie auch ähnliche Funktionalitäten wie die Klasse *CDialog*. Der Klassen-Assistent unterstützt das Anlegen von Variablen und Methoden für die von *CFormView* abgelei-

teten Klassen analog zur Klasse *CDialog*. Zu allen Steuerungen können Sie Member-Variable anlegen. Mit Hilfe des Klassen-Assistenten können Sie auch virtuelle Methoden wie *OnInitialUpdate* und *OnUpdate* für die formatierten Ansichts-Klassen überlagern.

13.7 Die Klasse *CSplitterWnd*

CSplitterWnd stellt eine besondere Darstellungsform für Ansichten zur Verfügung, und zwar geteilte Fenster. Diese Klasse ist direkt von *CWnd* abgeleitet. Es handelt sich um ein spezielles Rahmenfenster, das den Arbeitsbereich von *CFrameWnd* bei SDI- bzw. *CMDIChildWnd* bei MDI-Anwendungen voll ausfüllt. Die Größe und Aufteilung des Anzeigebereiches kann sowohl vom Programm als auch vom Anwender bestimmt werden. Die MFC besitzt zwei Formen von geteilten Fenstern: dynamische und statische.

In einem dynamisch geteilten Fenster kann der Anwender jederzeit das Fenster teilen und die Teilung auch wieder aufheben. Die einzelnen Ausschnitte gehören immer einer Ansichtsklasse an, d.h., in einem dynamisch geteilten Fenster sieht man verschiedene Teilabschnitte ein und derselben Ansicht. Dies ist vergleichbar mit einer Textverarbeitung, bei der man gleichzeitig verschiedene Stellen eines Dokumentes sehen und bearbeiten kann.

Bei einem statisch geteilten Fenster erfolgt die Aufteilung der Ausschnitte beim Aufbau der Objekte und kann danach nicht mehr geändert werden. In den Ausschnitten können verschiedene Ansichten dargestellt werden, wobei beim Aufbau bereits festgelegt werden muß, welche Ansichten sich wo befinden werden. Der Anwender kann nur die Fensterrahmen verschieben, nicht aber die Aufteilung aufheben.

Der Anwendungs-Assistent stellt in Schritt 4 unter WEITERE OPTIONEN ... eine Möglichkeit zur Verfügung, die Fensterstile zu ändern. Unter anderem auch die Option *Geteiltes Fenster verwenden*. Damit können sowohl SDI- als auch MDI-Rahmenfenster dynamisch geteilt werden. Bei SDI-Anwendungen ist der untere Teil des Dialoges inaktiv.

In der SDI-Anwendung muß die Klasse des Hauptfensters ein Objekt vom Typ *CSplitterWnd* und eine überladene Funktion *OnCreateClient* erhalten.

```
protected:
    CSplitterWnd m_wndSplitter;
protected:
    virtual BOOL OnCreateClient(LPCREATESTRUCT lpcs,
                               CCreateContext* pContext);
```

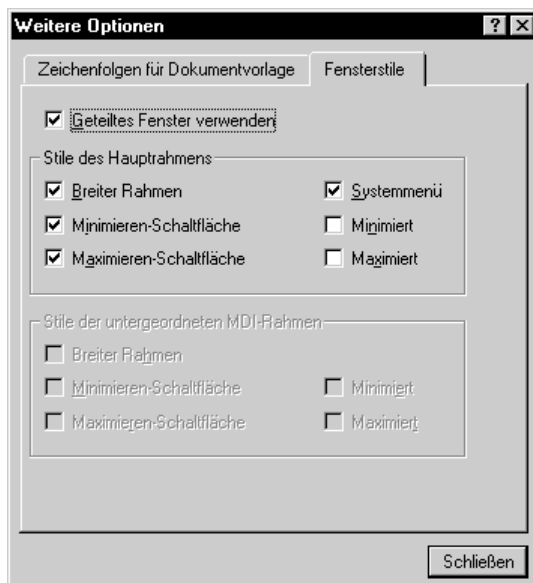


Abbildung 13.26: Weitere Optionen des Anwendungs-Assistenten für geteilte Fenster

Beim Anlegen des Hauptfensters wird die Methode *OnCreateClient* vom Programmgerüst aufgerufen. Anstelle eines einzelnen Ansichtsfensters, wie von der Vorlage vorgesehen, wird hier ein geteiltes Fenster aufgebaut. Die notwendigen Informationen für die Ansichtsklasse werden in *pContext* übergeben.

```

BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT /*lpcs*/,
                               CCreateContext* pContext)
{
    return m_wndSplitter.Create(this,
        2, 2,                // ZU ERLEDIGEN: Spalten- und Zeilenzahl
                               festlegen
        CSize(10, 10),      // ZU ERLEDIGEN: Minimale Größe der
                               Bearbeitungsfläche festlegen
        pContext);
}

```

Listing: Anlage des geteilten Fensters in einer SDI-Anwendung

Mit der Methode *Create* der Klasse *CSplitterWnd* wird ein dynamisch geteiltes Fenster angelegt.


```

CSplitterWnd::Create
    BOOL Create(CWnd* pParentWnd, int nMaxRows, int nMaxCols,
                SIZE sizeMin, CCreateContext* pContext,
                DWORD dwStyle = WS_CHILD|WS_VISIBLE|WS_HSCROLL|
                WS_VSCROLL|SPLS_DYNAMIC_SPLIT,
                UINT nID = AFX_IDW_PANE_FIRST );

```

Übergeben wird neben dem Zeiger auf das Hauptfenster die maximale Anzahl von Reihen und Spalten (2, 2). Mit *CSize(10, 10)* wird die minimale Ausschnittsgröße angegeben. Optional können die Fenstereigenschaften angegeben werden.

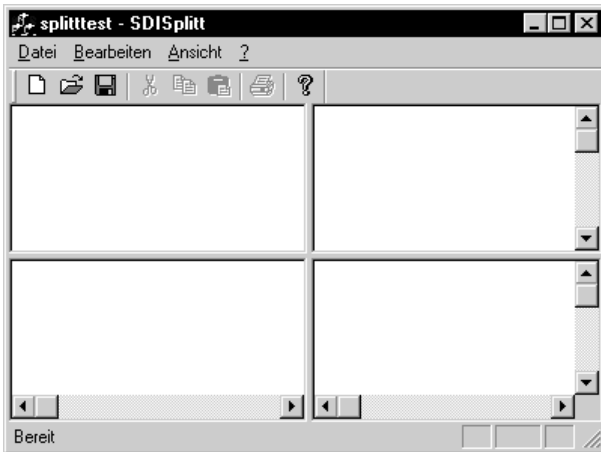


Abbildung 13.27: SDI-Anwendung mit dynamisch geteiltem Fenster

Für den Einsatz von *CSplitterWnd* in MDI-Anwendungen stellt der Anwendungs-Assistent in der von *CMDIChildWnd* abgeleiteten Klasse eine Instanz von *CSplitterWnd* – *m_wndSplitter* – und eine überladene Funktion *OnCreateClient* zur Verfügung.

Im Gegensatz zum dynamisch geteilten Fenster, bei dem die *Create* Funktion über *pContext* die Informationen zur anzuzeigenden Ansicht erhält, müssen Sie für ein statisch geteiltes Fenster die Methode *CreateStatic* verwenden.

```

CSplitterWnd::CreateStatic
    BOOL CreateStatic(CWnd* pParentWnd, int nRows, int nCols,
                    DWORD dwStyle = WS_CHILD | WS_VISIBLE,
                    UINT nID = AFX_IDW_PANE_FIRST );

```

Da der Anwendungs-Assistent die Verknüpfung der Ansichtsklassen mit dem statisch geteilten Rahmenfenster nicht automatisch unterstützt, müssen Sie im Anwendungsprogramm Ihre eigene Ansichtsklasse von *CView* oder deren Unterklassen ableiten und dem geteilten Fenster einzeln zuordnen. Dies geschieht über die Funktion *CreateView*.

```
class CSplitterWnd:
    virtual BOOL CreateView(int row, int col,
                           CRuntimeClass* pViewClass,
                           SIZE sizeInit,
                           CCreateContext* pContext );
```

Diese Zuordnung geschieht mit Hilfe des Makros `RUNTIME_CLASS`, der die Ansichtsklasse an die *CreateView*-Methode übergibt.

```
BOOL CChildFrame::OnCreateClient( LPCREATESTRUCT /*lpcs*/,
                                CCreateContext* pContext)
{
    BOOL bRet = m_wndSplitter.CreateStatic(this, 2, 2);
    bRet |= m_wndSplitter.CreateView(0, 0,
                                     RUNTIME_CLASS(CMyForm1), CSize(100, 100), pContext);
    bRet |= m_wndSplitter.CreateView(0, 1,
                                     RUNTIME_CLASS(CMyEdit1), CSize(100, 100), pContext);
    bRet |= m_wndSplitter.CreateView(1, 1,
                                     RUNTIME_CLASS(CMyEdit2), CSize(100, 100), pContext);
    bRet |= m_wndSplitter.CreateView(1, 0,
                                     RUNTIME_CLASS(CMyForm2), CSize(100, 100), pContext);
    return bRet;
}
```

Listing: Geteiltes Fenster mit statisch gebundenen Ansichten

Dabei bedeuten die ersten beiden Parameter die Position (Zeile, Spalte) im geteilten Fenster; der dritte Parameter ist die Ansicht selbst, und *CSize* bestimmt wieder die minimale Ausschnittsgröße.

In der Abbildung ist zu sehen, daß der Bereich mit der *FormView* nur Bildlaufleisten erhält, wenn die Dialogvorlage nicht komplett im Ausschnitt angezeigt werden kann. Alle anderen Ansichten haben separate Bildlaufleisten (Beispiel).



In der Klasse *CSplitterWnd* der MFC sind zwei Variablen enthalten, die die Breite der Teilungsfelder festlegen, `m_cxSplitter` und `m_cySplitter`. Leider sind diese als *protected* deklariert und können nur in abgeleiteten Klassen verändert werden.

Mit der Nachricht `ID_WINDOW_SPLIT` lässt sich die Ansichtsklasse teilen. Das Programmgerüst erkennt diese Nachricht und löst die entsprechende Behandlungsroutine aus. Zweckmäßigerweise ordnet man den entsprechenden Menüpunkt unter dem Eintrag `FENSTER` im Hauptmenü an. Der Anwender kann durch Doppelklick auf die Teilungsfelder die Teilung aufheben.



Abbildung 13.28: MDI-Anwendung mit statisch geteiltem Fenster

13.8 MDI-basierende Anwendung – ein Beispiel

13.8.1 Besonderheiten einer MDI-Anwendung

MDI-Anwendungen (Multiple Document Interface) sind Programme mit mehreren gleichzeitig aktiven Dokumenten. Diese Anwendungsart ist die bevorzugte Form der MFC-Anwendungen. Im Gegensatz zu SDI-Anwendungen hat die MDI-Anwendung ein eigenes Hauptfenster; *CMDIChildWnd* enthält hier die Ansichts-Klasse. Die Dokumentvorlage der MDI-Anwendung ist die Klasse *CMultiDocTemplate*. Sie wird wie *CSingleDocTemplate* vom Anwendungs-Assistenten in der Funktion *InitInstance* der Anwendung angelegt.

```
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(
    IDR_CDTYPE,
    RUNTIME_CLASS(CBibliothekDoc),
    RUNTIME_CLASS(CChildFrame),
```

```
// Benutzerspezifischer MDI-Child-Rahmen
RUNTIME_CLASS(CViewEinzel));
AddDocTemplate(pDocTemplate);
// ... weitere Dokumentvorlagen
```

Listing: Anlage der MDI-Dokumentvorlage

Da durch das Programmgerüst bei der MDI-Anwendung bereits ein Hauptfenster erzeugt wurde, kann der Aufruf von *ProcessShellCommand* in *InitInstance* auch entfallen. Das Hauptfenster wird komplett mit einem Menü angelegt und der Variablen *m_pMainWnd* zugewiesen. Über die Funktion *AfxGetApp* kann von der Anwendung aus auf das Hauptfenster zugegriffen werden. Dadurch ist es möglich, über das Menü des Hauptfensters den Aufruf der einzelnen Dokumentvorlagen zu steuern. Eine Dokumentvorlage kann außer mit *OnFileNew* auch mit *CreateNewFrame* und *InitialUpdateFrame* aktiviert werden.

Es gibt verschiedene Kombinationsmöglichkeiten von Dokument und Ansicht in einer MDI-Anwendung. So ist es möglich, die Dokumentvorlagen der Anwendung nur einmal anzulegen. Angenommen, in einer Anwendung werden zwei Dokumentvorlagen benötigt; dann sieht der Aufruf in *InitInstance* etwa folgendermaßen aus:



```
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(
    IDR_MDITYPE,
    RUNTIME_CLASS(CMdiDoc),
    RUNTIME_CLASS(CMDIChildWnd),
    RUNTIME_CLASS(CMdiView));
AddDocTemplate(pDocTemplate);
pDocTemplate = new CMultiDocTemplate(
    IDR_MDITYPE2,
    RUNTIME_CLASS(CMdiDocZwei),
    RUNTIME_CLASS(CMDIChildWnd),
    RUNTIME_CLASS(CMdiFormView));
AddDocTemplate(pDocTemplate);
```

Jede Dokumentvorlage hat eine eigene Ressource-ID. In der Anwendung gibt es zwei verschiedene Dokumente, von denen jedes mit einer Ansicht verbunden ist.

Es ist immer nur eine Vorlage aktiv. Der Titel des aktiven Fensters findet sich auch im Titel der Anwendung wieder. Die beiden Vorlagen halten jede eine eigene Instanz der Klasse. Es ist von der MFC nicht vorgesehen, daß die Dokumentvorlagen untereinander kommunizieren.

In einer MDI-Anwendung ist es aber auch wie bei einer SDI-Anwendung möglich, von einer Dokumentvorlage mehrere Instanzen anzulegen. Das bedeutet, daß mehrere Objekte von demselben Dokument und dessen Ansicht existieren. Auch hier findet keine Kommunikation zwischen den verschiedenen Ansichten statt. Vielmehr existieren alle Instanzen nebeneinander, ohne voneinander zu wissen.

Weiterhin ist es möglich, zu einer bestehenden Dokumentvorlage weitere Ansichten zu erzeugen. Dazu benutzt man die Nachricht `ID_WINDOW_NEW`, die sich unter `NEUES FENSTER` im Menü `FENSTER` befindet. Diese Nachricht wird im MDI-Hauptfenster (`CMDIFrameWnd`) des Programmgerüsts verarbeitet und führt dort die Methode `OnWindowNew` aus.

Damit existieren zu einer Instanz des Dokuments zwei Ansichtsobjekte, so daß die Ansichtsobjekte dieselben Daten benutzen können. Das macht es aber auch erforderlich, daß die einzelnen Objekte miteinander kommunizieren können, denn die einzelnen Objekte müssen reagieren, wenn sich die Daten geändert haben. Für dieses Problem stellt die MFC die Funktion `UpdateAllViews` der Dokumentenklasse zur Verfügung (siehe Kapitel 13.3).

In der MDI-Anwendung kann das jeweils aktive Rahmenfenster auch auf die Eingaben über die Symbolleiste reagieren. Dazu ist die entsprechende Nachricht der betreffenden Schaltfläche in die Nachrichtenbehandlungsroutine des Dokuments oder der Ansicht einzubinden.

13.8.2 Aufgabe

Wenn Sie bis an diese Stelle des Buches gekommen sind, kennen Sie bereits viele Funktionalitäten der Entwicklungsumgebung, insbesondere der Assistenten. Außerdem kennen Sie die wichtigsten Klassen der MFC und haben diese bereits in einfacher Form angewendet. Nun sollen diese Kenntnisse an einem weiteren Beispiel vertieft und erweitert werden. Das nun beschriebene Beispielprogramm `BIBLIOTHEK` wendet bereits Bekanntes an und fügt neue Funktionalität hinzu.

Es soll ein Archiv-Programm geschaffen werden, das vom Ansatz her erweiterbar sein soll. Beispielhaft soll eine CD-Bibliothek realisiert werden, es ist aber auch jede andere Form von Bibliothek möglich. Die Daten müssen auf Festplatte oder Diskette gespeichert und wieder eingelesen werden können. Es ist eine Ansicht als Übersicht und eine für Details vorzusehen. In der Detail-Ansicht müssen die Datensätze änderbar sein. Der Bediener soll sich komfortabel in den Datensätzen bewegen können.

Die Größe der Ansichtsklassenfenster muß sich an der Größe der Dialogvorlagen orientieren. Ein Minimieren der Anzeigefenster über die Titelleiste ist zu unterbinden. Außerdem soll es möglich sein, mehrere Bibliothek-Dokumente gleichzeitig zu öffnen.

13.8.3 Implementation

Das Projekt wird als MDI-Anwendung realisiert. Die Daten werden in einer eigenen Klasse gekapselt, die über die Eigenschaft der Serialisation verfügt, um ihre Daten auf den Datenträger zu speichern bzw. von dort lesen zu können. Die Datenstruktur ist einfach:

Bezeichnung	Variablenname	Beschreibung
Laufnummer	m_LfdNr	Nummer des Titels auf dieser CD
Titel	m_Titel	Einfach
Komponist	m_Komponist	<i>Komponist des Titels</i>
Interpret	m_Interpret	Interpreten des Titels
Datum	m_Datum	Datum der Aufnahme
Zeit	m_Zeit	Zeitdauer des Titels
Bemerkung	m_Bemerkung	Anmerkungen zum Titel

Tabelle 13.5: Datenbezeichner für Bibliothek

Die beiden Ansichten auf die Daten von BIBLIOTHEK sind als formatierte Ansichten zu realisieren, mit je einer eigenen Dialogvorlage. Die Synchronisation der beiden Anzeigen erfolgt mittels *UpdateAllViews*. Das Bewegen innerhalb der Datensätze geschieht entweder über das Menü oder mit Hilfe der Schaltflächen auf der Symbolleiste.

13.8.4 Projektanlage mit dem MFC-Anwendungs-Assistenten

Die Projektanlage mit dem Anwendungs-Assistenten wurde bereits mehrfach ausführlich erläutert. Deshalb wird hier nur noch auf die Besonderheiten dieses Projektes eingegangen.

Legen Sie also ein neues Projekt BIBLIOTHEK mit dem Anwendungs-Assistenten an und wählen in Schritt 1 des Assistenten *Mehrere Dokumente (MDI)*, welches die Voreinstellung ist. Im dritten Bild entfernen Sie die *AktivX-Steuerelemente*-Unterstützung. In Schritt 4 deaktivieren Sie die Option *Drucken und Seitenansicht*, da es für von *CFormView* abgeleitete Klassen keine einfache Implementation dieser Funktionalität gibt. Wählen Sie über die Schaltfläche *WEITERE OPTIONEN ...* das Dialogfeld zur Eingabe der *Zeichenfolgen für Dokumentvorlage* an. Der Assistent hat für Sie bereits fast alle Felder vorgelegt. Tragen Sie bei *Dateierweiterung* cd (ohne Punkt!) ein (siehe Kapitel 13.2.4 »Registrierung von Dokumenten«).

Wechseln Sie zur Registerkarte *Fensterstile*. Hier können Sie über das Aussehen der Fenster *CMDIChildWnd* und *CMDIFrameWnd* der Anwendung entscheiden. Die folgende Abbildung zeigt Ihnen die Möglichkeiten, Rahmen- und Titelzeilen-Eigenschaften anzupassen.



Abbildung 13.29: Ändern der MDI-Rahmen-Eigenschaften

Im BIBLIOTHEK-Programm erhalten die MDI-Kind-Fenster keine Schaltfläche zum Vergrößern bzw. Verkleinern in ihrer Titelzeile. Die Größe der Ansichtsfenster wird in der Anwendung direkt gesteuert.

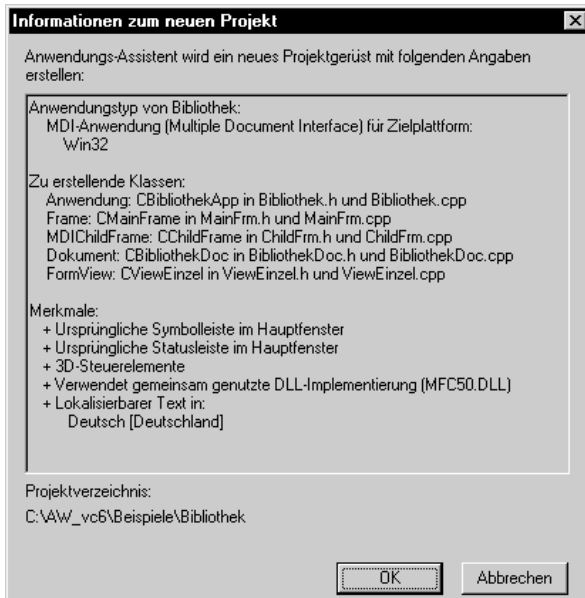


Abbildung 13.30: BIBLIOTHEK Projektinformationen

Da der Anwendungs-Assistent automatisch nur eine Ansichtsklasse anlegt, müssen Sie sich in Schritt 6 entscheiden, ob es die Einzel- oder die Übersichtsansicht sein soll. Vorschlag: Ändern Sie den Klassennamen in *CViewEinzel* ab und verwenden Sie *ViewEinzel* als Namen für die Schnittstellen- und Implementationsdatei. Vergessen Sie nicht, die Basisklasse auf *CFormView* umzustellen. Weitere Anpassungen sind nicht erforderlich. Nach Betätigen der Schaltfläche FERTIGSTELLEN zeigt Ihnen der Anwendungs-Assistent wie immer folgende Übersicht:

Dabei sind folgende Dateien automatisch vom Anwendungs-Assistenten generiert worden:

Dateiname	Bedeutung, Inhalt
Bibliothek.h Bibliothek.cpp	Implementierungs- und Schnittstellendatei für die Anwendung BIBLIOTHEK; in diesen Dateien ist die von <i>CWinApp</i> abgeleitete Applikationsklasse enthalten, zusätzlich ist noch die Info über ...-Dialogfeldklasse <i>CAboutDlg</i> integriert.
MainFrm.h MainFrm.cpp	In diesen Dateien ist die Klasse <i>CMainFrame</i> des Haupt-MDI-Rahmenfensters enthalten. Sie ist von <i>CMDIFrameWnd</i> abgeleitet.
ViewEinzel.cpp ViewEinzel.h	Hier befindet sich die Ansichtsklasse <i>CViewEinzel</i> der Anwendung. Sie hat <i>CFormView</i> als Basisklasse.
BibliothekDoc.cpp BibliothekDoc.h	Diese Dateien enthalten das Dokument <i>CBibliothekDoc</i> der Anwendung.
ChildFrm.cpp ChildFrm.h	Enthält die Klasse <i>CChildFrame</i> als Rahmenfenster des untergeordneten MDI-Fensters. Basisklasse ist <i>CMDIChildWnd</i> der MFC.
Bibliothek.rc Resource.h	Dies ist die Ressource-Datei zum Projekt mit allen bereits wichtigen Ressourcen wie Icon, Info über ...-Dialog, Toolbar, leere Dialogvorlage für <i>CViewEinzel</i> usw. In <i>Resource.h</i> sind die zugehörigen symbolischen Konstanten definiert. Weitere Ressourcen sind im Unterverzeichnis RES abgelegt.
StdAfx.h StdAfx.cpp	Enthält die Anweisungen zum Einbinden der Schnittstellendatei der MFC. Aus diesen wird die vorkompilierte Header-Datei <i>Bibliothek.pch</i> erzeugt, um den Erstellungsvorgang zu beschleunigen.
Bibliothek.clw	Binärdatei mit Informationen für den Klassen-Assistent.
ReadMe.txt	Eine Textdatei mit Informationen zu den angelegten Dateien, ähnlich wie in dieser Tabelle beschrieben.

Tabelle 13.6: Dateien des Projekts BIBLIOTHEK

Damit ist das Programmgerüst fertig, und Sie können das Projekt zum ersten Mal erstellen und testen (Step1). Nun geht es daran, die Funktionalität auszubauen.

13.8.5 Erweitern und Anpassen der Ressourcen

Dialogvorlagen

Im Beispiel BIBLIOTHEK werden zwei Dialogvorlagen für die formatierten Ansichten auf die Daten des Dokuments benötigt. Eine davon wurde als leere Maske durch den Anwendungs-Assistenten erzeugt. Die zweite Dialogvorlage erstellen Sie am einfachsten, indem Sie die Vorlage `IDD_BIBLIOTHEK_FORM` im *ResourceView*-Fenster des Arbeitsbereiches markieren und über BEARBEITEN|KOPIEREN und EINFÜGEN sozusagen verdoppeln. Dadurch stellen Sie sicher, daß die Dialogvorlage alle erforderlichen Eigenschaften hat, um für eine formatierte Ansicht als Vorlage zu dienen. Geben Sie der neuen Vorlage im *Eigenschaften*-Fenster den Namen `IDD_BIBLIOTHEK_LISTE`.

Abbildung 13.31: Dialogvorlage für Einzelsicht

Nun geht es an das Gestalten der beiden Dialogvorlagen. Die Vorlage `IDD_BIBLIOTHEK_FORM` wird nur Eingabefelder und eine Schaltfläche zur Datenanzeige und -erfassung enthalten. Die Schaltfläche dient zur Übernahme der editierten Daten in das Dokument (siehe Abbildung 13.31). Dabei werden folgende Symbolnamen für die einzelnen Steuerungen verwendet:

Steuerung	Symbolname
Lfd.Nr.	IDC_LFDNR
Titel	IDC_TITEL
Interpret	IDC_INTERPRET
Komponist	IDC_KOMPONIST
Datum	IDC_DATUM

Tabelle 13.7: Symbolnamen der Steuerungen

Steuerung	Symbolname
Zeit	IDC_ZEIT
Bemerkung	IDC_BEMERKUNG
Übernehmen	IDC_UEBERNEHMEN

Tabelle 13.7: Symbolnamen der Steuerungen

Die zweite formatierte Ansicht, `IDD_BIBLIOTHEK_LISTE`, enthält eine Übersicht in Form eines einfachen Listenfeldes mit entsprechenden Überschriften. Achten Sie darauf, daß beide Vorlagen nach Möglichkeiten die gleiche Größe (215x150 Pixel) haben. Für das Listenfeld wurden die Eigenschaften *Formate Sortieren* und *Keine Gesamthöhe* abweichend zur Vorbelegung gewählt (siehe Abbildung 13.32). Damit entspricht die Reihenfolge der Datensätze in der Listbox der tatsächlichen Abfolge (keine Sortierung), und die Höhe des Listenfeldes paßt sich automatisch vollen Zeileneinträgen an. Geben Sie der Steuerung den Symbolnamen `IDC_CD_LISTE`.

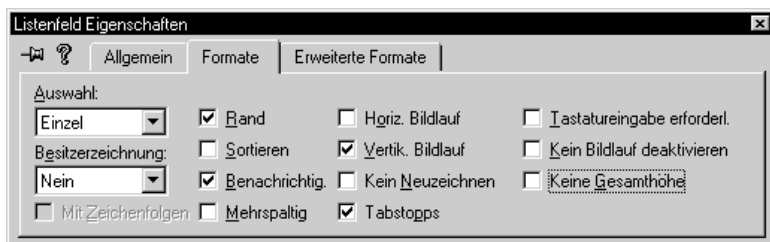


Abbildung 13.32: Listenfeld-Eigenschaften

Erweitern der Symbolleiste

Erweitern Sie die Standard-Symbolleiste mit Hilfe des Symbolleisten-Editors um vier Navigations-Schaltflächen sowie um eine zum Löschen des aktuellen Datensatzes und um eine zum Anhängen eines neuen Satzes.

Schaltfläche	Symbolname
Gehe zum ersten Datensatz	IDC_ANFANG
Gehe einen Satz zurück	IDC_ZURUECK
Gehe einen Satz vor	IDC_VOR
Gehe zum letzten Datensatz	IDC_ENDE
Datensatz Neu	IDC_NEU
Datensatz löschen	IDC_LOESCHEN

Tabelle 13.8: Symbolnamen der Schaltflächen

Das Ergebnis sollte dann etwa so aussehen:



Abbildung 13.33: Erweiterte Symbolleiste

Anpassen und Erweiterung des Menüs

Für das MDI-Projekt legt der Anwendungs-Assistent zwei Menü-Ressourcen an: Eine für das Hauptfenster der Anwendung und eine zweite für das MDI-Kind-Fenster.

Im Hauptmenü müssen Sie nicht viel ändern. Es ist nur notwendig, zwei Menütexte anzupassen, damit das Menü optisch besser der Anwendung angeglichen wird. Dazu öffnen Sie das Menü IDR_MAINFRAME im Menü-Editor und ändern die Texte unter DATEI von NEU in NEUE CD-BIBLIOTHEK und von ÖFFNEN in CD BIBLIOTHEK ÖFFNEN ab. Die zugehörigen Symbolnamen bleiben unverändert.

Im Menü des MDI-Kind-Fensters mit dem Symbolnamen IDR_CDTYPE muß für das Datei-Menü entsprechend verfahren werden. Darüber hinaus muß das BEARBEITEN-Menü neu gestaltet werden. Löschen Sie zunächst die Menüfelder unter BEARBEITEN. Fügen Sie dann analog zur Symbolleiste die Menüfelder zum Navigieren sowie für NEU und LÖSCHEN ein. Verwenden Sie dieselben Symbolnamen. Das Menü sollte nun so aussehen:

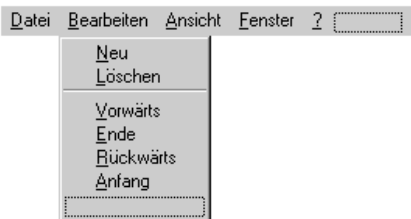


Abbildung 13.34: Menü BEARBEITEN des MDI-Kind-Fensters

Um das Menü zu perfektionieren, können Sie zu jedem Menüpunkt im *Menübefehl-Eigenschaften*-Fenster einen Statuszeilentext hinterlegen.



Darüber hinaus wird das Menü ANSICHT des MDI-Kind-Fensters erweitert. Um zwischen den beiden Ansichten bzw. den Dokumentvorlagen hin- und her- schalten zu können, bekommt jede von ihnen einen Menüpunkt zum Aktivieren:



Abbildung 13.35: Menü ANSICHT des MDI-Kind-Fensters

Vergeben Sie die Symbolnamen `ID_CD_UEBERSICHT` und `ID_EINZEL_CD` für die Menüpunkte.

Sonstiges

Wie in den anderen Beispielen auch, können Sie hier das Icon der Anwendung und des Dokumentes nach Ihren Vorstellungen anpassen. Außerdem sollten Sie sich die Dialogvorlage des *Info über ...*-Dialogs ansehen und gegebenenfalls an Ihre Wünsche angleichen. Jetzt wäre es auch an der Zeit, sich anzuschauen, was sich hinter dem Symbolnamen `VS_VERSION_INFO` verbirgt.

Erstellen Sie das Projekt erneut (Step2), und prüfen Sie die Funktionalität und vor allem das Aussehen der geänderten Ressourcen.

13.8.6 Zusätzlicher Programmcode

Die zweite Ansichtsklasse

Da der Anwendungs-Assistent standardmäßig nur eine Ansichtsklasse anlegt, müssen Sie nun die zweite mit Hilfe des Klassen-Assistenten anlegen. Der Assistent zum Anlegen neuer Klassen hat – je nachdem, wie er aufgerufen wird – ein unterschiedliches Erscheinungsbild und eine geringfügige andere Funktionalität. Beim Aufruf über den MFC Klassen-Assistenten, wie im Abschnitt »*Dialogfeld-Klasse mit dem Klassen-Assistenten*« beschrieben, können nur Klassen angelegt werden, die auch der Klassen-Assistent unterstützt, d.h. Klassen, die von `CComdTarget` abgeleitet sind. Beim Aufruf über das Hauptmenü `EINFÜGEN|NEUE KLASSE` oder über das Menü der rechten Maustaste im Klassen-Fenster des Arbeitsbereiches, können Sie im Dialog *Neue Klasse* zusätzlich den Klassentyp wählen.

Für das Erweitern unseres Beispiels ist es der Typ *Formularklasse*. Tragen Sie unter *Formularinformationen* die erforderlichen Angaben ein. Die für die Klasse `CFormView` benötigte Dialogvorlage steht bereits fertig zur Verfügung.

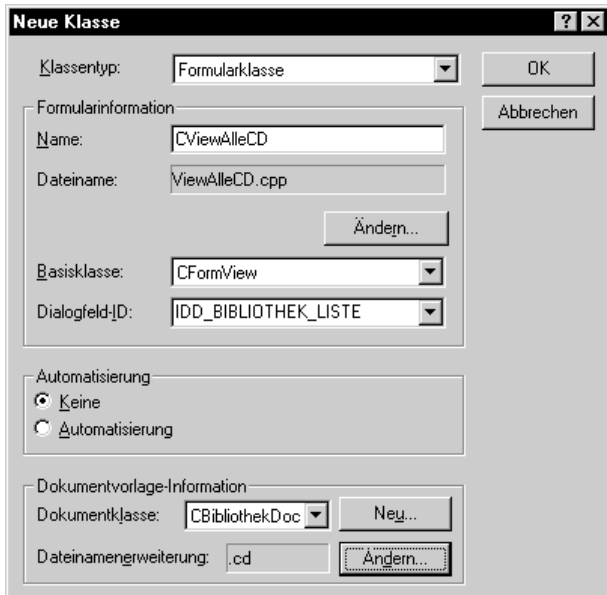


Abbildung 13.36: Neue Ansichtsklasse für BIBLIOTHEK

Unter *Dokumentvorlage-Informationen* ist die Dokumentenklasse auszuwählen. Da es nur die Klassen *CBibliothekDoc* vom Typ *CDocument* in diesem Beispiel gibt, ist keine Auswahl möglich. Interessant ist dieser Punkt allerdings für Erweiterungen mit neuen Dokumenten für zusätzliche Bibliotheken.

Der Klassen-Assistent legt für Sie nicht nur die komplette Klasse für die Ansicht an, sondern erweitert gleich die Methode *InitInstance* von *CBibliothekApp* um eine zweite Dokumentvorlage. Außerdem erhält die Implementierungsdatei *ViewAlleCD.cpp* der Ansichtsklasse *CViewAlleCd* einen Eintrag zur Schnittstellendatei der Dokumentenklasse *#include »BibliothekDoc.h«*.

```
{ // BLOCK: Registrierung der Dokumentvorlage
    // Registrieren der Dokumentvorlage. Dokumentvorlagen dienen
    // als Verbindung zwischen Dokumenten, Rahmenfenstern und
    // Ansichten.
    // Verbinden Sie dieses Formular mit einem anderen Dokument
    // oder Rahmenfenster, indem Sie
    // das Dokument oder die Rahmenklasse im unten folgenden
    // Konstruktor ändern.
    CMultiDocTemplate* pNewDocTemplate=new CMultiDocTemplate(
        IDR_VIEWALLECD_TMPL,
```

```

        RUNTIME_CLASS(CBibliothekDoc), // Dokumentklasse
        RUNTIME_CLASS(CMDIChildWnd), // Rahmenklasse
        RUNTIME_CLASS(CViewAllCD)); // Ansichtsklasse
AddDocTemplate(pNewDocTemplate);
}

```

Listing: Anlage der Dokumentvorlage durch den Klassen-Assistenten

Die CD-Klasse

Nun ist mal wieder Handarbeit angesagt. Beim Erstellen einer Klasse, die die Daten einer CD repräsentieren soll, kann Ihnen der Klassen-Assistent auch gute Dienste leisten. Der Klassen-Assistent unterstützt einen weiteren Klassentyp: *Allgemeine Klassen*. Für das Anlegen der CD-Klasse benötigen Sie *CObject* als Basisklasse. Der Unterschied bei diesem Klassentyp besteht für den Klassen-Assistenten darin, daß er für dieses Objekt keine Nachrichten verwalten kann.

Für die Neuanlage wählen Sie nun im Dialogfeld *Neue Klasse* den Klassentyp *Allgemeine Klasse* aus. Dadurch verändert sich die Auswahlmöglichkeit im Dialog erneut (siehe Abbildung 13.37). Sie können Ihre Klassen hier von beliebigen anderen Klassen ableiten oder auch von keiner, also ganz ohne Basisklasse. Ergänzen Sie die Eingabefelder wie folgt:

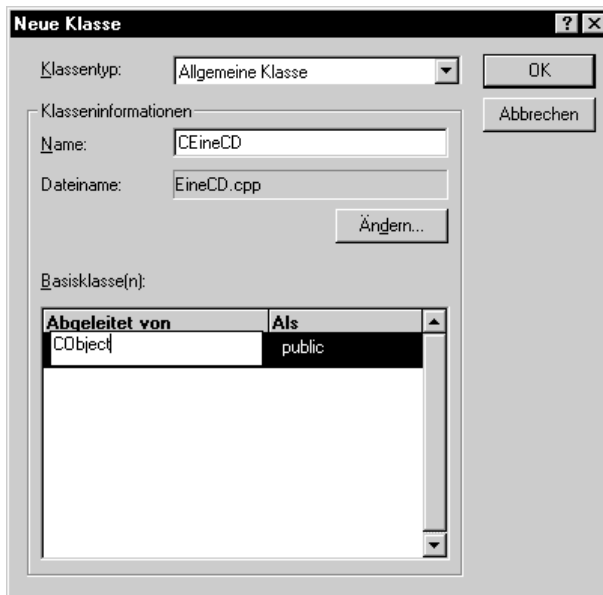


Abbildung 13.37: Neue »Allgemeine Klasse«

Nach dem Bestätigen mit der OK-Schaltfläche teilt Ihnen der Klassen-Assistent mit, daß er keine entsprechende Schnittstellendatei gefunden habe und fragt, ob Sie die Klasse trotzdem anlegen wollen. Im Projekt BIBLIOTHEK ist die Klasse *CObject* über die Schnittstellendatei *Stdafx.h* bereits bekannt. Sie können diesen Hinweis also übergehen.

Legen Sie nun die eigentlichen Datenmember für die Klasse über *Member-Variable hinzufügen* an. Die Variablen entsprechen den Steuerungen aus der Ansichtsklasse *CEinzelView*. Die Variablen sind alle vom Typ *CString* und haben den Zugriffsstatus *public*.

```
class CEineCD : public CObject
{
public:
    CString    m_strLfdNr;
    CString    m_strTitel;
    CString    m_strInterpret;
    CString    m_strKomponist;
    CString    m_strDatum;
    CString    m_strZeit;
    CString    m_strBemerkung;
    CEineCD();
    virtual ~CEineCD();
};
```

Listing: Der Prototyp der Klasse CEineCD

Die wichtigste Eigenschaft der Klasse fehlt aber noch. Sie soll sich über die Serialisation auf den Datenträger schreiben und von dort auch wieder eingelesen werden können. Dazu müssen Sie die Methode *Serialize* der Basisklasse *CObject* überschreiben. Da hier der Klassen-Assistent versagt, fügen Sie diese virtuelle Funktion über *Member-Funktion hinzufügen* der Klasse hinzu.



Abbildung 13.38: Funktion hinzufügen

Da auch die Klasse *CString* die *Serialize*-Methode beherrscht, ist das Programmieren der Funktion für die Klasse *CEineCD* eine Kleinigkeit für Sie:

```
void CEineCD::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        // ZU ERLEDIGEN: Hier Code zum Speichern einfügen
        ar << m_strLfdNr;
        ar << m_strTitel;
        ar << m_strInterpret;
        ar << m_strKomponist;
        ar << m_strDatum;
        ar << m_strZeit;
        ar << m_strBemerkung;
    }
    else
    {
        // ZU ERLEDIGEN: Hier Code zum Laden einfügen
        ar >> m_strLfdNr;
        ar >> m_strTitel;
        ar >> m_strInterpret;
        ar >> m_strKomponist;
        ar >> m_strDatum;
        ar >> m_strZeit;
        ar >> m_strBemerkung;
    }
}
```

Listing: Serialisation der Klasse *CEineCD*

Bist jetzt kann die Klasse *CEineCD* ihre Daten in Form von *CString*-Variablen speichern und wieder einlesen, doch das gilt noch nicht für die Klasse selbst. Um *CEineCD* komplett zur Serialisation tauglich zu machen, müssen Sie noch zwei Makros der MFC einsetzen, die eine Verbindung zur Klasse *CArchive* für Sie herstellt (befreundete Klasse). In die Schnittstellen-datei fügen Sie `DECLARE_SERIAL` und in die Implementierungsdatei `IMPLEMENT_SERIAL` ein (siehe Kapitel 13.2.3 »Verwaltung der Daten«).

```
#define DECLARE_SERIAL(class_name) \
    _DECLARE_DYNCREATE(class_name) \
    friend CArchive& AFXAPI operator>>(CArchive& ar,class_name* \
    &pOb);
```



```
#define IMPLEMENT_SERIAL(class_name, base_class_name, wSchema) \
    CObject* PASCAL class_name::CreateObject() \
    { return new class_name; } \
    _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, wSchema, \
    class_name::CreateObject) \
    static const AFX_CLASSINIT _init_##class_name( \
        RUNTIME_CLASS(class_name)); \
    CArchive& AFXAPI operator>>(CArchive& ar, class_name* &pOb) \
    { pOb = (class_name*)ar.ReadObject(RUNTIME_CLASS(class_name)); \
    return ar; } \
```

Die Anwendung dieser Makros ist sehr viel einfacher als der Programmcode, der dahintersteckt. In der Schnittstellendatei tragen Sie das Makro `DECLARE_SERIAL(CEineCD)` innerhalb der Klassendeklaration noch vor dem Konstruktor ein. Das Makro `IMPLEMENT_SERIAL(CEineCD, CObject, 0)` setzen Sie in der Implementierungsdatei noch vor dem Konstruktor ein.

Damit ist die Klasse für die Datenmember einsatzbereit. Erstellen Sie das Projekt neu, um Tippfehler zu finden.

Aktivieren Sie an dieser Stelle Browse-Informationen. Am einfachsten geht dies durch einmaliges Drücken der `[F12]`-Taste. Wenn die Option noch nicht aktiv ist, fragt Sie das Developer Studio, ob Sie die Option aktivieren möchten.



Um den Zugriff auf die neue Klasse zu ermöglichen, müssen Sie diese in den einzelnen Implementierungsdateien des Projektes bekanntmachen. Am besten geht es in diesem Fall durch das Includieren der Schnittstellendatei in `StdAfx.h` (`#include »EineCD.h«`).

Zugriff auf Steuerungen

In der `CViewEinzel` werden die angezeigten oder eingegebenen Daten in lokalen Variablen zwischengespeichert. Dazu ist es notwendig, die Eingabefelder der Dialogvorlage mit entsprechenden Variablen zu verbinden. Da in der Daten-Klasse `CEineCD` die Variablen vom Typ `CString` sind, bietet es sich an, auch hier diese Basisklasse zu verwenden. Dafür kommt der Klassen-Assistent zum Einsatz. Verbinden Sie alle Eingabefelder wie folgt:

Bei dieser Gelegenheit können Sie die maximale Eingabelänge festlegen (*Maximale Anzahl von*). Das Programmgerüst stellt eine Prüfungsfunktion zur Verfügung, die vor jeder Datenübernahme die Länge der Felder bestimmt und bei Überschreitung einen Hinweis ausgibt. Verbinden Sie die Schaltfläche `ÜBERNEHMEN` mit der hinzugefügten Funktion `OnUebernehmen`.

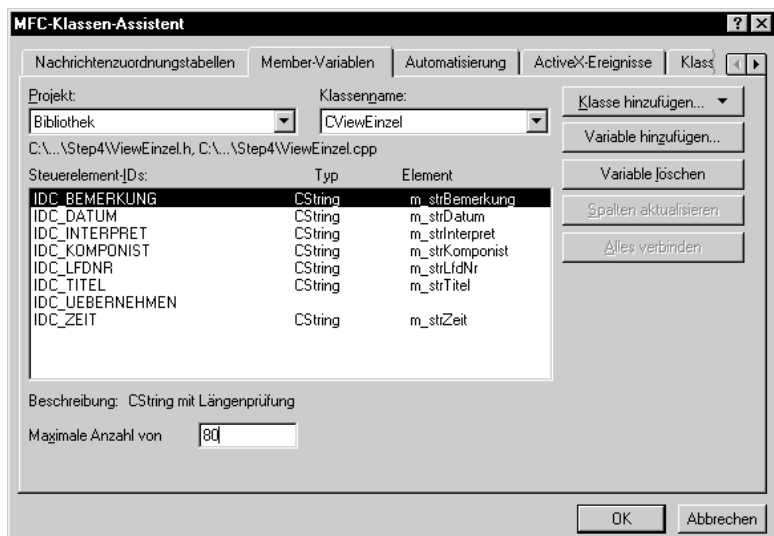


Abbildung 13.39: Variablen hinzufügen

In der Schnittstellendatei der Klasse *CviewEinzel* – *ViewEinzel.h* – werden die Variablen vom Klassen-Assistenten automatisch implementiert.

```
class CViewEinzel : public CFormView
{
    //...
public:
    //{AFX_DATA(CViewEinzel)
    enum { IDD = IDD_BIBLIOTHEK_FORM };
    CString m_strBemerkung;
    CString m_strInterpret;
    CString m_strKomponist;
    CString m_strLfdNr;
    CString m_strTitel;
    CString m_strZeit;
    CString m_strDatum;
    //}AFX_DATA
    //...
}
```

Listing: Variablendeklaration in der Schnittstellendatei

Die Initialisierung erfolgt dann im Konstruktor der Klasse *CViewEinzel* in der Implementationsdatei *ViewEinz.cpp*. Dabei kann auch eine Vorbelegung vorgenommen werden.

```

CViewEinzel::CViewEinzel()
: CFormView(CViewEinzel::IDD)
{
    //{AFX_DATA_INIT(CViewEinzel)
    m_strBemerkung = _T("");
    m_strInterpret = _T("");
    m_strKomponist = _T("");
    m_strLfdNr = _T("");
    m_strTitel = _T("");
    m_strZeit = _T("");
    m_strDatum = _T("");
    //}}AFX_DATA_INIT
    // ZU ERLEDIGEN: Hier Code zur Konstruktion einfügen,
}

```

Listing: Variableninitialisierung im Konstruktor der Klasse

Die eigentliche Verbindung zwischen den Variablen und den Steuerungen erfolgt wieder über die Methode *DoDataExchange* der Klasse. An dieser Stelle wird auch die oben erwähnte Überprüfung der Länge des eingegebenen Textes vorgenommen.

```

void CViewEinzel::DoDataExchange(CDataExchange* pDX)
{
    CFormView::DoDataExchange(pDX);
    //{AFX_DATA_MAP(CViewEinzel)
    DDX_Text(pDX, IDC_BEMERKUNG, m_strBemerkung);
    DDV_MaxChars(pDX, m_strBemerkung, 80);
    DDX_Text(pDX, IDC_INTERPRET, m_strInterpret);
    DDV_MaxChars(pDX, m_strInterpret, 25);
    DDX_Text(pDX, IDC_KOMPONIST, m_strKomponist);
    DDV_MaxChars(pDX, m_strKomponist, 25);
    DDX_Text(pDX, IDC_LFDNR, m_strLfdNr);
    DDV_MaxChars(pDX, m_strLfdNr, 3);
    DDX_Text(pDX, IDC_TITEL, m_strTitel);
    DDV_MaxChars(pDX, m_strTitel, 20);
    DDX_Text(pDX, IDC_ZEIT, m_strZeit);
    DDV_MaxChars(pDX, m_strZeit, 5);
    DDX_Text(pDX, IDC_DATUM, m_strDatum);
    DDV_MaxChars(pDX, m_strDatum, 10);
    //}}AFX_DATA_MAP
}

```

Listing: Verbindung der Variablen mit den Steuerungen

Für die zweite Ansichtsklasse ist der Aufwand wesentlich geringer. Die Dialogvorlage enthält nur eine aktive Steuerung, das Listenfeld IDC_CD_LISTE. Verbinden Sie dieses mit Hilfe des Klassen-Assistenten mit einer Variablen vom Typ *CListBox* und geben ihr den Namen *m_lbListe*. Weitere Einstellungen sind hier nicht erforderlich.

Initialisierung der Ansichtsklassen

Um eine formatierte Ansicht von Anfang an in der vollen Größe darzustellen, stellt die MFC die Methode *ResizeParentToFit* der Klasse *CScrollView* zur Verfügung. Dadurch wird die Fenstergröße der im Dialog-Editor definierten Größe der Dialogvorlage angepaßt. Dazu fügen Sie die Methode *OnInitialUpdate* in beide Ansichtsklassen ein und ergänzen diese um *ResizeParentToFit*.

```
void CViewEinzel::OnInitialUpdate()
{
    CFormView::OnInitialUpdate();
    ResizeParentToFit(FALSE);
}
```

Listing: Anpassen der Ansichtgröße durch ResizeToFit

Für das Listenfeld von *CViewAlleCD* müssen Sie die Tabstop-Position anpassen. Dadurch richten sich die Texte für den INTERPRETEN immer an dieser Position aus, gleichgültig, wie lang der Titel eingegeben wurde. Die Position läßt sich in diesem Fall leicht errechnen, da nur ein Tabstop für das Listenfeld benötigt wird. Nehmen Sie die horizontale Position des Textes INTERPRET in der Dialogvorlage (hier 100 Punkte), und ziehen Sie die horizontale Position des Listenfeldes (hier 7 Punkte) davon ab. Diese Werte können Sie z.B. im Dialogeditor in der Statuszeile ablesen.

```
void CViewAlleCD::OnInitialUpdate()
{
    CFormView::OnInitialUpdate();
    m_lbListe.SetTabStops(93);
    ResizeParentToFit(FALSE);
}
```

Listing: Setzen des Tabulators in dem Listenfeld

Dokument erweitern

Die Klasse *CEineCD* enthält die Informationen für genau einen Datensatz. In der Anwendung sollen aber mehrere Datensätze erfaßt werden können. Diese Daten werden am besten in einem Array abgelegt. Legen Sie also im Dokument eine öffentliche Instanz der MFC-Klasse *CObArray* an. Sie verwaltet dann die einzelnen Objekte von *CEineCD*. Darüber hinaus wird

noch ein Satzzeiger benötigt, der die aktuell angezeigte Position repräsentiert. Dafür reicht eine Variable vom Typ *int*. Initialisieren Sie `m_nPos` im Konstruktor mit »-1«.

```
class CBibliothekDoc : public CDocument
{
    //...
public:
    int m_nPos;
    CObArray m_aCDListe;
    //...
}
```

Listing: Zusätzliche Variablen im Dokument

Da *Cdocument* – und dadurch auch *CbibliothekDoc* – von *CCmdTarget* abgeleitet ist, kann diese Klasse auch auf Menüfeld-Eingaben reagieren. In diesem Beispielprogramm soll nun nicht die Ansichtsklasse oder das Rahmenfenster, sondern das Dokument die notwendigen Funktionen aufnehmen. Verbinden Sie den Menü-Symbolnamen `IDR_CDTYPE` mit der Klasse *CBibliothekDoc*. Legen Sie mit dem Klassen-Assistenten für alle Menüpunkte unter BEARBEITEN eine entsprechende Methode im Dokument an. Mit wenigen Maus-Klicks ist das erledigt.

```
class CBibliothekDoc : public CDocument
{
    //...
    // Generierte Message-Map-Funktionen
protected:
    //{AFX_MSG(CBibliothekDoc)
    afx_msg void OnAnfang();
    afx_msg void OnEnde();
    afx_msg void OnVor();
    afx_msg void OnZurueck();
    afx_msg void OnNeu();
    afx_msg void OnLoeschen();
    //}AFX_MSG
    DECLARE_MESSAGE_MAP()
    //...
};
```

Listing: Funktionen vom Menü zur Navigation

Jetzt geht es an das Ausprogrammieren dieser Funktionen. Zuerst die Navigation in den Datensätzen bzw. innerhalb des Arrays: In diesen Funktionen muß nur der Satzzeiger `m_nPos` entsprechend neu belegt werden. Da-

bei ist der Gültigkeitsbereich wichtig, der durch die Größe von `m_aCDListe` vorgegeben ist. Nach der Veränderung des Satzzeigers müssen Sie nur noch die Ansichtsklassen von dieser Veränderung unterrichten. Dabei kommt die bereits bekannte Methode *UpdateAllViews* zum Einsatz.

```
void CBibliothekDoc::OnAnfang()
{
    if (m_nPos > 0)
    {
        m_nPos = 0;
        UpdateAllViews(NULL, m_nPos);
    }
}

void CBibliothekDoc::OnEnde()
{
    if (m_nPos < m_aCDListe.GetUpperBound())
    {
        m_nPos = m_aCDListe.GetUpperBound();
        UpdateAllViews(NULL, m_nPos);
    }
}

void CBibliothekDoc::OnVor()
{
    if (m_nPos < m_aCDListe.GetUpperBound())
    {
        m_nPos++;
        UpdateAllViews(NULL, m_nPos);
    }
}

void CBibliothekDoc::OnZurueck()
{
    if (m_nPos > 0)
    {
        m_nPos--;
        UpdateAllViews(NULL, m_nPos);
    }
}
```

Listing: Die ausprogrammierten Navigationsfunktionen

Die Funktionen *OnNeu* und *OnLoeschen* sind ähnlich einfach zu realisieren. Bei *OnNeu* müssen Sie nur eine neue Instanz von *CEineCD* mit *new* dynamisch erzeugen und an das Array hinzufügen; in *OnLoeschen* ist es genau umgekehrt. Das aktuelle Element wird aus dem Array entfernt und die Instanz von *CEineCD* mit *delete* freigegeben. Vor dem Löschen stellen Sie

dem Bediener eine Sicherheitsabfrage durch eine *MessageBox*. Mit *SetModifiedFlag(TRUE)* merkt sich das Dokument, daß sich die Daten seit dem letzten Sichern geändert haben.

```
void CBibliothekDoc::OnNeu()
{
    CEineCD* tmp_CD = new CEineCD();
    m_nPos = m_aCDListe.Add(tmp_CD);
    UpdateAllViews(NULL, m_nPos);
    SetModifiedFlag(TRUE);
}

void CBibliothekDoc::OnLoeschen()
{
    if (m_aCDListe.GetUpperBound() > 0)
    {
        if (AfxMessageBox("Titel wirklich löschen?",
                          MB_YESNO) == IDYES)
        {
            delete m_aCDListe.GetAt(m_nPos);
            m_aCDListe.RemoveAt(m_nPos);
            m_nPos = 0;
            UpdateAllViews(NULL);
        }
    }
}
```

Listing: Neuanlage und Löschen eines Datenobjektes

Jetzt fehlt nur noch das Initialisieren und die Endebehandlung. Wenn das Programmgerüst die Nachricht *ID_FILE_NEW* bekommt, ruft es die Methode *OnFileNew* auf, die wiederum unter anderem die Funktion *OnNewDocument* des Dokumentes aktiviert. Sie müssen in diesem Fall dem Bediener einen neuen Datensatz der Klasse *CEineCD* zur Verfügung stellen. Rufen Sie dazu in der Funktion *OnNewDocument* der Klasse *CBibliothekDoc* einfach die Methode *OnNeu* auf, die wie oben beschrieben das Notwendige tut.

```
BOOL CBibliothekDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;
    OnNeu();
    // (SDI-Dokumente verwenden dieses Dokument)
    return TRUE;
}
```

Listing: Anlegen eines neuen Dokuments

Durch die Nachrichten `ID_FILE_OPEN` und `ID_FILE_SAVE` wird letztlich durch das Programmgerüst die *Serialize*-Methode des Dokumentes aufgerufen. Hier werden dann die einzelnen Objekte der Klasse *CEineCD* auf den Datenträger geschrieben bzw. die Daten wieder in das Array übernehmen. Da Sie die Klasse *CEineCD* bereits mit einer *Serialize*-Methode versehen haben, beschränkt sich der Aufwand auf den Aufruf von *Serialize* für das Array `m_aCDListe`. Alles andere geht jetzt wie von selbst. Der Mechanismus um die Methode *Serialize* zeigt, wie einfach programmieren werden kann, wenn man objektorientiert arbeitet und dabei die MFC richtig nutzt. Es empfiehlt sich, nach erfolgreichem Einlesen, also wenn mindestens ein Datenobjekt eingelesen wurde, den Positionszähler `m_nPos` neu zu initialisieren.

```
void CBibliothekDoc::Serialize(CArchive& ar)
{
    m_aCDListe.Serialize(ar);
    if (ar.IsStoring())
    {
        // ZU ERLEDIGEN: Hier Code zum Speichern einfügen
    }
    else
    {
        if (m_aCDListe.GetSize() > 0)
            m_nPos = 0;
    }
}
```

Listing: Das Anwenden der Methode *Serialize* für das Objekt-Array

Beim Schließen des Dokumentes müssen Sie die dynamisch angelegten *CEineCD*-Objekte auch wieder aus dem Speicher entfernen. Alle weiteren Aufgaben wie Bedienerabfrage und Sichern übernimmt das Programmgerüst für Sie.

```
CBibliothekDoc::~~CBibliothekDoc()
{
    for (int i = 0; i < m_aCDListe.GetSize(); i++)
        delete m_aCDListe.GetAt(i);
    m_aCDListe.RemoveAll();
}
```

Listing: Freigabe aller Datenobjekte im Destruktor

In diesem Beispiel kommt die Methode *DeleteContents* des Dokumentes nicht direkt zum Einsatz, da sich die Daten im gesamten Lebenszyklus im Dokument befinden. Erstellen Sie das Projekt neu, und korrigieren Sie eventuelle Tippfehler.

Aktualisieren der Ansichtsklassen

Die Funktion *OnUpdate* dient auch in diesem Beispiel zum Aktualisieren der lokalen Daten und der Steuerelemente der Ansichten. Fügen Sie die Methode mit dem Klassen-Assistenten in die beiden Ansichtsklassen ein.

Für *CViewEinzel* bedeutet das, den aktuellen Datensatz aus dem Dokument zu holen und in den lokalen Variablen zu speichern. Dabei wird ein temporärer Zeiger auf das aktuelle Datenelement angelegt. Anschließend werden die Steuerelemente über die Methode *UpdateData* aktualisiert und letztlich zur Anzeige gebracht.

```
void CViewEinzel::OnUpdate(CView* pSender, LPARAM lHint, CObject*
pHint)
{
    CEineCD* pCD =
        (CEineCD*)GetDocument()->m_aCDListe.GetAt(lHint);
    m_strBemerkung = pCD->m_strBemerkung;
    m_strInterpret = pCD->m_strInterpret;
    m_strKomponist = pCD->m_strKomponist;
    m_strLfdNr = pCD->m_strLfdNr;
    m_strTitel = pCD->m_strTitel;
    m_strZeit = pCD->m_strZeit;
    m_strDatum = pCD->m_strDatum;
    UpdateData(FALSE);
}
```

Listing: Aktualisierung der lokalen Daten der Ansichtsklasse und Neuanzeige der Daten

Bei *CViewAlleCD* wird unterschieden, ob nur die derzeitige Position aktualisiert werden muß, oder ob das gesamte Listenfeld neu gefüllt werden soll, wenn Datenobjekte gelöscht oder dazugekommen sind. Zum Aufbau der Zeile für das Listenfeld wird die lokale Methode *PosString* verwendet. Sie holt sich den entsprechenden Datensatz aus dem Dokument und setzt daraus die Zeile für das Listenfeld zusammen. Dabei werden Titel und Interpret durch einen Tabulator (*\t*) getrennt, was die formatierte Ausgabe im Listenfeld zur Folge hat. Die fertige Zeichenkette ist gleichzeitig der Rückgabewert dieser Funktion. Da *CViewAlleCD* mit dem Klassen-Assistenten zusätzlich angelegt wurde, ist die Methode *GetDocument* nicht überlagert worden. Deshalb müssen Sie eine Typwandlung des Zeigers beim Zugriff auf das Dokument einfügen:

```
CString CViewAlleCD::PosString(int Pos)
{
    CString strTemp;
    CEineCD* pCD = (CEineCD*)
        ((CBibliothekDoc*)GetDocument())->m_aCDListe.GetAt(Pos);
```

```

// Zeile für Listenfeld zusammensetzen
strTemp = pCD->m_strTitel;
strTemp += '\t';
strTemp += pCD->m_strInterpret;

return strTemp;
}

```

Listing: Funktion zum Zusammensetzen einer Zeile für das Listenfeld

Jetzt ist die Methode *OnUpdate* für die Ansichtsklasse *CViewAlleCD* schnell realisiert: Der Parameter *lHint* steuert, ob nur ein Eintrag oder die ganze Liste aktualisiert werden soll.

```

void CViewAlleCD::OnUpdate(CView* pSender, LPARAM lHint,
                          CObject* pHint)
{
    if (!lHint)
    {
        m_lbListe.ResetContent();
        for (int i=0;
             i < ((CBibliothekDoc*)GetDocument())->
                                     m_aCDListe.GetSize(); i++)
        {
            m_lbListe.AddString(PosString(i));
        }
    }
    else
    {
        m_lbListe.DeleteString(lHint);
        m_lbListe.InsertString(lHint, PosString(lHint));
    }
    // aktuelle Zeile auswählen
    m_lbListe.SetCurSel(lHint);
}

```

Listing: Aktualisierung der Ansichtsklasse mit dem Listenfeld

Damit das Beispielprogramm auch anspruchsvollen Benutzern gerecht wird, soll nun auch eine Synchronisation zwischen den Ansichtsklassen ermöglicht werden. Jedesmal, wenn der Bediener im Listenfeld einen Datensatz selektiert, soll in *CViewEinzel* auch der entsprechende Datensatz angezeigt werden. Dazu muß die Nachricht *LBN_SELCHANGE* des Listenfeldes durch Klassen-Assistenten mit einer Funktion in *CViewAlleCD* verbunden werden. Diese wird dann vom Programmgerüst jedesmal ausgeführt, wenn ein anderer Eintrag im Listenfeld markiert wird. Ihre Aufgabe

besteht jetzt nur noch darin, das dem Dokument und der zweiten Ansichtsklasse mitzuteilen:

```
void CViewAlleCD::OnSelchangeCdListe()
{
    ((CBibliothekDoc*)GetDocument()->m_nPos =
        m_lbListe.GetCurSel();
    GetDocument()->UpdateAllViews(this, m_lbListe.GetCurSel());
}
```

Listing: Abfangen der Änderung im Listenfeld

Übernahme von Änderungen

Für den umgekehrten Weg, der Aktualisierung der Daten im Dokument, implementieren Sie die Methode *OnUebernehmen*, die ausgeführt wird, wenn der Bediener die Schaltfläche ÜBERNEHMEN in *CViewAlleCD* betätigt. Dabei müssen die Daten zuerst von den Steuerelementen durch die Methode *UpdateData* in die lokalen Variablen übernommen werden. An dieser Stelle erfolgt eine kurze syntaktische Prüfung der Daten. Es sollte mindestens der Titel eingegeben werden, bevor die Daten übernommen werden. Dazu wird gegebenenfalls wieder ein Hinweis durch eine *AfxMessageBox* ausgegeben:

```
void CViewEinzel::OnUebernehmen()
{
    UpdateData(TRUE);
    if (m_strTitel.IsEmpty())
    {
        AfxMessageBox("Bitte Titel eingeben!");
        GetDlgItem(IDC_TITEL)->SetFocus();
        return;
    }
    CEineCD* pCD = (CEineCD*)
        GetDocument()->m_aCDListe.GetAt(GetDocument()->m_nPos);
    pCD->m_strBemerkung = m_strBemerkung;
    pCD->m_strInterpret = m_strInterpret;
    pCD->m_strKomponist = m_strKomponist;
    pCD->m_strLfdNr = m_strLfdNr;
    pCD->m_strTitel = m_strTitel;
    pCD->m_strZeit = m_strZeit;
    pCD->m_strDatum = m_strDatum;
    GetDocument()->m_aCDListe.SetAt(GetDocument()->m_nPos, pCD);
    GetDocument()->UpdateAllViews(this, GetDocument()->m_nPos);
    GetDocument()->SetModifiedFlag(TRUE);
}
```

Listing: Übernahme der geänderten Daten in das Datenobjekt des Dokuments

Anschließend werden die lokalen Daten an den aktuellen Datensatz des Dokuments übergeben und die andere Ansicht über die Änderung durch die Methode *UpdateAllViews* des Dokuments informiert. Zum Schluß wird dem Dokument über die Methode *SetModifiedFlag* noch mitgeteilt, daß sich die Daten seit dem letzten Sichern geändert haben: Beim Schließen der Dokumentvorlage kommt ein entsprechender Hinweis.

Anpassung im Hauptprogramm

Die Funktionalität zum Bearbeiten mehrerer Dokumentvorlagen erfordert ein wenig Aufwand. Wenn im Programm mehrere Vorlagen über *AddDocTemplate* in die Liste der aktiven Dokumentvorlagen – *m_templateList* – aufgenommen wurden, erscheint bei jedem DATEI|ÖFFNEN oder DATEI|NEU eine Auswahl dieser Vorlagen. Um das Verhalten zu umgehen, müssen die entsprechenden Methoden überlagert werden. Außerdem ist es erforderlich, für jede Dokumentvorlage der Anwendung einen eigenen Zeiger zu haben, der auch außerhalb von *InitInstance* der Anwendung bekannt ist. Also legen Sie zwei entsprechende Zeiger in *CBibliothekApp* an

```
class CBibliothekApp : public CWinApp
{
public:
    CMultiDocTemplate* pDocTemplateAlle;
    CMultiDocTemplate* pDocTemplateEinzel;
    // ...
}
```

Listing: Zwei Zeiger auf zwei unterschiedliche Dokumentvorlagen

Der Anwendungs-Assistent hat die erste Dokumentvorlage bereits beim Anlegen des Projektes angelegt. Die zweite Dokumentvorlage wurde durch den Klassen-Assistenten beim Bereitstellen der zweiten Formularklasse in *InitInstance* zur Verfügung gestellt. Da die Dokumentvorlagen dort nur auf lokalen Zeigern angelegt und in die Liste der Vorlagen eingetragen wurden, müssen Sie den automatisch generierten Programmcode in *InitInstance* wie folgt ändern:

```
pDocTemplateEinzel = new CMultiDocTemplate(
    IDR_CDTYPE,
    RUNTIME_CLASS(CBibliothekDoc),
    RUNTIME_CLASS(CChildFrame), // Benutzerspezifischer MDI...
    RUNTIME_CLASS(CViewEinzel));
AddDocTemplate(pDocTemplateEinzel);
pDocTemplateAlle = new CMultiDocTemplate(
    IDR_CDTYPE,
    RUNTIME_CLASS(CBibliothekDoc),
```

```

    RUNTIME_CLASS(CChildFrame), // Benutzerspezifischer MDI...
    RUNTIME_CLASS(CViewAllCD));
AddDocTemplate(pDocTemplateAlle);

```

Listing: Anpassung in InitInstance

Damit das Programmgerüst nicht versucht, eine Dokumentvorlage zu öffnen, entfernen Sie den Aufruf von *ProcessShellCommand* in *InitInstance*. Dadurch sind Sie jetzt aber selbst für das Öffnen der entsprechenden Dokumentvorlage verantwortlich. Dazu müssen, wie oben bereits erwähnt, die Methoden *OnFileNew* und *OnFileOpen* in der Klasse *CBibliothekApp* abgeleitet bzw. überschrieben werden. Dort wird dann mit der entsprechenden Dokumentvorlage die Methode *OpenDocumentFile* ausgeführt.

```

void CBibliothekApp::OnFileNew()
{
    pDocTemplateEinzel->OpenDocumentFile(NULL);
}

void CBibliothekApp::OnFileOpen()
{
    CString strName;
    // Datei Öffnen-Dialog
    if (!DoPromptFileName(strName, AFX_IDS_OPENFILE,
                          OFN_HIDEREADONLY | OFN_FILEMUSTEXIST,
                          TRUE, pDocTemplateAlle))
        return;
    pDocTemplateAlle->OpenDocumentFile(strName);
}

```

Listing: Überlagerte Methoden zum Öffnen und Neuanlegen eines Dokuments

Erstellen Sie die Anwendung erneut, und testen Sie die erweiterte Funktionalität (Step3). Überprüfen Sie das Verhalten der Anwendung, wenn Sie die Menüpunkte DATEI|NEUE CD-BIBLIOTHEK und DATEI|CD BIBLIOTHEK ÖFFNEN ... auswählen

Fenstersteuerung

Das Beispielprogramm im jetzigen Zustand hat zwar die gesamte Funktionalität, doch der Ablauf vor allem der Ansichten erfüllt noch nicht alle Anforderungen. Wenn Sie über DATEI|NEUE CD-BIBLIOTHEK eine Dokumentvorlage öffnen, erhalten Sie die Einzelansicht, bei DATEI|CD BIBLIOTHEK ÖFFNEN ... die Übersichtsansicht, doch beide gleichzeitig oder gar ein Wechsel zwischen beiden ist derzeit noch nicht möglich. Dazu müssen Sie noch eine Funktion im Hauptfenster der Anwendung realisieren, die das Problem löst. Dabei muß unterschieden werden, welche Ansicht aktiv ist,

und ob das Rahmenfenster bereits vorhanden ist oder neu angelegt werden muß. Die Basisklassen der MFC stellen alle erforderlichen Methoden zur Verfügung, um an benötigte Informationen zu kommen.

Die Methode *MDIGetActiv* der Klasse *CMDIFrameWnd* liefert Ihnen den Zeiger auf das gerade aktive MDI-Kind-Fenster.

```
CMDIFrameWnd::MDIGetActive  
    CMDIChildWnd* MDIGetActive(BOOL* pbMaximized = NULL) const;
```

Da *CMDIFrameWnd* von *CFrameWnd* abgeleitet ist, kann mit dem so erhaltenen Zeiger die Methode *GetActiveDocument* verwendet werden, um das aktuelle Dokument zu ermitteln.

```
CFrameWnd::GetActiveDocument  
    virtual CDocument* GetActiveDocument();
```

Nun ist es wiederum möglich, sich über die Methoden *GetFirstViewPosition* und *GetNextView* von *CDocument* zu den zum aktiven Dokument gehörenden Ansichten entlangzuarbeiten und diese dabei mit der zu aktivierenden Ansicht zu vergleichen.

Wenn die Ansicht bereits vorhanden ist, muß nur ihr Rahmenfenster aktiviert werden. Dazu wird mit der *CWnd*-Methode *GetParentFrame* das zur Ansicht gehörenden Rahmenfenster bestimmt und mit *ActivateFrame* aktiviert.

Ist die Ansicht noch nicht vorhanden, bedeutet es, daß das Rahmenfenster der Dokumentvorlage ebenfalls noch nicht existiert. Deshalb muß mit der Methode *CreateNewFrame* der Dokumentvorlage *CDocTemplate* erst das Rahmenfenster angelegt werden

Mit *InitialUpdateFrame* nimmt dann endlich die Sache ihren Lauf. Diese Methode initialisiert nicht nur das Rahmenfenster und das Dokument, sondern ruft auch *OnInitialUpdate* der Ansichtsklasse auf.

```
CDocTemplate::InitialUpdateFrame  
    virtual void InitialUpdateFrame(CFrameWnd* pFrame,  
                                     CDocument* pDoc,  
                                     BOOL bMakeVisible = TRUE );
```

Diese auf den ersten Blick aufwendige und vielleicht auch komplizierte Art und Weise des Umgangs mit den Klassen zeigt noch einmal eindrucksvoll, wie umfangreich doch die Möglichkeiten der MFC in der Praxis sind. In unserem Beispiel wird der Zusammenhang in der Methode *CreateOrActivateFrame* noch einmal deutlich.

```

void CMainFrame::CreateOrActivateFrame(CDocTemplate * pTemplate,
                                       CRuntimeClass * pViewClass)
{
    // ermittelt entsprechendes Dokument
    CMDIChildWnd* pMDIActive = MDIGetActive();
    ASSERT(pMDIActive != NULL);
    CDocument* pDoc = pMDIActive->GetActiveDocument();
    ASSERT(pDoc != NULL);

    CView* pView;
    POSITION pos = pDoc->GetFirstViewPosition();
    // Suche, ob gewünschte View bereits existiert
    while (pos != NULL)
    {
        pView = pDoc->GetNextView(pos);
        if (pView->IsKindOf(pViewClass))
        {
            // View vorhanden -> aktivieren
            pView->GetParentFrame()->ActivateFrame();
            return;
        }
    }
    // neues MDI-Rahmenfenster anlegen
    CMDIChildWnd* pNewFrame
        =(CMDIChildWnd*)(pTemplate->CreateNewFrame(pDoc, NULL));
    if (pNewFrame == NULL) // Fehler beim Anlegen
        return;
    ASSERT(pNewFrame->IsKindOf(RUNTIME_CLASS(CMDIChildWnd)));
    pTemplate->InitialUpdateFrame(pNewFrame, pDoc);
}

```

Listing: Aktivierung oder Neuanlage des Rahmenfensters

Jetzt können Sie auch die beiden letzten Menüpunkte aktivieren. Legen Sie die Funktionen für CD ÜBERSICHT und EINE CD im Hauptfenster an. Fügen sie extern `CBibliothekApp theApp` im Kopf von *MainFrm.cpp* ein, um einen Zugriff auf die Zeiger der Dokumentvorlagen zu erhalten. Includieren Sie außerdem die Schnittstellendateien des Dokumentes und der beiden Ansichtsklassen. Rufen Sie in den neuen Funktionen nur noch *CreateOrActivateFrame* mit dem entsprechenden Parameter auf, und die Fenstersteuerung ist fertig.

```

void CMainFrame::OnCdUebersicht()
{
    CreateOrActivateFrame(theApp.pDocTemplateAlle,

```

```
        UNTIME_CLASS(CViewAlleCD));  
    }  
    void CMainFrame::OnEinzelCd()  
    {  
        CreateOrActivateFrame(theApp.pDocTemplateEinzel,  
                               UNTIME_CLASS(CViewEinzel));  
    }
```

Listing: Steuerung der Ansichten über die Methode CreateOrActivateFrame

Um die Sache abzurunden, erweitern Sie die Klasse *CViewAlleCD* dahingehend, daß bei einem Doppelklick im Listefeld die Ansichtsklasse *CViewEinzel* aktiviert wird.

Dieses Beispielprogramm veranschaulicht deutlich den Umgang mit dem Dokument-Ansicht-Paradigma der MFC. Viele Zusammenhänge zwischen Ansicht, Dokument und Dokumentvorlage wurden erläutert, so daß Sie jetzt in der Lage sein sollten, auch umfangreiche Projekte mit Visual C++ und den MFC zu realisieren.

Es bieten sich auch einige interessante Erweiterungen an: So könnten Sie weitere Formularklassen anlegen, die dann als Bibliothek für Videos dienen könnten; oder Sie benutzen geteilte Fenster, um die Ansichtsklassen gleichzeitig anzuzeigen.



Die integrierte Hilfe

14

Kapitelübersicht

14.1	Der Einsatz von WinHelp	586
14.1.1	Allgemeines	586
14.1.2	Anwendungs-Assistent und die Hilfe	586
14.1.3	Aufbau und Ablauf des Hilfe-Prozesses	588
14.1.4	Änderungen und Anpassungen der Hilfe-Dateien	589
14.1.5	Der Einsatz in der Anwendung	590
14.2	HTML-Help	593
14.2.1	Grundlagen und Voraussetzungen	593
14.2.2	Anpassungen im Projekt	595
14.2.3	Der Aufruf des Help Viewers in der Anwendung	596

14.1 Der Einsatz von WinHelp

14.1.1 Allgemeines

Windows stellt ein leistungsfähiges Hilfesystem zur Verfügung. Fast alle Windows-Programme benutzen diesen Standard. *WinHelp* ist inzwischen so leistungsfähig, daß es praktisch Handbücher und Anleitungen überflüssig macht. Um alle Anwendungsmöglichkeiten des komplexen Hilfesystems, wie Hilfefenster mit Grafik, Hyperlinks, Popup-Felder usw., zu erläutern, reicht ein der Kapitel dieses Buches nicht aus. Zu diesem Thema sind eigene Bücher geschrieben worden. Der Schwerpunkt liegt deshalb hier in der Nutzung der durch das Programmgerüst und den Anwendungs-Assistenten zur Verfügung gestellten Möglichkeiten.

WinHelp benutzt ein spezielles Dateiformat (.HLP). In diesen Dateien müssen sowohl Texte und Textattribute als auch Grafiken und Querverweise gespeichert werden. Die Verbindung aller dieser Elemente geschieht durch einen speziellen Compiler.

Voraussetzung zur Erstellung und Bearbeitung eigener Hilfetexte ist ein Textverarbeitungssystem, was das *Rich Text Format* (RTF) unterstützt. Die Steuerung aller Sonderfunktionen, wie Schriftart und -größe, Einbetten von Grafiken usw., die in der fertigen Hilfe dann zu sehen sein sollen, erfolgt hier über bestimmte Escape-Sequenzen. Microsoft Word unterstützt z.B. dieses Format. Jedoch ist die Bearbeitung mit einem Textverarbeitungssystem für größere Dateien schwierig und unübersichtlich. Für professionelle Anwendung empfiehlt es sich, ein spezielles Programm zur Hilfe-Gestaltung zu verwenden.

14.1.2 Anwendungs-Assistent und die Hilfe

Beim Anlegen des Programmgerüsts mit Hilfe des Anwendungs-Assistenten muß bereits im Schritt 4 mit dem entsprechenden Kontrollkästchen festgelegt werden, ob *Kontextabhängige Hilfe* benutzt werden soll oder nicht. Wenn die entsprechende Option aktiviert wird, legt der Anwendungs-Assistent je nach Programmausprägung folgende Funktionen und Dateien zusätzlich an bzw. erweitert sie:

Veränderung/Erweiterung	Bedeutung
MakeHelp.bat	Stapeldatei zur Erstellung einer HLP-Datei aus den zur Verfügung gestellten Komponenten.
hlp\AfxCore.rtf	RTF-Datei mit vordefinierten Hilfetexten zu Themen des Programmgerüsts wie Menü, Symbolleiste, Datei öffnen usw.
hlp\AfxPrint.rtf	RTF-Datei mit vordefinierten Hilfetexten zum Drucken, zur Seitenansicht und zum Einrichten des Druckers.

Veränderung/Erweiterung	Bedeutung
hlp\Afxolecl.rtf und hlp\Afxolesv.rtf	RTF-Dateien mit vordefinierten Hilfetexten zum Einbetten und Verknüpfen von OLE-Objekten.
<Projektname>.cnt	Textdatei mit Schlüsselwörtern für das Hilfe-Inhaltsverzeichnis in Baumstruktur.
<Projektname>.hpi	Projektdatei für den Help Workshop mit verschiedenen Sektionen wie [FILES] für alle eingefügten RTF-Dateien oder [ALIAS] zur Gleichsetzung von Kontextkennungen usw.
<Projektname>.hm	MAP-Datei mit den Zahlenwerten für die Kontextkennung bei Aufrufen aus der Anwendung, wird von <i>MakeHelp.bat</i> erzeugt
*.bmp	Bild- bzw. Grafikdateien zur Illustrierung der Hilfetexte aus den entsprechenden Hilfethemen.
Menüerweiterung	Das Standardhilfemenü <?> wird um den Menüpunkt Hilfethemen erweitert.
Erweiterung der Symbolleiste	Die Symbolleiste erhält eine zusätzlichen Schaltfläche zur Kontextkennung mit der Maus.
Message-Map	Die <i>Message-Map</i> des Hauptfensters der Anwendung wird erweitert (siehe Listing).

Tabelle 14.1: Erweiterungen durch den Anwendungs-Assistenten

```

BEGIN_MESSAGE_MAP(CMainFrame, CMDIFrameWnd)
   //{{AFX_MSG_MAP(CMainFrame)
    //...
   //}}AFX_MSG_MAP
    // Globale Hilfebefehle
    ON_COMMAND(ID_HELP_FINDER, CMDIFrameWnd::OnHelpFinder)
    ON_COMMAND(ID_HELP, CMDIFrameWnd::OnHelp)
    ON_COMMAND(ID_CONTEXT_HELP, CMDIFrameWnd::OnContextHelp)
    ON_COMMAND(ID_DEFAULT_HELP, CMDIFrameWnd::OnHelpFinder)
END_MESSAGE_MAP()

```

Listing: Nachrichtenbehandlung für die Hilfebefehle

Außerdem wird im Fenster *Dateien* des Arbeitsbereichs ein neuer Ordner *Hilfedateien* angelegt. Dort finden Sie alle verwendeten Bitmap-Dateien, die eigentlichen Hilfetexte im RTF-Format und die Datei für das Inhaltsverzeichnis.

Während der Übersetzung des Programmes wird automatisch der Hilfe-Compiler mit aktiviert. Zu erkennen ist dies durch das kurze Einblenden des Hilfe-Workshop Fensters während des Übersetzungsvorgangs in der Symbolleiste.

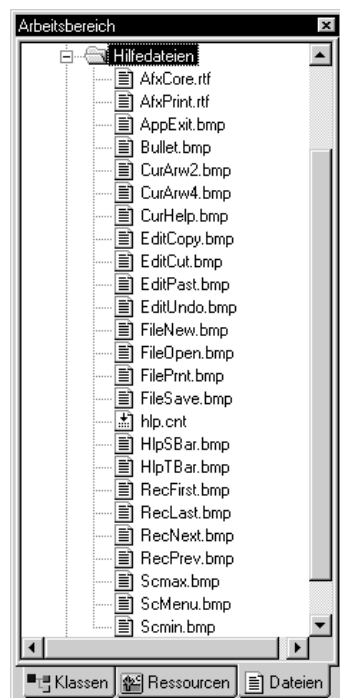


Abbildung 14.1: Der Ordner Hilfedateien im Arbeitsbereichsfenster

Beim ersten Start des Hilfe-Compilers werden die benötigten Dateien in ein Unterverzeichnis des Projektes (*hlp*) kopiert. Die Übersetzung der Hilfedatei erfolgt nicht bei jedem Übersetzungsvorgang der Anwendung. Nur wenn sich die Hilfe-Projektdatei in Datum und Uhrzeit geändert hat, wird auch der Hilfe-Compiler aktiviert.

Zur Laufzeit der Anwendung ermittelt das Programmgerüst bei Bedarf den benötigten Hilfekontext anhand der Kennung des aktiven Programmelementes. Jeder Menüpunkt, jedes Fenster und jede Symbolleiste haben eine solche Kennung.

14.1.3 Aufbau und Ablauf des Hilfe-Prozesses

Das Übersetzen der Hilfedateien eines Hilfe-Projektes ist ein kompliziertes Verfahren. Viele verschiedene Dateien in verschiedenen Stadien dienen schließlich zur Erzeugung einer HLP-Datei.

Das Hauptproblem liegt darin, daß der Hilfecompiler keine Anweisungen der Art `HID_MAINFRAME = ID_MAINFRAME + 0x20000` bearbeiten kann. Deshalb muß ein Präprozessor (*MAKEHM.EXE*) die Symbolnamen aus der Datei *RESOURCE.H* in die entsprechenden Hilfekontextwerte überführen.

Eine Verschiebung der Kontextwerte gegenüber den ursprünglichen Symbolnamen ist erforderlich, damit zwischen den verschiedenen Programmelementen unterschieden werden kann. So wird z.B. IDR_MAINFRAME u. a. für das Hauptfenster, das Menü und die Symbolleiste verwendet. Damit der Kontext diese unterscheiden kann, wird zu den Basiswerten eine entsprechende Konstante addiert. Diese Konstante hat je nach Art des Programmelementes einen anderen Wert.

Programmelement	Präfix in der Anwendung	Präfix der Hilfe	Basiswert (hex)
Menüpunkt	ID_, IDM_	HID_, HIDM_	10 000
Ansichten, Dialoge, Rahmenfenster	IDR_, IDD	HIDR_, HIDD_	20 000
Fehler- und Statusfenster	IDP_	HIDP_	30 000
Andere Fensterelemente	andere	H_	40 000
Symbolleiste	IDW_	HIDW_	50 000

Tabelle 14.2: Konstanten für Hilfekontext

Ausgangsbasis für das Erzeugen einer HLP-Datei sind die Projektdatei (HPJ) und die Inhaltsverzeichnisdatei (CNT). Die in Sektion [FILES] aufgeführten RTF-Dateien werden eingebunden und mit den in Sektion [MAP] definierten Kontextkonstanten verbunden. Die Konstanten können dabei direkt unter [MAP] eingetragen oder über den *#include*-Befehl dazugebunden werden. MAKEHM.EXE erzeugt die Datei *<Projektname>.HM* aus RESOURCE.H des Projektes. Ergebnis dieses ganzen Prozesses ist eine HLP-Datei, die von *WinHelp* verarbeitet werden kann.

Der Aufruf der Windows-Hilfe aus der Anwendung erfolgt über die Methode *WinHelp* der Klasse *CWinApp*. Diese Methode hat zwei Parameter:

```
CWinApp::WinHelp
    virtual void WinHelp(DWORD dwData, UINT nCmd = HELP_CONTEXT);
```

Der erste Parameter *dwData* ist die Kontextkennung. Über den zweiten Parameter kann die Art und Weise des Hilfeaufrufes klassifiziert werden.

14.1.4 Änderungen und Anpassungen der Hilfe-Dateien

In den vom Anwendungs-Assistenten erstellten RTF-Dateien sind an den entsprechenden Stellen Hinweise zur Erweiterung bzw. Ergänzung enthalten. Dazu gehört u.a. das Ersetzen des Textes *<YourApp>* durch den eigentlichen Namen der Anwendung. Außerdem müssen die Hinweise zum Umgang mit den Daten in den Dokumenten an den entsprechenden Stellen erläutert werden.

Für alle Erweiterungen und Änderungen im Programmgerüst wie Menü, Symbolleiste zusätzliche Dokumente, Ansichten und Dialoge, muß ein entsprechender Hilfetext in den RTF-Dateien mit einer entsprechenden Textverarbeitung angepaßt oder erstellt werden. Gegebenenfalls sollte man auch eigene neue RTF-Dateien anlegen. Diese müssen dann nur in die Projektdatei unter *FILES* eingetragen werden.

Zur Bearbeitung des Hilfeprojektes steht der *Microsoft Hilfe-Workshop* zur Verfügung. Dieses Programm heißt *HCRTF.EXE* und ist unter *Dienstprogramme* zu finden. Mit diesem Werkzeug kann das Hilfeprojekt sowohl menügesteuert verwaltet als auch das Projekt erstellt und getestet werden.



Abbildung 14.2: Der Hilfe-Workshop

Auf den Umgang und die Bedienung des Hilfe-Workshops wird hier nicht näher eingegangen. Schwerpunkt ist die Anwendung der Windows-Hilfe in eigenen Programmen.

14.1.5 Der Einsatz in der Anwendung

Hinter dem Menüpunkt *HILFETHEMEN* unter dem Fragezeichen (?) im vom Anwendungs-Assistenten angelegten Menü verbirgt sich folgender Aufruf der Windows-Hilfe:

```
WinHelp(0L, HELP_FINDER);
```

Dieser wird allerdings nicht direkt ausgeführt, sondern über die Methode *WinHelp* der Klasse *CWinApp* an die *WinHelp* Methode des Hauptfensters weitergegeben..

```
CWinApp::WinHelp
virtual void WinHelp(DWORD dwData, UINT nCmd = HELP_CONTEXT);
```

Erst hier erfolgt der eigentliche Aufruf. Der Parameter *HELP_FINDER* bewirkt den Aufrufen des *Help Topics* Dialoges.

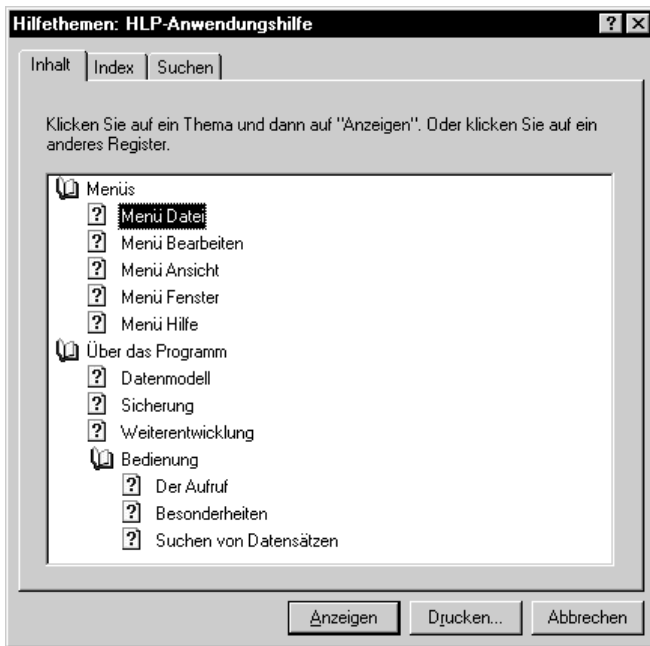


Abbildung 14.3: Der Hilfethemen Dialog

Es ist jedoch möglich, die Hilfe mit anderen Parametern zu starten. Für den Aufruf der Windows-Hilfe in der alten Form ist folgender Parameter vorgesehen:

```
WinHelp(0L, HELP_INDEX);
```

Er startet sofort die Windows-Hilfe mit dem Hauptfenster. Dieser Aufruf wird vom Programmgerüst standardmäßig bei drücken der **[F1]** Taste ausgelöst. Die dabei angezeigte Startseite ist in der vom Anwendungs-Assistenten angelegten Datei *AfxCore.rtf* enthalten.



Abbildung 14.4: Die Anwendungshilfe mit F1

Soll das Hilfemenü um den Menüpunkt *Hilfe verwenden* erweitert werden, dann sieht der Aufruf so aus:

```
WinHelp(0L, HELP_HELPONHELP);
```

Mit dem Parameter `HELP_HELPONHELP` wird der *Help Topics* Dialog mit der Hilfe zur Hilfe aufgerufen. Der Inhalt ist vom bei Windows NT etwas anders als bei W95 oder W98. Auf jeden Fall erhalten Sie so eine Beschreibung zum Bedienen von WinHelp.

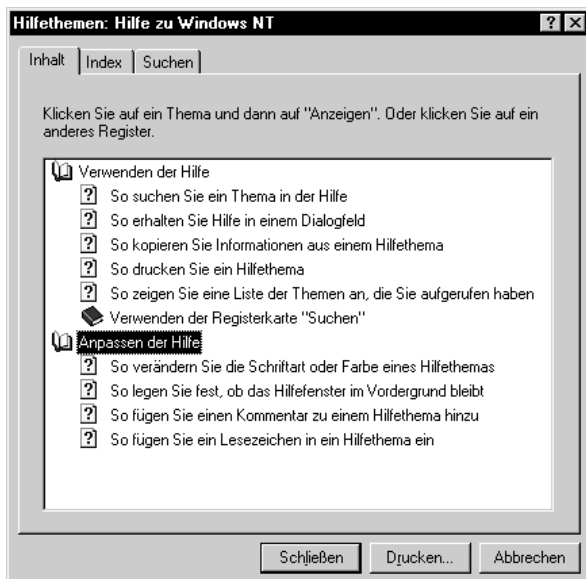


Abbildung 14.5: Der Hilfethemen Dialog mit der Hilfe zur Hilfe

Neben der kontextgesteuerten Hilfe über Konstanten ist auch ein textorientierter Einsatz möglich. Sollen z.B. Schlüsselwörter direkt gesucht werden, kann folgender Aufruf verwendet werden:

```
CString strSuchen("Beenden");
WinHelp((DWORD) (LPCSTR) strSuchen, HELP_KEY);
```



Bei dem Aufruf von *WinHelp* mit dem Parameter `HELP_KEY` wird ein zusätzlicher Parameter übergeben, der in der Hilfe-Datei als Schlüssel vorhanden sein muß. Wenn der Ausdruck gefunden wurde, zeigt die Windows-Hilfe das entsprechende Thema sofort an. Ist der Schlüssel nicht bekannt, wird der *Help Topics* Dialog angezeigt mit aktiviertem Index-Reiter.

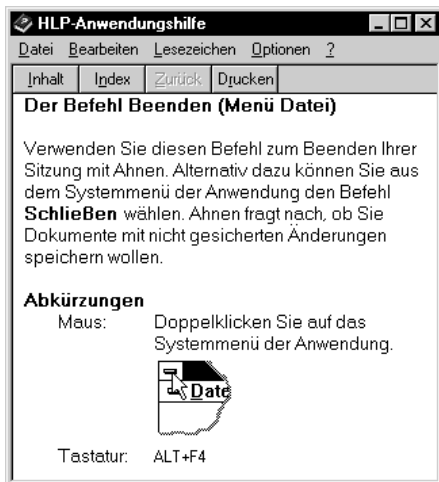


Abbildung 14.6: Direkter Aufruf eines Hilfethemas zur Hilfe

Der erste Parameter von *WinHelp* wird für viele Zwecke verwendet. Deshalb ist diese doppelte Konvertierung des notwendig. Die Verwendung des ersten Parameters wird über den zweiten Parameter gesteuert.

14.2 HTML-Help

14.2.1 Grundlagen und Voraussetzungen

Die zunehmende Verbreitung des HTML-Formats für Dokumente rund um den PC legt es nahe, auch die Hilfe-Dateien und Dokumentationen in diesem Format zur Verfügung zu stellen. Dieses Format unterstützt neben den Möglichkeiten von *WinHelp* auch weitere multimediale Effekte wie Animation und Klang. Außerdem gibt es eine Reihe komfortabler Werkzeuge zum Erstellen solcher Dokumente.

Mit der Version 6 von Visual C++ wird das Hilfesystem *HTML Help* mit ausgeliefert. Dazu gehört auch der *HTML Help Workshop* zum Erstellen der Online-Hilfe für Ihre Anwendungen. Diese Werkzeuge werden allerdings nicht automatisch mit installiert. Sie müssen die Datei *Htmlhelp.exe* im Unterverzeichnis *\HtmlHelp* Ihrer Visual C++ CD ausführen, um diese Komponenten zu installieren.

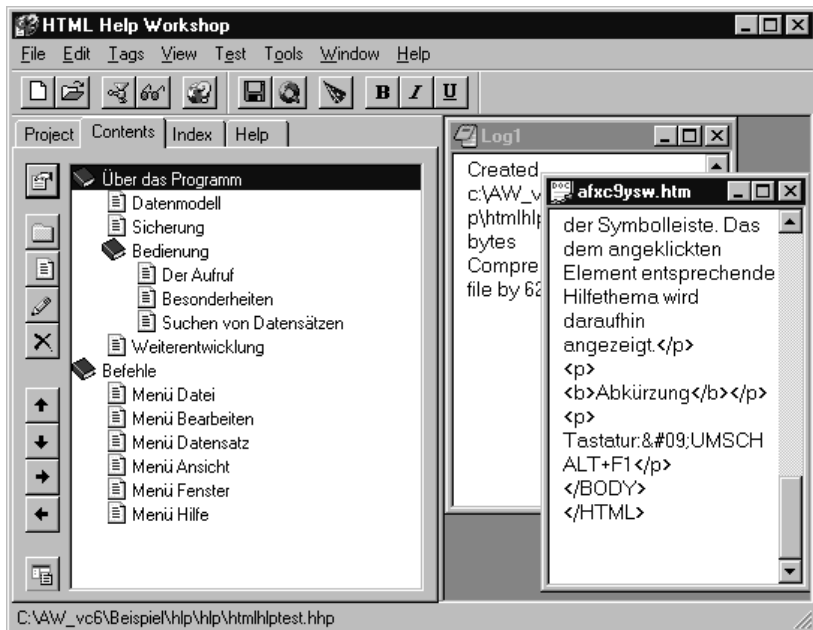


Abbildung 14.7: Der HTML Help Workshop

Damit aus der Anwendung heraus auf *HTML Help* zugegriffen werden kann, ist der Internet-Explorer 3.x oder höher auf dem System erforderlich. *HTML Help* benutzt Komponenten vom Internet-Explorer für die Anzeige der Dokumente. Da Microsoft die Windows-Betriebssysteme mit dem kostenlosen Internet-Explorer ausliefert, kann man von einer großen Anzahl von Installationen ausgehen.

WinHelp und *HTML Help* können gleichzeitig in der Anwendung verwendet werden. Jedoch ist eine Kombination beider in der Regel nicht sinnvoll bzw. wünschenswert, da zwischen beiden Hilfesystemen keine Verbindung besteht.

14.2.2 Anpassungen im Projekt

Die Verwendung von *HTML Help* wird nicht durch den Anwendungs-Assistenten unterstützt. Im Schritt 4 des Anwendungs-Assistenten können Sie nur die Unterstützung von *WinHelp* aktivieren. Für den Einsatz von *HTML Help* ist das nicht erforderlich. Vielmehr müssen Sie selber Hand an das Projekt legen. Im Einzelnen sind folgende Anpassungen erforderlich:

1. Ergänzen Sie in `Stdafx.h` das Include `#include <htmlhelp.h>`.
2. Damit der Präprozessor diese neue Datei auch findet, tragen Sie unter PROJEKT|EINSTELLUNGEN in der Registerkarte C/C++ in der Kategorie *Präprozessor* das *zusätzliche Include-Verzeichnis* der HTML Help Workshops (z.B. C:\Programme\HTML Help Workshop\include) ein. Führen Sie die Einstellung für *Alle Kategorien* aus.
3. Tragen Sie unter PROJEKT|EINSTELLUNGEN in der Registerkarte LINKER in der Kategorie *Allgemein* unter *Objekt-/Bibliothek-Module* `Htmlhelp.lib` ein. Unter der Kategorie *Eingabe* tragen Sie bei *Zusätzlicher Bibliothekspfad* den HTML Help Workshop Bibliothekspfad (z.B. C:\Programme\HTML Help Workshop\lib) ein. Führen Sie auch diese Einstellung für *Alle Kategorien* aus.

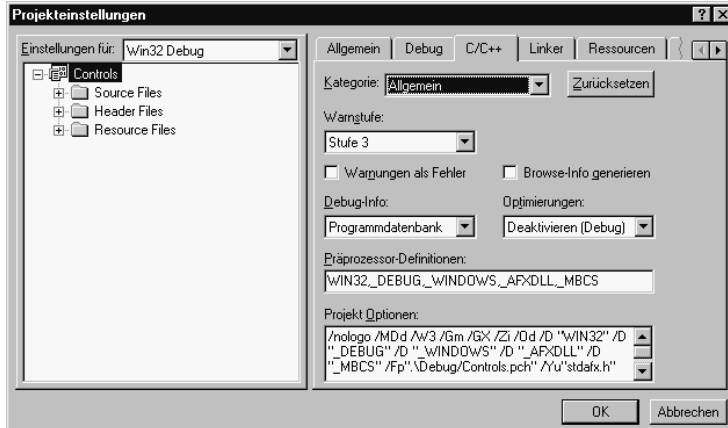


Abbildung 14.8: Änderung der Projekteinstellung für die Verwendung von HTML-Help

Im Gegensatz zur Unterstützung von *WinHelp* durch den Anwendungs-Assistenten, werden für *HTML Help* keine Dateien angelegt, die die Hilfetexte für die im Programmgerüst verwendeten Elemente enthalten. Der *HTML Hilfe Workshop* hat jedoch eine Funktion, ein *WinHelp*-Projekt in ein *HTML Help*-Projekt umzuwandeln. Dabei werden alle Dateien aus dem RTF-Format in HTML umgewandelt.

14.2.3 Der Aufruf des Help Viewers in der Anwendung

Der Aufruf des *Help Viewers* erfolgt immer über die Funktion *HtmlHelp* der eingebundenen *Htmlhelp.lib*.

```
HWND HtmlHelp(HWND hwndCaller, LPCSTR pszFile,
              UINT uCommand, DWORD dwData);
```

Dabei haben die Parameter folgende Bedeutung: Der erste *hwndCaller* ist ein Fenster-Handle des aufrufenden Fensters und dient der weiteren Kommunikation zwischen *Help Viewer* und Anwendung. Der zweite Parameter ist die Hilfedatei, die verwendet werden soll. Dabei kann es sich um eine HTML-Datei, eine URL oder eine kompilierte HTML-Datei handeln. Auch ein Fenster innerhalb der kompilierten Hilfe-Datei kann damit angesprochen werden. *uCommand* ist, wie der Name schon sagt das Kommando an den *Help Viewer*. Mit diesem Parameter wird nicht nur das Anzeigen der Hilfethemen ausgelöst, sondern der *Help Viewer* kann darüber regelrecht gesteuert werden. Der letzte Parameter steht in Verbindung mit dem Kommando und enthält zusätzliche Informationen.

Es gibt verschiedene Möglichkeiten den *Help Viewer* von der Anwendung aus aufzurufen. Der Parameter *HH_DISPLAY_TOPIC* von *HtmlHelp* hat etwa die gleiche Funktionalität wie der Parameter *HELP_FINDER* von *WinHelp*.



```
HtmlHelp(NULL, "Ahnen.chm::/topic.htm", HH_DISPLAY_TOPIC, 0);
```

In diesem Beispiel wird die Topic-Liste aus der kompilierten HTML-Datei *Ahnen.chm* im *Help Viewer* angezeigt. Die Bezeichnung der Seite *topic.htm* kann entfallen, wenn im Hilfeprojekt eine Standardseite angegeben wurde.

Es ist aber auch möglich kontextabhängige Hilfe mit dem *Help Viewer* zu realisieren. Damit können Sie auf das Drücken der rechte Maustaste mit der Anzeige eines Hilfetextes reagieren. *HtmlHelp* benutzt dafür den Parameter *HH_TP_HELP_CONTEXTMENU*. Voraussetzung ist, daß Sie ein zweidimensionales modulglobales Array anlegen, welches alle unterstützten ID's der Steuerelemente enthält und die dazugehörigen Topic-ID's.

```
static DWORD aKontext[] = {
    ID_BUTTON_LOESCHEN, 1,
    IDOK, 2,
    ID_SUCHEN, 3,
    ID_STATIC, -1,
    0,0};
```

Listing: Feld mit der Zuordnung Steuerungs-ID zu Topic-ID

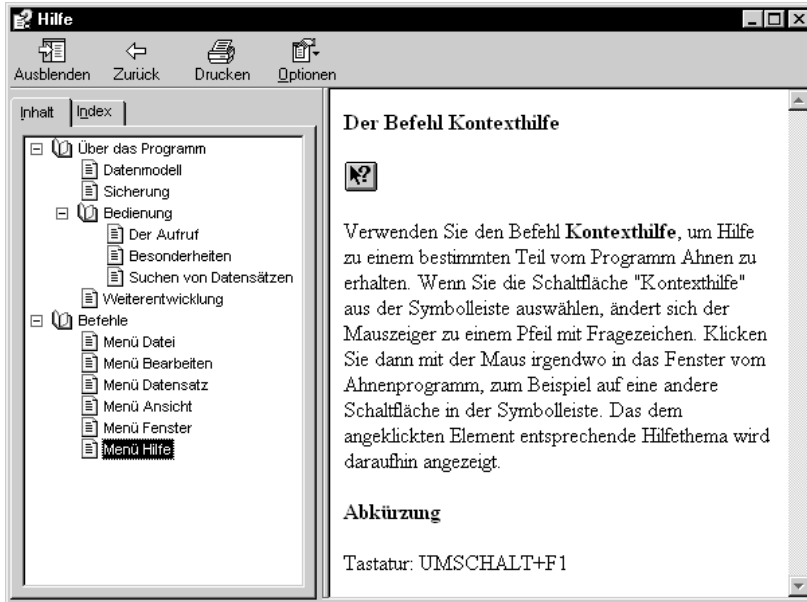


Abbildung 14.9: Standardfenster des HTML-Viewers

Den Abschluß des Feldes muß Paar 0,0 bilden. Für Steuerungen ohne Hilfe geben Sie als Topic-ID -1 an. Jetzt müssen Sie nur noch in einer Text-Datei die Zuordnung Topic-ID zu Anzeigetext vornehmen und diese Datei in Ihr HTML-Hilfeprojekt aufnehmen. Diese Datei hat etwa folgenden Aufbau:

```
.topic 1
diese Schaltfläche löscht den aktuellen Datensatz
.topic 2
Sichern der Daten
.topic 3
Suche nach einem Datensatz
```



Um in Ihrer Anwendung die kontextabhängige Hilfe aufzurufen, müssen Sie auf die Nachricht WM_CONTEXTMENU reagieren. Dazu erstellen Sie mit dem Anwendungs-Assistenten eine Behandlungsmethode für das betreffende Fenster. Diese Methode wird dann vom Programmgerüst indirekt durch die Nachricht WM_RBUTTONDOWN ausgelöst, wobei bereits die betroffene Steuerung über der die Maustaste gedrückt wurde ermittelt ist. Sie müssen dann nur noch *HtmlHelp* mit den entsprechenden Parametern aufrufen.

```
void CAhnenView::OnContextMenu(CWnd* pWnd, CPoint point)
{
    HtmlHelp(pWnd->GetSafeHwnd(),
              "Ahnen.chm:/dlghlp.txt",
              HH_TP_HELP_CONTEXTMENU,
              (DWORD)(LPVOID)aKontext);
}
```

Listing: Aufruf der kontextabhängigen Hilfe

Wenn Sie diese Funktionalität implementiert haben, ist sehr einfach den **[F1]**-Zugriff auf die kontextabhängige Hilfe zu realisieren. Legen Sie eine Behandlungsmethode für die Nachricht WM_HELPINFO in Ihrem Fenster an. Der Parameter HH_TP_HELP_WM_HELP von HtmlHelp unterstützt diese Art von Hilfeaufruf.

```
BOOL CAhnenView::OnHelpInfo(HELPINFO* pHelpInfo)
{
    if (pHelpInfo->iContextType == HELPINFO_WINDOW)
    {
        return HtmlHelp((HWND)pHelpInfo->hItemHandle,
                        "Ahnen.chm:/dlghlp.txt",
                        HH_TP_HELP_WM_HELP,
                        (DWORD)(LPVOID)aKontext) != NULL;
    }
    return TRUE;
}
```

Eine weitere interessante Möglichkeit zum Anzeigen von Hilfetexten ist die Unterstützung von Fenstern. Sie könne im Hilfe Workshop Fenster mit verschiedenen Eigenschaften anlegen und diese dann direkt aus Ihrem Programm aufrufen. Dabei können Sie u. a. Form und Größe aber auch Art und Umfang der Symbolleiste festlegen (siehe Abbildung 14.10).

Sie können dann verschiedene Hilfetexte in einem Fenster oder nur die default-Datei anzeigen lassen. Der Aufruf sieht dann z. B. so aus:

```
HtmlHelp(NULL, "Ahnen.chm>WndEnde", HH_DISPLAY_TOPIC , 0);
```

Das Größer als Zeichen (>) weist auf die Verwendung des entsprechenden Fensters hin.

Es gibt auch einen Steuerbefehl zum Schließen der Hilfe-Fensters im Help Viewer. Mit dem Kommando HH_CLOSE_ALL werden die Fenster geschlossen.

```
HtmlHelp(NULL, NULL, HH_CLOSE_ALL, 0);
```



Abbildung 14.10: HTML-Hilfe in einem eigenen Fenster

Es konnten in diesem Abschnitt nur die wichtigsten Funktionen der neuen HTML Hilfe vorgestellt werden. Doch diese Möglichkeiten reichen aus, um eine leistungsfähige und attraktive Hilfe mit diesem Standard zu realisieren. Im Lieferumfang des HTML Help Workshops ist auch ein Image-Editor, der eine Reihe von Funktionen zur Bildbearbeitung zur Verfügung stellt.



**Datenbank-
unterstützung**

TEIL V



MFC

Datenbankschnittstellen

15

Kapitelübersicht

15.1 Grundlagen	604
15.1.1 Elementares	604
15.1.2 Konzept	604
15.1.3 Voraussetzungen	606
15.2 Unterschied zur Serialisation	606
15.3 Die Klasse CDatabase	608
15.4 Die Klasse CRecordset	610
15.5 Die Klasse CRecordView	613
15.6 ODBC-Konfiguration	614
15.7 Assistentenunterstützung	616
15.7.1 Anwendungs-Assistent	616
15.7.2 Klassenassistent	618
15.8 Datenbank-Ausnahmebehandlung der MFC	619
15.9 ODBC-Datenbankanwendung - ein Beispiel	620
15.9.1 Vorteile und Besonderheiten	620
15.9.2 Aufgabe und Implementation	621
15.9.3 Projektanlage mit dem Anwendungs-Assistenten	622
15.9.4 Erweitern und Anpassen der Ressourcen	628
15.9.5 Zusätzlicher Programmcode	630
15.9.6 Fazit und Anregungen	641
15.10 Erweiterte SQL-Funktionen	641

15.1 Grundlagen

15.1.1 Elementares

Unter den kommerziell eingesetzten Programmen nehmen Anwendungen mit Zugriff auf Datenbanken eine herausragende Stelle ein. Unterschiedliche Konzepte und Hersteller führten im PC-Bereich zu verschiedenen, nicht unmittelbar miteinander verträglichen Lösungen. Üblicherweise werden in sich geschlossene Datenbanksysteme verkauft, die Komponenten zur Datenverwaltung (Speicherung, Indizierung etc.) und Möglichkeiten zu deren Bearbeitung und Auswertung (Abfragesprache oder vollwertige Programmiersprache) beinhalten. Das älteste und auch bekannteste dieser PC-Datenbanksysteme ist dBASE mit vielen Klones.

Eine andere, nicht nur auf PCs beschränkte Variante von Datenbanksystemen sind SQL-Server, die insbesondere unter Netzen oder bei Client/Server-Anwendungen durch entsprechende Schnittstellen anderen, in beliebigen Programmiersprachen geschriebenen Programmen den Zugriff auf eine Datenbank ermöglichen. Diese Systeme sind nach außen hin offen, der Zugriff auf die Daten ist aber nur über den SQL-Server möglich, der nur Daten seines eigenen Formates bearbeiten kann.

15.1.2 Konzept

Zwei verschiedene Konzepte für eine Datenbankschnittstelle wurden in der MFC realisiert. Dabei sind die Datenbank-API-Funktionen in eigene Klassen gekapselt und vereinfachen so den Zugriff auf die Daten. Beide Möglichkeiten werden von den Anwendungs-Assistenten unterstützt.

Das ODBC-Konzept (Open Database Connectivity) von Microsoft stellt die erste Datenbankschnittstelle dar. Sie gestattet über den ODBC-Server und diverse Treiber den Zugriff auf verschiedene Datenbanken, z.B. MS-Access, dBASE, MS SQL-Server, Oracle-Server oder Excel-Tabellen. Der ODBC-Treiber kann wegen seiner definierten Schnittstelle von verschiedenen Programmiersprachen aus angesprochen werden. Seine SQL-basierte Schnittstelle ist weitestgehend unabhängig vom konkreten Format der Datenbank. Das ODBC-Konzept gestattet also verschiedenen Programmen den Zugriff auf Daten der unterschiedlichsten Datenbanksysteme über eine einheitliche Schnittstelle.

Normalerweise wird der ODBC-Server über API-Aufrufe angesprochen. In den MFC existieren einige Klassen, die sowohl die Benutzung des ODBC-Servers vereinfachen als auch ein an der Dokumentansicht orientiertes Anwendungsgerüst für Datenbank Anwendungen zur Verfügung stellen.

Die ODBC-basierten MFC-Datenbankklassen bieten Ihnen Zugriff auf jede Datenbank, für die ein ODBC-Treiber zur Verfügung steht. Da die Klassen ODBC verwenden, kann Ihre Anwendung auf Daten in vielen verschiede-

nen Datenformaten zugreifen. Sie müssen keinen speziellen Code für den Umgang mit bestimmten Datenbank-Management-Systemen (DBMS) schreiben.

Eine zweite Datenbankschnittstelle von Microsoft stellt das DAO-Konzept (Data Access Objects) dar. DAO verfügt über eine hierarchische Gruppe von Objekten, die das Microsoft Jet-Datenbankmodul für den Zugriff auf Daten und die Datenbankstruktur verwenden. DAO arbeitet am besten mit Microsoft Access(*.mdb)-Datenbanken zusammen, kann über ODBC aber auch auf andere Datenbanktypen zugreifen. DAO ist eine auf OLE basierende Anwendungsprogrammierschnittstelle, die auf das Microsoft Jet-Datenbankmodul hin optimiert wurde. Dabei bietet die DAO eine umfangreichere Datenbankschnittstelle als ODBC.

DAO ist ebenfalls über API-Aufrufe möglich, doch auch hier existieren Klassen in den MFC, die die Funktionalität kapseln. Die MFC-DAO-Klassen haben zu DAO eine ähnliche Beziehung wie die allgemeinen MFC-Klassen zur Windows-Programmierung mit der Windows-API: Die MFC kapselt den DAO-Funktionsumfang in mehrere Klassen ein, die eng an die DAO-Recordsets angelehnt sind. Die Kapselung ist zwar umfassend, aber nicht vollständig.

MFC und DAO stellen für einige Objekte, die in Microsoft Access verwendet werden, keine Abstraktionen zur Verfügung. Wenn Sie eine Microsoft Access-Datenbank erstellen und diese von einer MFC-Anwendung aus ändern wollen, können Sie auf einige Objekte, wie Formular, Modul, Bericht, Bildschirm usw., nicht zugreifen. Die MFC stellt auch keine Klassen oder Schnittstellen zu den DAO-Recordsets **Gruppe** und **Anwender** zur Verfügung. Die MFC kapselt auch keine DAO-Property-Objekte ein, die MFC-DAO-Klassen geben Ihnen allerdings Zugriff auf die Eigenschaften aller offgelegten Objekte.

DAO wird unter Win32s nicht unterstützt. Außerdem ist DAO 3.x nicht tauglich bei Threads. Sie dürfen DAO nur im primären Programmfaden einer Anwendung verwenden.



Mit etwas Abstand kann man feststellen, daß DAO, von den MFC aus gesehen, ein optimiertes ODBC ist, welches aber seine Stärken nur bei Microsoft Access ausspielen kann.

Ob Sie nun die MFC-DAO-Klassen oder die MFC-ODBC-Klassen einsetzen, hängt vom Einsatzzweck und den Anforderungen ab. Die Klassen und deren Methoden sind sehr ähnlich, so daß ein Wechsel zwischen den beiden Verfahren nachträglich leicht möglich ist. Im weiteren werden in diesem Buch überwiegend die MFC-ODBC-Klassen benutzt und beschrieben. Die Funktionalität für DAO ist dann entsprechend.

15.1.3 Voraussetzungen

Bereits bei der Installation von Visual C++ müssen Sie wählen, ob Sie die Datenbankkomponenten installieren wollen oder nicht. Dort stehen die zu installierenden ODBC-Treiber zur Auswahl. Sie müssen auf jeden Fall mindestens einen ODBC-Treiber auswählen, egal, ob Sie DAO oder ODBC verwenden wollen. Dieser Treiber wird dann zusammen mit dem ODBC-Treiber-Manager und dem ODBC-Administrationsprogramm auf Ihrer Festplatte installiert.

Bei der Standardinstallation kopiert das Setup-Programm folgende ODBC-Treiber:

- ▼ Microsoft FoxPro
- ▼ Microsoft Access
- ▼ dBASE
- ▼ Microsoft SQL-Server

Bei der benutzerdefinierten Installation können Sie zusätzlich folgende ODBC-Treiber installieren:

- ▼ Textdateien
- ▼ Paradox
- ▼ Microsoft Excel
- ▼ Oracle
- ▼ FoxPro / Visual FoxPro

Visual C++ enthält darüber hinaus weitere ODBC-Komponenten wie die notwendigen Schnittstellendateien, Bibliotheken, DLLs und Programme. Zu den installierten Dienstprogrammen gehören zum Beispiel das ODBC-Datenquellen-Administratorprogramm, das Sie in der Systemsteuerung wiederfinden und welches zur Konfiguration der ODBC-Datenquellen benötigt wird. Auch der ODBC-Treiber-Manager wird installiert. ODBC-Treiber für die bekanntesten DBMS sind enthalten; eine Liste dieser Treiber finden Sie im Artikel »Installierte ODBC-Treiber«.

15.2 Unterschied zur Serialisation

Sie werden sich nun sicher fragen: Wann kommen Dokumentobjekte und wann das Serialisieren für die dateibasierte Ein- und Ausgabe zum Einsatz? Dazu eine kurze Frageliste, anhand der Sie entscheiden können, welches Verfahren am geeignetsten ist:

Frage	Serialisation	Dokumentobjekte
Sind die primären Daten in einer Datenträgerdatei gespeichert?	Ja	
Soll die Anwendung die gesamte Datei in den Hauptspeicher einlesen (kleine Datenmenge)?	Ja	
Sie benötigen den Befehl <i>Datei öffnen</i> .	Ja	(Ja)
Benötigen Sie eine transaktionsbasierende Aktualisierung?		Ja
Sind die Daten in einer ODBC-Datenquelle gespeichert?		Ja
Gibt es Schlüsselfelder in Ihren Daten, nach denen eine Sortierung erfolgen soll (Indizes)?		Ja
Sind Zugriffe auf die Daten von fremden Programmen möglich?		Ja
Sollen mehrere Benutzer gleichzeitig mit den Daten arbeiten können?		Ja

Tabelle 15.1: Entscheidungshilfen

In jedem Fall sollte ein *CDocument*-Objekt die Grundlage Ihrer Anwendung mit Datenverwaltung sein. Der augenfälligste Unterschied besteht im Menüpunkt DATEI. Das Programmgerüst unterstützt die Befehle zum ÖFFNEN, SCHLIESSEN, SPEICHERN und SPEICHERN UNTER nur für die Dokumentserialisierung. Die Serialisierung liest und schreibt Daten (einschließlich Objekte, die von der Klasse *CObject* abgeleitet wurden) von einer und in eine Datenträgerdatei.

In einer MFC-Datenbankanwendung müssen Sie sich um die Interpretation und Realisierung der Menübefehle ÖFFNEN, SCHLIESSEN, SPEICHERN und SPEICHERN UNTER des Menüpunktes DATEI selbst kümmern, oder Sie eliminieren die Befehle vollständig, wenn es keine sinnvolle Verwendung gibt. Es ist aber auch möglich, beide Techniken in einem Programm zu benutzen. Mit der Serialisierung speichern Sie Benutzerprofile und Programmeinstellungen, und bei MFC-Datenbankklassen bearbeiten Sie die Daten in einer Datenbank.

Um einen Befehl im Menüpunkt DATEI selbst zu behandeln, müssen Sie eine oder mehrere Nachrichtenbehandlungsroutinen überschreiben, zumeist in der von *CWinApp* abgeleiteten Klasse. Wenn Sie z. B. die Methode *OnFileOpen* (ID_FILE_OPEN) überschreiben wollen, damit sie in Ihrer MFC-Datenbankanwendung die Bedeutung »Datenbank öffnen« erhält, dann rufen Sie nicht *OnFileOpen* der Basisklasse *CWinApp* auf. Verwenden Sie statt dessen die Behandlungsroutine, um ein Dialogfeld anzuzeigen, in dem die Datenquellen aufgeführt werden. Sie können ein derartiges Dialogfeld anzeigen, indem Sie die Methoden *OpenEx* oder *Open* der Klasse *CDatabase* mit dem Parameter NULL aufrufen. Dadurch wird ein ODBC-Dialogfeld geöffnet, in dem alle auf dem PC zur Verfügung stehenden Datenquellen angezeigt werden.

Die Serialisation ist leicht zu verwenden und für viele Ihrer Anforderungen ausreichend, kann aber in Anwendungen mit häufigem Datenzugriff ungeeignet sein. Solche Anwendungen aktualisieren die Daten oft auf der Grundlage von Transaktionen. Sie aktualisieren nur die betroffenen Datensätze, anstatt die ganze Datei auf einmal zu lesen oder zu schreiben.

Es existieren noch weitere Verfahren, um unter Windows auf Daten zugreifen zu können. Über OLE oder DDE besteht auch die Möglichkeit, auf andere Datenbank-Management-Systeme zuzugreifen. Doch auf solche Methoden kann in diesem Buch nicht näher eingegangen werden.

15.3 Die Klasse CDatabase

Bevor Sie auf eine Datenbank zugreifen können, muß die Anwendung eine Verbindung zu der jeweiligen Datenquelle herstellen. Innerhalb der MFC erfolgt dies durch die Klasse *CDatabase*. Diese Klasse übernimmt über die API-Schnittstelle die gesamte Kommunikation mit dem ODBC-Treiber und damit der Datenbank. Der Zugriff erfolgt über Methoden dieser Klasse, die damit die eigentlichen ODBC-API-Funktionen kapseln.

Eine der wichtigsten Methoden ist das Öffnen der Datenbank, denn hier werden ihre Eigenschaften und ihr Verhalten bestimmt. Die Klasse *CDatabase* besitzt zwei Methoden zum Öffnen: *Open* und *OpenEx*.

```
class Cdatabase:
    virtual BOOL Open(LPCTSTR lpszDSN,
                     BOOL bExclusive = FALSE,
                     BOOL bReadOnly = FALSE,
                     LPCTSTR lpszConnect = "ODBC;",
                     BOOL bUseCursorLib = TRUE);
    throw(CDBException, CMemoryException);
    virtual BOOL OpenEx(LPCTSTR lpszConnectionString,
                       DWORD dwOptions= 0);
    throw(CDBException, CMemoryException);
```

Dabei haben die Parameter folgende Bedeutung:

1. *lpszDSN* – ODBC-Datenbankquelle im DSN-Format
2. *bExclusive* – z.Z. noch nicht unterstützter Parameter
3. *bReadOnly* – Schreibschutz für die Datenbank
4. *lpszConnect* – Anmeldezeichenkette mit User und Password
5. *bUseCursorLib* – legt fest, ob die ODBC Cursor Library DLL benutzt wird

6. `lpszConnectionString` – ausführliche Anmeldezeichenfolge (siehe unten)
7. `dwOptions` – mit ODBC verknüpfte Eigenschaften beim Öffnen
 - ▼ `openReadOnly` – Schreibschutz für die Datenbank
 - ▼ `openExclusive` – z.Z. nicht unterstützt
 - ▼ `useCursorLib` – ODBC Cursor Library DLL benutzen
 - ▼ `noOdbcDialog` – unterdrückt ODBC-Auswahldialog
 - ▼ `forceOdbcDialog` – ruft ODBC-Auswahldialog auf

Der Parameter `lpszConnectionString` der Methode *OpenEx* ist interessant, da man mit seiner Hilfe das oft lästige Registrieren der ODBC-Datenbank umgehen kann. In diesem Parameter können Sie alle Informationen unterbringen, die zum Öffnen einer ODBC-Datenquelle erforderlich sind.

Die Methoden zum Öffnen der Datenbank unterstützen eine Ausnahmebehandlung. Wenn es Probleme beim Öffnen gab, ist nicht nur der Rückgabewert der Methode `FALSE`, sondern es wird die Funktion zur Behandlung der Ausnahme angesprungen (siehe Kapitel 15.8).

Das Gegenteil zum Öffnen der Datenbank ist das Schließen. Dafür hat die Klasse *CDatabase* die Methode *Close*.

```
CDatabase::Close  
virtual void Close( );
```

Diese Methode hat keine Parameter und auch keinen Rückgabewert. Nach dem Schließen der Datenbank mit dieser Methode kann eine neue Datenbank mit anderen Parametern mit dieser Klasse geöffnet werden.

Eine weitere interessante Methode ist *ExecuteSQL*. Mit ihrer Hilfe können Sie SQL-Befehle direkt an eine geöffnete Datenbank absetzen.

```
CDatabase::ExecuteSQL  
void ExecuteSQL(LPCSTR lpszSQL);  
throw(CDBException);
```

Übergeben Sie dieser Methode den Befehl einfach als Zeichenkette. Auch hier ist eine Ausnahmebehandlung vorgesehen.

Die Klasse enthält nicht nur Methoden zum Öffnen, Anmelden und Schließen der Datenbank, sondern auch zur Transaktionssteuerung, zum Aufruf gespeicherter Prozeduren der Datenbank usw.

Für das Steuern von Transaktionen stehen drei Methoden der Klasse *CDatabase* zur Verfügung. Den Start bildet die Methode *BeginTrans*.

```
CDatabase::BeginTrans  
    BOOL BeginTrans( );
```

Wurde diese Methode erfolgreich ausgeführt, können Sie Datensätze ändern, hinzufügen oder löschen, z. B. über eine Datensatzgruppe (siehe unten). Wurden alle Daten erfolgreich geändert, beenden Sie die Transaktion mit *CommitTrans*.

```
CDatabase::CommitTrans  
    BOOL CommitTrans();
```

Sollen dagegen die Änderungen zurückgenommen werden, rufen Sie *Rollback* auf, und alle Änderungen seit dem Aufruf von *BeginTrans* werden rückgängig gemacht.

```
CDatabase::Rollback  
    BOOL Rollback();
```

Auch das war nur eine Auswahl der wichtigsten Methoden dieser Klasse. Die analoge Klasse zum Bearbeiten von DAO-Datenbanken heißt *CDAO-Database*.

15.4 Die Klasse CRecordset

CRecordset ist die virtuelle Schnittstelle zwischen Anwendung und Datenbank. Eine Instanz dieser Klasse führt eine SQL-Anweisung aus und stellt die damit selektierten Datensätze über spezielle Datenelemente und Klassenfunktionen in einer Datensatzgruppe der Anwendung zur Verfügung. Der Konstruktor hat einen optionalen Parameter, ein *CDatabase*-Objekt.

```
class Crecordset:  
    CRecordset( CDatabase* pDatabase = NULL);
```

Dadurch kann eine bereits geöffnete und parameterisierte Datenbank in der Datensatzgruppe verwendet werden. Wenn kein Parameter übergeben wird, legt die Klasse ein eigenes *CDatabase*-Objekt an.

In Datenbankanwendungen müssen Sie stets eine abgeleitete Klasse erzeugen, da spezielle Einstellungen, wie Tabellename und SQL-Abfrage, für den Datentransfer erforderlich sind und Klassenfunktionen zum typischen Zugriff auf die verschiedenen Datenfelder erzeugt werden müssen. Ein *CRecordset* ist damit immer an eine vorgegebene Datenstruktur gebunden.

Für den Datentransfer zwischen *CRecordset* und der Datenbank dient der *RFX*-Mechanismus.

```

void CAhnenSet::DoFieldExchange(CFieldExchange* pFX)
{
    //{{AFX_FIELD_MAP(CAhnenSet)
    pFX->SetFieldType(CFieldExchange::outputColumn);
    ...RFX_Int(pFX, _T("[lfd_Nr]"), m_lfd_Nr);
    RFX_Text(pFX, _T("[Name]"), m_Name);
    RFX_Text(pFX, _T("[Vorname]"), m_Vorname);
    RFX_Text(pFX, _T("[Geburtsname]"), m_Geburtsname);
    RFX_Date(pFX, _T("[Geburtsdatum]"), m_Geburtsdatum);
    RFX_Text(pFX, _T("[Geburtsort]"), m_Geburtsort);
    RFX_Date(pFX, _T("[Todestag]"), m_Todestag);
    RFX_Text(pFX, _T("[Bemerkung]"), m_Bemerkung);
    //}}AFX_FIELD_MAP
}

```

Listing: Verbindung zwischen Datensatzgruppe und Datenbankfeldern

Außerdem besitzt die Klasse zwei virtuelle Methoden zur Steuerung der SQL-Abfrage.

```

class Crecordset:
    virtual CString GetDefaultConnect();
    virtual CString GetDefaultSQL();

```

Über *GetDefaultConnect* erfährt die Klasse, wo sich die Datenbank befindet und mit welchen Parametern sie geöffnet werden soll. Die Assistenten tragen hier die Datenquelle ein, die Sie beim Anlegen der Klasse angegeben haben, z.B. eine registrierte ODBC-Datenbank.

```

CString CAhnenSet::GetDefaultConnect()
{
    return _T("ODBC; DSN=Ahnen");
}

```

Listing: Standardweg zur Datenbank

Mit *GetDefaultSQL* wird die Abfrage für die Datenbank ermittelt. Das Programmgerüst wertet die Angaben aus und baut daraus eine SQL-Abfrage zusammen. Die Assistenten tragen hier den Tabellennamen der Datenbank ein.

```

CString CAhnenSet::GetDefaultSQL()
{
    return _T("[Personen]");
}

```

Listing: Standardabfrage zum Füllen der Datensatzgruppe

Da beide Methoden *virtual* sind, werden sie vom Klassen- bzw. Anwendungs-Assistenten in der abgeleiteten Klasse immer überlagert.

Mit der Methode *GetSQL* erhalten Sie das Ergebnis der Auswertung von *GetDefaultSQL* durch das Programmgerüst. Sie können beide Methoden ändern und an Ihre Bedürfnisse anpassen. So ist es z.B. möglich, *GetDefaultConnect* die Zeichenkette *ODBC;DSN=* zurückzugeben. Dadurch ruft das Programmgerüst das Dialogfeld *Datenquelle auswählen* des ODBC-Managers auf, und der Anwender kann zwischen einer passenden Datei- oder Computer-Datenquelle auswählen. Dies ist z.B. sinnvoll, wenn Sie mit einer Anwendung auch eine DSN-Datei ausliefern.

Mit zwei Member-Variablen der Klasse *CRecordset* können Sie ebenfalls die SQL-Abfrage beeinflussen. Durch *m_strFilter* kann eine Bedingung bzw. ein Filter der Abfrage hinzugefügt werden. Das entspricht der WHERE-Klausel in SQL. Mit Hilfe der Variablen *m_strSort* können Sie der Abfrage eine Sortieranweisung hinzufügen. Diese entspricht der ORDER BY-Anweisung in SQL.

Darüber hinaus besitzt die Klasse Methoden zum Öffnen und Schließen von Datenbanken, zum Navigieren innerhalb der Tabelle und zum Ändern der Daten. Diese überschneiden sich jedoch zum Teil mit denen der Klasse *CDatabase* bzw. gehen auf diese zurück.

Für das Bearbeiten eines einzelnen Datensatzes aus der Datensatzgruppe steht die Methode *Edit* zur Verfügung. Damit wird der aktuelle Datensatz zum Bearbeiten freigegeben.

```
CRecordset::Edit  
    virtual void Edit( );  
        throw(CDBException, CMemoryException);
```

Mit der Methode *AddNew* wird ein neuer Datensatz erzeugt und ebenfalls zum Bearbeiten freigegeben. Beide Methoden setzen voraus, daß der angemeldete Benutzer in der Datenbank ändern darf und beim Öffnen die entsprechenden Optionen gesetzt wurden.

```
CRecordset::AddNew  
    virtual void AddNew( );  
        throw(CDBException);
```

Wenn die Änderungen beendet sind, muß der Datensatz noch in die Datenbank zurückgeschrieben werden. Dazu dient die Methode *Update*.

```
CRecordset::Update  
    virtual BOOL Update( );  
        throw(CDBException);
```

Die Methode *Update* ist so »schlau«, daß sie nur dann wirklich die Daten zurückschreibt, wenn sie sich seit dem letzten Sichern auch tatsächlich geändert haben. Sonst gibt sie FALSE zurück, was in diesem Fall eigentlich kein Fehler ist.

Alle drei Methoden zur Datensatzbearbeitung arbeiten mit einer Ausnahmebehandlung. Über diesen Mechanismus können sie auftretende Fehler abfangen (siehe Kapitel 15.8).

15.5 Die Klasse CRecordView

Die Klasse *CRecordView* übernimmt die Anzeige der Daten eines Datensatzes in einer Dialogvorlage. Seine Funktionalität ist voll auf diese Darstellung und Navigation ausgerichtet. Dafür besitzt die Klasse einen Zeiger auf ein *CRecordset*-Objekt, mit dem sie Zugriff auf die Datenfelder bekommt.

CRecordView hat als Basisklasse *CFormView* und bekommt im Konstruktor den Symbolnamen der Dialogvorlage übergeben.

```
class CRecordView:
    CRecordView( LPCSTR lpszTemplateName );
    CRecordView( UINT nIDTemplate );
```

CFormView vererbt auch alle Zugriffsmethoden auf die Steuerungen. So kommt auch hier der aus Dialogen bekannte DDX-Mechanismus zum Datenaustausch zum Einsatz.

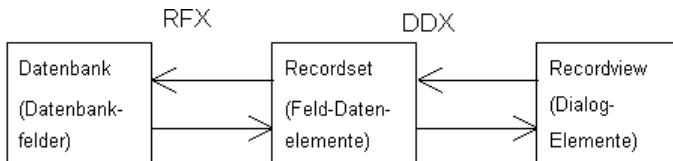


Abbildung 15.1: Datenaustausch bei Datenbankanwendungen

Für die Navigation in der Tabelle des *CRecordset*-Objektes stellt *CRecordView* die Methode *OnMove* zur Verfügung.

```
class CRecordView:
    virtual BOOL OnMove(UINT nIDMoveCommand);
    throw( CDBException );
```

Diese Methode akzeptiert vier Parameter:

1. `ID_RECORD_FIRST` – holt den ersten Satz in die Datensatzgruppe.
2. `ID_RECORD_LAST` – holt den letzten Satz in die Datensatzgruppe.
3. `ID_RECORD_NEXT` – holt den nächsten Satz in die Datensatzgruppe.
4. `ID_RECORD_PREV` – holt den vorherigen Satz in die Datensatzgruppe.

Die Menüfelder, die der Anwendungs-Assistent unter dem Menüpunkt DATENSATZ anlegt, werden vom Programmgerüst automatisch mit den Funktionen der Klasse *CRecordView* verknüpft, so daß die Navigation komplett durch die Assistenten implementiert wird. Je nach Position in der Tabelle werden sogar die entsprechenden Menüpunkte inaktiv.

Damit das alles funktionieren kann, muß die virtuelle Methode *OnGetRecordset* von *CRecordView* in der abgeleiteten Klasse überlagert werden. Sie liefert den Zeiger auf das aktuelle *CRecordset*-Objekt zurück, auf das sich die Steuerung durch die Basisklasse bezieht.

Außerdem muß die virtuelle Methode *OnInitialUpdate* überlagert werden, denn hier holt sich die Ansichtsklasse den Zeiger der Datensatzgruppe aus dem Dokument.

```
void CAhnenView::OnInitialUpdate()
{
    m_pSet = &GetDocument()->m_ahnenSet;
    CRecordView::OnInitialUpdate();
}
```

Listing: Zeiger auf die Datensatzgruppe setzen

Beim Anlegen der Klasse mit dem Anwendungs-Assistenten erfolgt das Überschreiben dieser Methoden automatisch.

15.6 ODBC-Konfiguration

Damit Programme auf ODBC-Datenquellen zugreifen können, ist die Installation der entsprechenden ODBC-Treiber eine Voraussetzung. Mit dem Visual C++ liefert Microsoft bereits wichtige Treiber mit aus, die bei der Installation der Software bei Bedarf mit auf Ihren PC installiert werden (siehe Kapitel 15.1.3).

Für die Herstellung einer Verbindung zwischen Ihrem Programm und einer ODBC-Datenbank benötigt die Anwendung Informationen über die ODBC-Datenbank. Diese Informationen werden unter dem Begriff DSN (Data Source Name) zusammengefaßt. Der *ODBC Driver Manager* verwendet diese Informationen zum Herstellen der Verbindung.

Diese Informationen müssen Sie einmalig mit dem *ODBC-Datenquellen-Administrator* festlegen. Den Administrator können Sie aus der Systemsteuerung heraus aufrufen.



Abbildung 15.2: Benutzer-Datenquellen im Administrator

Dabei kommen zwei unterschiedliche Verfahren zum Einsatz. DSN können entweder in der Windows-Registrierung gespeichert werden (Computer-DSN). Dazu zählen Benutzer- und System-DSN. Die entsprechenden Einträge können Sie sich mit dem Registrierungseditor unter dem Schlüsselwort ODBC ansehen. Von einem Bearbeiten im Editor ist jedoch abzuraten.

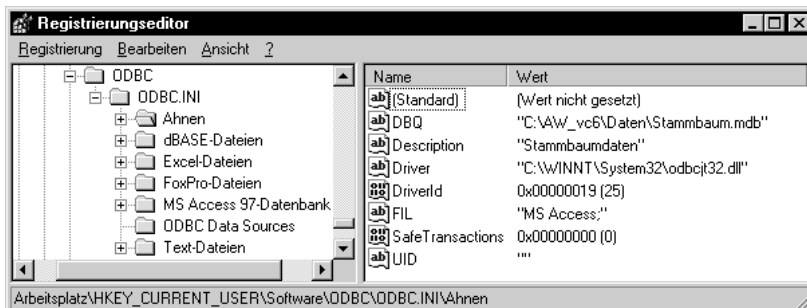


Abbildung 15.3: Benutzer-Datenquellen im Registrierungseditor

Der Datenquelleneintrag unter den System-DSN sieht analog aus. Er ist dann nur unter dem Schlüssel `Arbeitsplatz\HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\` in der Registerdatenbank zu finden.

Das zweite Verfahren ermöglicht das Speichern der Informationen in einer DSN-Datei. Diese Datei ist eine Textdatei mit der Erweiterung `.DSN`. Die Verbindungsformationen bestehen aus Parametern und entsprechenden Werten, die der *ODBC Driver Manager* zum Herstellen einer Verbindung verwenden kann. Diese Datei können Sie mit Ihrer Anwendung ausliefern. Dadurch entfällt die Registrierung der ODBC-Datenquelle auf den Zielsystemen. Die Datei hat etwa folgenden Aufbau:

```
[ODBC]
DRIVER=Microsoft Access-Treiber (*.mdb)
UID=admin
UserCommitSync=Yes
Threads=3
SafeTransactions=0
PageTimeout=5
MaxScanRows=8
MaxBufferSize=512
ImplicitCommitSync=Yes
FIL=MS Access
DriverId=25
DefaultDir=C:\AW_vc6\Daten
DBQ=C:\AW_vc6\Daten\Stammbaum.mdb
```

Listing: Aufbau einer DSN-Datei

Die DSN-Dateien werden vom System standardmäßig im Verzeichnis `LW:\PROGRAMME\GEMEINSAME DATEIEN\ODBC\DATA SOURCES` geschrieben und gesucht. Welche Informationen alles in einem DSN stehen, hängt vom jeweiligen Treiber ab. Bei der Registrierung mit dem *ODBC-Datenquellen-Administrator* wird ein Dialogfeld des ODBC-Treibers eingeblendet, der die erforderlichen Eingaben abfragt.

15.7 Assistentenunterstützung

15.7.1 Anwendungs-Assistent

Bei der Projektanlage mit dem Anwendungs-Assistenten haben Sie in Schritt 2 die Wahl zwischen vier Arten der Datenbankunterstützung.

Im einzelnen verbergen sich folgende Auswirkungen hinter den Optionen:

1. *Keine* – wählen Sie, wenn Sie keine Datenbankanwendung anlegen wollen oder über eigene Klassen auf eine Datenbank zugreifen wollen (z.B. DDE).
2. *Nur Header-Dateien* – bietet Ihnen die geringste Unterstützung in Ihrem MFC-Programm. Der Assistent fügt nur die Schnittstellendateien und Bibliotheken hinzu.
3. *Datenbankansicht ohne Dateiunterstützung* – Sie erhalten eine SDI-Datenbankanwendung, jedoch ohne Serialisationsfunktion im Dokument.
4. *Datenbankansicht mit Dateiunterstützung* – es kann eine MDI- oder SDI-Datenbankanwendung erstellt werden, die zusätzlich die Serialisationsfunktion im Dokument zur Verfügung stellt.

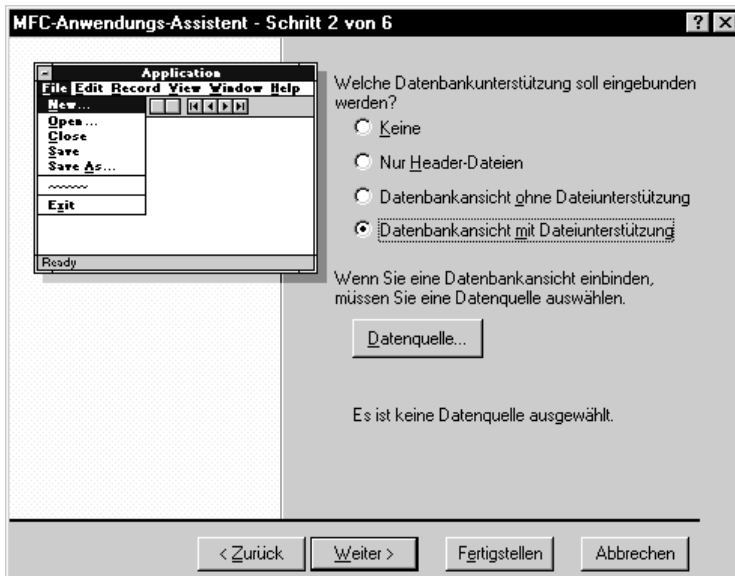
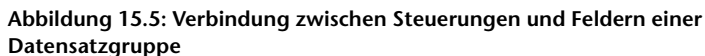


Abbildung 15.4: Datenbankunterstützung im Anwendungs-Assistenten

Über die Schaltfläche DATENQUELLE... teilen Sie dem Anwendungs-Assistenten die Art und den Ort der Datenquelle mit. Die Schaltfläche ist nur aktiv, wenn Sie eine der beiden Datenbankansichtsoptionen ausgewählt haben. Es erscheint ein Auswahldialog, in dem Sie zwischen einer ODBC- oder DAO-Datenquelle wählen können. Weiterhin können Sie einen Datensatzgruppentyp wählen. *Snapshot* ist vorbelegt. Sie können aber auch *Dynaset* und bei DAO-Datenquellen *Tabelle* auswählen.

Die Menüs und die Symbolleiste werden ebenfalls an die Datenbankorientierung der Anwendung angepaßt. Sie erhalten Einträge für die Navigation.

Zwei Besonderheiten bietet der Klassenassistent in Datenbankanwendungen. Als erstes ermöglicht er das Neuanlegen und die Verwaltung der MFC-Datenbankklassen *CRecordset* und *CDAORecordset*. Dies ist daher eine Besonderheit, da sonst nur Klassen, die von *CCmdTarget* abgeleitet sind, unterstützt werden. Der Assistent behandelt die Klassen als Fremdobjekte und ermöglicht es Ihnen, den Dialogdatenaustausch (DDX) auf diese Klassen anzuwenden. Das ist die zweite Besonderheit. Beim Hinzufügen von Variablen in einer von *CRecordView* oder *CDAORecordView* abgeleiteten Klasse wird eine direkte Verbindung zwischen dem Steuerungssymbolnamen und den Feldern der Datensatzgruppe hergestellt.



Diese Verknüpfung mit Fremdvariablen erleichtert das Einbinden von Feldern aus Datensatzgruppen erheblich; vor allem deshalb, da das Programmgerüst bei der Navigation in einer Datensatzgruppe, die sich in einer Ansichtsklasse befindet, die Aktualisierung automatisch vornimmt.

15.8 Datenbank-Ausnahmebehandlung der MFC

In diesem Abschnitt des Buches geht es um die Ausnahmebehandlung beim Umgang mit den MFC-Datenbankklassen. Ausnahmen treten dann ein, wenn ein Programm auf Grund von Bedingungen, die es nicht selbst beeinflussen kann, nicht mehr normal ausgeführt werden kann. Solche Bedingungen sind z.B. Speichermangel oder Ein- und Ausgabefehler. Abnormale Situationen müssen durch Auslösen und Empfangen von Ausnahmen behandelt werden. Für die Datenbankklassen können zwei verschiedene Mechanismen zum Einsatz kommen:

1. *MFC-Ausnahme-Makros* – mit einem umfangreichen Ausnahmebehandlungsmechanismus. Obwohl die Verwendung dieser Makros für die Entwicklung neuer Programme nicht mehr empfohlen wird, werden die Makros aus Gründen der Abwärtskompatibilität weiter unterstützt.
2. *C++-Ausnahmen* – die Programmiersprache selbst stellt einen eingebauten Mechanismus für die Behandlung zur Verfügung. Außergewöhnliche Situationen können jederzeit während der Ausführung Ihres Programms auftreten und mit Hilfe der C++-Ausnahmebehandlung an einen übergeordneten Ausführungskontext weitergereicht werden. Diese Ausnahmen werden durch Programmcode behandelt, der außerhalb des normalen Programmflusses liegt.

Es steht noch ein dritter Mechanismus zur Verfügung, die strukturierte Ausnahmebehandlung (SEH = structured exception handling). Diese basiert auf Funktionen des im Betriebssystem vorhandenen Ausnahmemechanismus. SEH wird jedoch für C++- oder MFC-Programmierung nicht empfohlen.

Dieser Abschnitt beschränkt sich auf die C++-Ausnahmebehandlung im Zusammenhang mit MFC-Datenbankklassen. Das Verfahren ist für DAO und ODBC annähernd das gleiche. In C++ wird eine Ausnahme mit dem Schlüsselwort *try* ausgelöst (siehe auch Kapitel 11.5.4). Diese Ausnahme wird dann von einer dafür vorgesehenen *catch*-Routine abgefangen.

```
void CAhnenDoc::OnNeu()  
{  
    try  
    {
```

```

        m_AhnenSet.AddNew();
    }
    catch (CDBException* e)
    {
        AfxMessageBox(e->m_strError);
        e->Delete();
        return;
    }
}

```

Listing: Ausnahmen bei Datenbankoperationen

Spezielle Klassen der MFC – *CDBExceptions* und *CDaoException* – ermöglichen es auf einfache Weise, aufgetretene Ausnahmen zu behandeln. Sie besitzen Member-Variablen, die alle benötigten Informationen enthalten. Die Klasse *CDBExceptions* besitzt die Variable *m_strError*, die einen Fehler-text enthält. Zusätzlich kann über *m_nRetCode* der ODBC-Rückgabewert ausgewertet werden. Bei *CDaoException* ist es ähnlich. Diese Klasse besitzt eine Variable *m_pErrorInfo*, die einen Zeiger auf ein *CDaoErrorInfo*-Objekt enthält. Das kapselt wiederum Fehlerinformationen in der DAO-Auflistung von Fehlerobjekten ein, die mit der Datenbank zusammenhängen.

Durch den konsequenten Einsatz der Ausnahmebehandlung machen Sie Ihr Programm robuster. Gerade bei Datenbankanwendungen ist dies wichtig, um Datenverluste zu vermeiden.

15.9 ODBC-Datenbankanwendung – ein Beispiel

15.9.1 Vorteile und Besonderheiten

Sobald es darum geht, größere Datenmengen in einer Satzstruktur zu verwalten, anzuzeigen und zu verändern, sollten Sie sich für eine Datenbankanwendung entscheiden. Das Developer Studio und auch die MFC unterstützen Datenobjekte in SDI- und MDI-Anwendungen, aber auch dialogbasierende Anwendungen.

Der Unterschied zu den bisherigen Anwendungen besteht u.a. in der Behandlung des Menüpunktes DATEI. Die Befehle NEU, ÖFFNEN, SPEICHERN und SPEICHERN UNTER haben in einer reinen Datenbankanwendung normalerweise keine Bedeutung, denn die Daten sind jetzt anders strukturiert in Datenbanken und Tabellen. Das Programm benutzt andere Methoden, um Datenbanken zu öffnen. Vor allem DATEI|NEU und SPEICHERN UNTER sind in solch einer Anwendung sinnlos. Diese Menüpunkte erfüllen ihren Zweck nur bei der Dokument-Serialisierung, die die Daten einschließlich der Objekte lesen und schreiben kann. Ihnen bleibt nichts weiter übrig, als diese Menüpunkte zu entfernen oder zumindest einige anders zu inter-

pretieren. Nutzen Sie z. B. den Befehl **ÖFFNEN** als »Datenbank öffnen«, und zeigen Sie dem Benutzer eine Liste der Datenquellen Ihrer Anwendung.

Eine weitere Besonderheit von Datenbankanwendungen besteht in der Möglichkeit, auch auf entfernte Datenquellen zuzugreifen und Client/Server-Anwendungen zu realisieren. Über einen ODBC-Treiber ist z. B. der Zugriff auf das SQL-Datenbanksystem gegeben. Bei der Nutzung der ODBC-Schnittstelle der MFC sind Sie in gewissen Grenzen in der Anwendung auch unabhängig von der jeweiligen Datenbankversion.

Einen kleinen Wermutstropfen gibt es allerdings auch. Da die Datentypen in den einzelnen Datenbanksystemen unterschiedlich sind, können die MFC-Klassen diese nicht immer korrekt abbilden, sondern es gilt der kleinste gemeinsame Nenner als gegeben. Besonders betroffen sind Datums- und Zeit-Datentypen, die es in sehr vielen Varianten unterschiedlicher Genauigkeit gibt. Deshalb kann die MFC-Klasse *CTime* dafür auch nicht ohne weiteres eingesetzt werden.

15.9.2 Aufgabe und Implementation

Die Aufgabe besteht darin, ein SDI-Beispielprogramm **AHNEN** zu realisieren. Bei diesem Programm handelt es sich um die Darstellung und Pflege eines Familienstammbaums.

Die Daten befinden sich in einer Tabelle **PERSONEN** einer Access-Datenbank **STAMMBAUM.MDB**. Das Datenmodell ist bewußt einfach gehalten, da der Schwerpunkt dieses Buches der Einsatz der MFC ist und nicht das Erstellen von Datenmodellen. Die Tabelle **Personen** hat folgende Felder:

Feldname	Typ / Länge	Bemerkung
lfd_Nr	int	siehe unten
Name	C / 20	Nachname
Vorname	C / 40	die Vornamen
Geburtsname	C / 20	Geburtsname, wenn vorhanden
Geburtsdatum	C / 10	Geburtsdatum im Datentyp Text
Geburtsort	C / 20	Ort der Geburt
Todestag	C / 10	Todestag im Datentyp Text
Bemerkung	Memo	freies Textfeld zum Lebenslauf

Tabelle 15.2: Datenfelder der Tabelle Personen

Die Zuordnung der Generationen erfolgt über das Feld *lfd_nr* der Tabelle. Es gibt eine einfache Formel, mit der die Hierarchie eines Stammbaumes abgebildet werden kann. Wenn Sie von einem beliebigen Knoten im Baum den Wert für *lfd_nr* der Eltern bestimmen wollen, gilt:

```
Vater.lfd_nr = aktuelle.lfd_nr * 2  
Mutter.lfd_nr = aktuelle.lfd_nr * 2 + 1
```

Umgekehrt ist der Wert für *lfd_nr* eines Kindes von einem Blatt aus gesehen immer über eine ganzzahlige Division durch zwei zu bestimmen.

```
Kind.lfd_nr = Vater.lfd_nr / 2  
Kind.lfd_nr = Mutter.lfd_nr / 2
```

oder

```
Kind.lfd_nr = Vater.lfd_nr >> 1  
Kind.lfd_nr = Mutter.lfd_nr >> 1
```

Dadurch kommt das Programm mit wenig Formeln aus, und eine Orientierung innerhalb des Stammbaumes ist an jeder beliebigen Stelle möglich, egal, in welche Richtung Sie navigieren wollen.

Die Darstellung auf dem Bildschirm erfolgt in zwei Ansichten: eine Baumansicht, die einen Überblick über die Struktur liefert, und eine Detailansicht, die alle Felder des Datensatzes enthält. In der Detailansicht soll gleichzeitig das Ändern und Erfassen von Datensätzen möglich sein. Beide Ansichten sind gleichzeitig auf dem Bildschirm in einem geteilten Rahmenfenster.

Die Datenbank wird über die ODBC-Schnittstelle der MFC angesprochen. Deshalb müssen Sie vor dem Erstellen des Projekts die Datenbank-Treiber mit dem ODBC-Administrator installieren. Dazu starten Sie das Administratorprogramm aus der Systemsteuerung. Fügen Sie unter *Benutzer-* oder *System-DSN* den *Microsoft Access-Treiber (*.mdb)* hinzu.

Über *AUSWÄHLEN...* geben Sie dem Treiber bekannt, wo sich die Datenbankdatei auf Ihrem System befindet. Speichern Sie die Einstellungen unter dem Namen *Ahnen* ab (siehe Abbildung 15.6).

Legen Sie zusätzlich noch ein *Datei-DSN* für die Datenbank an, und nennen Sie sie *Ahnen.dsn*.

15.9.3 Projektanlage mit dem Anwendungs-Assistenten

Auch bei diesem Beispiel werden Ihnen die Assistenten gute Dienste leisten. Doch zeigen sich hier erstmals einige Grenzen, die Sie nur durch eigenen Programmcode überwinden können.

Zunächst legen Sie ein neues SDI-MFC-Projekt *AHNEN* mit dem Anwendungs-Assistenten an. In Schritt 2 wählen Sie *Datenbankansicht ohne Dateiunterstützung* aus.

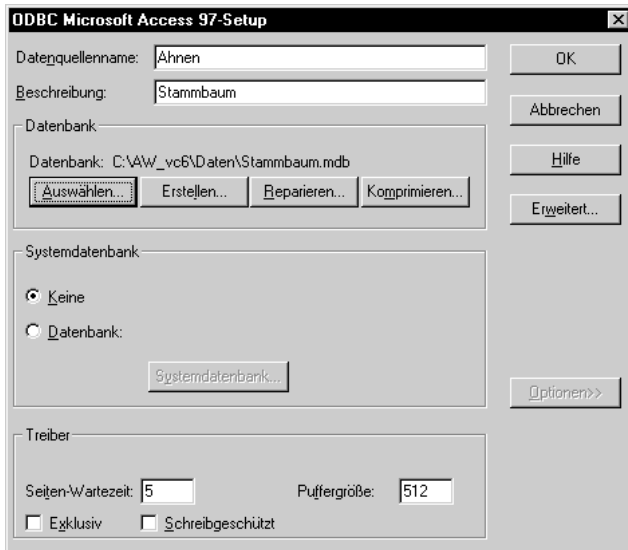


Abbildung 15.6: Hinzufügen der ODBC-Datenquelle Ahnen

Bei Auswahl von *Datenbankansicht mit Dateiunterstützung* können Sie auch eine MDI-Anwendung erzeugen.



Als nächstes betätigen Sie die Schaltfläche DATENQUELLE... in Schritt 2 des Anwendungs-Assistenten. Es erscheint folgender Auswahldialog:

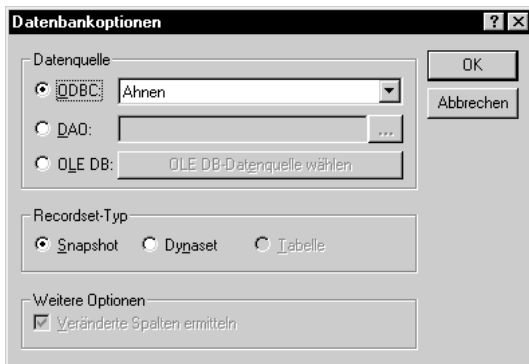


Abbildung 15.7: Auswahl der Datenquelle

Im Kombinationsfeld *ODBC* finden Sie alle auf Ihrem PC mit dem ODBC-Administrator registrierten *Benutzer-* und *System-DSN*. Dort steht Ihnen auch die bereits von Ihnen registrierte Datenbankquelle *Ahnen* zur Auswahl. Nun erscheint ein Dialogfeld zur Auswahl der Tabellen in der Datenbank. Es existiert nur die Tabelle *PERSONEN*, die Sie auswählen müssen.

Bei Schritt 3 können Sie die Option zur Unterstützung der *AktivX Steuerelemente* deaktivieren. In Schritt 4 setzen Sie die Anzahl für die Anzeige der Liste der zuletzt geöffneten Dateien auf 0. Unter *Weitere Optionen* in der Karteikarte *Fensterstile* aktivieren Sie *Geteilte Fenster verwenden* (siehe Kapitel 13.7).

Der Anwendungs-Assistent erzeugt neben den bereits bekannten Dateien für Dokument- und Ansichtsklasse zwei zusätzliche Dateien: `AHNENSET.CPP` und `AHNENSET.H`. Darin befindet sich die Deklaration und Implementierung einer von `CRecordset` abgeleiteten Klasse (`CAhnenSet`). Über sie erfolgt dann der Zugriff auf die Daten der Datenbank.

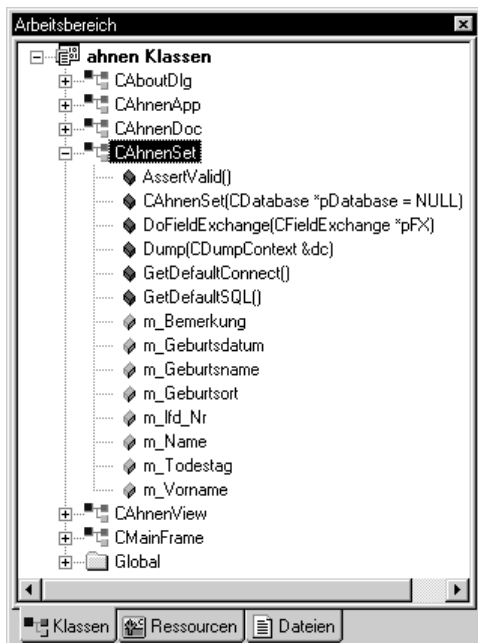


Abbildung 15.8: Die Datenfeldklasse `CAhnenSet`

Die Dokumentenklasse `CAhnenDoc` der Anwendung enthält jetzt eine Instanz der Klasse `CAhnenSet` mit dem Namen `m_ahnenSet`. Auch die Ansichtsklasse `CAhnenView` erhält eine Zeiger `m_pSet` auf diese Klasse.

Das Beispielprogramm hat ein geteiltes Fenster und soll zwei verschiedene Ansichten enthalten, die beide statisch eingebunden sind. Der Anwendungs-Assistent hat aber nur eine Ansichtsklasse `CAhnenView` und ein dynamisch geteiltes Fenster angelegt. Deshalb müssen Sie im nächsten Schritt eine weitere Ansichtsklasse anlegen. Als Basisklasse bietet sich `CTreeView` an, denn sie beinhaltet bereits die Klasse `CTreeCtrl` zur Darstel-

lung der Baumstruktur des Stammbaums. Fügen Sie also der Anwendung mit dem Klassenassistenten eine neue Klasse *CAhnenTreeView* hinzu. Die erforderliche Implementierungsdatei *<afxview.h>* müssen Sie selbst in die Datei *stdafx.h* einfügen, soweit gehen die Dienste des Klassenassistenten leider nicht.

Für die statische Teilung des Fensters muß die Methode *OnCreateClient* von *CMainFrame* angepaßt werden.

```

BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT /*lpcs*/,
    CCreateContext* pContext)
{
    if (!m_wndSplitter.CreateStatic(this, 1, 2))
    {
        TRACE0("Fehler bei CreateStaticSplitter\n");
        return FALSE;
    }
    // erste View anhängen
    if (!m_wndSplitter.CreateView(0, 1,
        pContext->m_pNewViewClass, CSize(0, 0), pContext))
    {
        TRACE0("Failed to create first pane\n");
        return FALSE;
    }
    // zweite View anlegen und anhängen
    if (!m_wndSplitter.CreateView(0, 0,
                                RUNTIME_CLASS(CAhnenTreeView),
                                CSize(250, 500), pContext))
    {
        TRACE0("Failed to create second pane\n");
        return FALSE;
    }
    SetActiveView((CView*)m_wndSplitter.GetPane(0,0));
    return TRUE;
}

```

Dabei ist es wichtig, die Ansicht *CAhnenView* zuerst einzubinden, denn deren Basisklasse *CRecordView* übernimmt in der Methode *OnInitialUpdate* das Öffnen der Datenbank und das Einlesen der Tabelle. Durch den Zeiger *m_pSet* hat die Basisklasse Zugriff auf das Datenfeld *m_ahnenSet* der Dokumentenklasse. Die neu angelegte Ansichtsklasse *CAhnenTreeView* wird über das Makro *RUNTIME_CLASS* eingebunden und bekommt auch eine Startgröße übergeben. Vergessen Sie nicht, die Implementierungsdatei für die neue Klasse einzufügen.

Erstellen Sie das Projekt und starten die Anwendung (Step1). Wenn Ihre Datenbank bereits Datensätze in der Tabelle enthält, sind die Navigations-Schaltflächen in der Symbolleiste aktiv, und die Daten werden von der Klasse *CAhnenSet* schon eingelesen. Doch eine Anzeige erfolgt noch nicht.

Um das zu überprüfen, starten Sie die Anwendung im Debug-Modus [F5]. Wenn das Programm geladen ist, wechseln Sie zurück in das Developer Studio und setzen einen Haltepunkt [F9] auf die virtuelle Methode *OnGetRecordset* der Klasse *CAhnenView*.

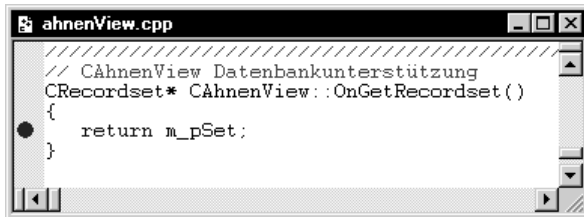


Abbildung 15.9: Haltepunkt in OnGetRecordset

Das Programm hält dann an dieser Stelle an, denn diese Methode wird von der Basisklasse *CRecordView* bei jedem Datenbankzugriff, der von ihr ausgeht, aufgerufen, um sich den benötigten Zeiger auf die Datensatzgruppe der abgeleiteten Klasse zu beschaffen. Im Fenster *Variablen* können Sie sich den Zeiger *m_pSet* ansehen. Wie Sie sehen, sind die Daten bereits eingelesen, ohne daß Sie eine Zeile selbst programmieren mußten.



Abbildung 15.10: Variablen in der Datensatzgruppe

Wenn Sie ein Projekt mit einer Datenbankanwendung bearbeiten, ist es oft von Vorteil, dem Arbeitsbereich ein Datenbankprojekt hinzuzufügen. Sie haben dann aus der Entwicklungsumgebung direkt Zugriff sowohl auf die Tabellen als auch auf die Daten in den Tabelle. Dazu wählen Sie *Neues Projekt dem Arbeitsbereich hinzufügen...* und suchen im Assistenten *Datenbank-Projekt* aus. Geben Sie ihm den Namen *AhnenDat*, und legen Sie es in das Verzeichnis DATEN.

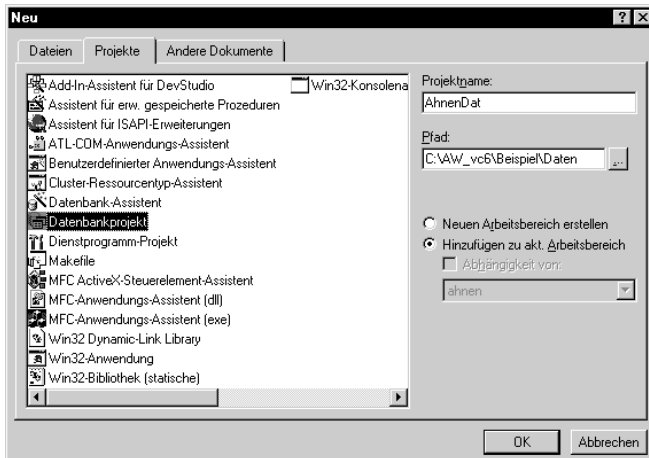


Abbildung 15.11: Datenbankprojekt dem Arbeitsbereich hinzufügen

Es erscheint dann der bereits bekannte Datenquellen-Dialog (siehe Abbildung 15.2) zur Auswahl einer registrierten ODBC-Datenquelle. Wählen Sie hier unter Computer-Datenquellen *Ahnen* aus.



Abbildung 15.12: Daten-Ansicht im Datenbankprojekt

Das *Arbeitsbereich*-Fenster wird um die *Daten-Ansicht* erweitert. Dort sehen Sie den Aufbau der Datenbank mit den Tabellen und Sichten.

Im *Editor*-Fenster ist das Bearbeiten der Tabellen wie unter Microsoft Access möglich. Sie können hier Daten verändern, löschen oder hinzufügen, aber auch SQL-Abfragen erstellen, testen und speichern.

15.9.4 Erweitern und Anpassen der Ressourcen

Die Dialogvorlage

Für die Ansichtsklasse *CAhnenView* muß nun die vom Anwendungs-Assistenten erstellte Dialogvorlage *IDD_AHNEN_FORM* angepaßt werden. Jedes Feld in der Tabelle soll ein entsprechendes Eingabefeld mit einem vorangestellten Textfeld erhalten. Außerdem ist eine Schaltfläche *SICHERN* vorzusehen, die aber standardmäßig nicht sichtbar und auch nicht aktiv ist.

Das Feld *lfd_Nr* wird im Programm nie eingegeben bzw. geändert werden können. Deshalb wird das Eingabefeld ebenfalls deaktiviert.

Bitmap für die Strukturansicht

Die Strukturansicht in der Klasse *CAhnenTreeView* soll mit einer Bildliste arbeiten. Die entsprechende Klasse *CImageList* erwartet dafür ein entsprechendes Bitmap. In dem Beispielprogramm soll es drei verschiedene Bilder in der Strukturansicht geben: eins für die Wurzel und je ein Symbol für Vater und Mutter. Da die Strukturansicht zwischen aktiviert und inaktiv unterscheiden soll, müssen insgesamt sechs Einzelbilder in einem Bitmap untergebracht werden.

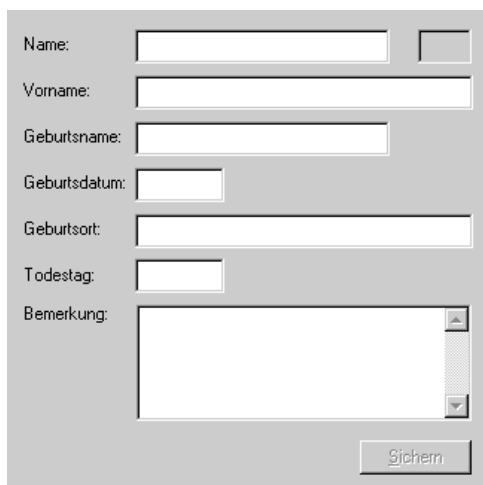
The image shows a standard Windows-style dialog box with a light gray background. It contains several labeled input fields: 'Name:' with a single-line text box and a small square icon to its right; 'Vorname:' with a single-line text box; 'Geburtsname:' with a single-line text box; 'Geburtsdatum:' with a date picker control; 'Geburtsort:' with a single-line text box; 'Todestag:' with a date picker control; and 'Bemerkung:' with a multi-line text area. At the bottom right, there is a button labeled 'Sichern'.

Abbildung 15.13: Dialogvorlage für *CAhnenView*

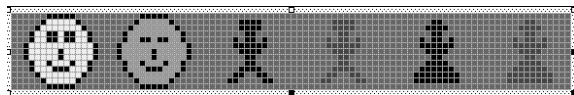


Abbildung 15.14: Bitmap für die Bildliste der Strukturansicht

Legen Sie ein neues Bitmap mit dem Symbolnamen `IDB_IMAGELIST` an. Zeichnen Sie die Bilder nebeneinander, und achten Sie darauf, daß alle genau 16 Punkte breit sind. Die Hintergrundfarbe darf in den einzelnen Bildern nicht vorkommen, denn sie wird später bei der Implementation ausgeblendet.

Die Symbolleiste

Leider kann von der Symbolleiste, die der Anwendungs-Assistent angelegt hat, nicht viel verwendet werden. Die Funktionalität der Zwischenablage können Sie komplett entfernen. Auch die Navigationsschaltflächen für die von *CRecordView* abgeleitete Klasse *CAhnenView* müssen entfernt werden. Das Ahnenprogramm bewegt sich nicht von Datensatz zu Datensatz sequentiell durch die Tabelle, sondern folgt einer Baumstruktur. Dadurch ist bei jedem Navigationsschritt ein Sprung innerhalb der Tabelle notwendig. Fügen Sie deshalb drei neue Navigationsschaltflächen in die Symbolleiste ein. Dabei haben die Schaltflächen folgende Bedeutung:

Schaltfläche	Symbolname
Zurück zum Kind	ID_RECORD_KIND
Wechsel zum Vater	ID_RECORD_VATER
Wechsel zur Mutter	ID_RECORD_MUTTER

Tabelle 15.3: Symbolnamen der Schaltflächen

Wenn Sie in Ihrer Anwendung keine Druckerunterstützung ausprogrammieren wollen, entfernen Sie auch diese Schaltfläche.



Abbildung 15.15: Die Symbolleiste der Anwendung

Das Menü und sonstiges

Auch am Menü ist einiges anzupassen. Der Menüpunkt `DATEI` kann unverändert bleiben, wenn Sie eine Druckerunterstützung wollen. Ansonsten entfernen Sie die betroffenen Menüpunkte. Bei den Menüpunkten `BEARBEITEN` und `DATENSATZ` gilt das gleiche wie bei der Symbolleiste. Die vom

Anwendungs-Assistenten eingefügten Untermenüpunkte können Sie alle entfernen. Unter BEARBEITEN fügen Sie ÄNDERN und SICHERN ein, unter DATENSATZ ZUM VATER, ZUR MUTTER und ZUM KIND, wobei Sie die selben Symbolnamen wie bei der Symbolleiste verwenden. Den Menüpunkt ANSICHT erweitern Sie durch die beiden Punkte EXPANDIEREN und ZUSAMMEN. Diese dienen der Steuerung der Strukturansicht.

Weitere Änderungen sind nicht notwendig, jedoch können Sie noch weitere Veränderungen vornehmen. So können Sie das Programmsymbol nach Belieben verändern. Auch die Dialogvorlage für den Dialog *Info über...* können Sie an Ihre Bedürfnisse anpassen.

15.9.5 Zusätzlicher Programmcode

Erweitern von CAhnenView

Die vom Anwendungs-Assistenten erzeugte Ansichtsklasse *CAhnenView* bringt über die Basisklasse *CRecordView* bereits einige Funktionalität mit. Doch kann, wie bereits erwähnt, nicht alles davon in dieser Anwendung genutzt werden, da die Navigationsgrundlage hier eine andere ist.

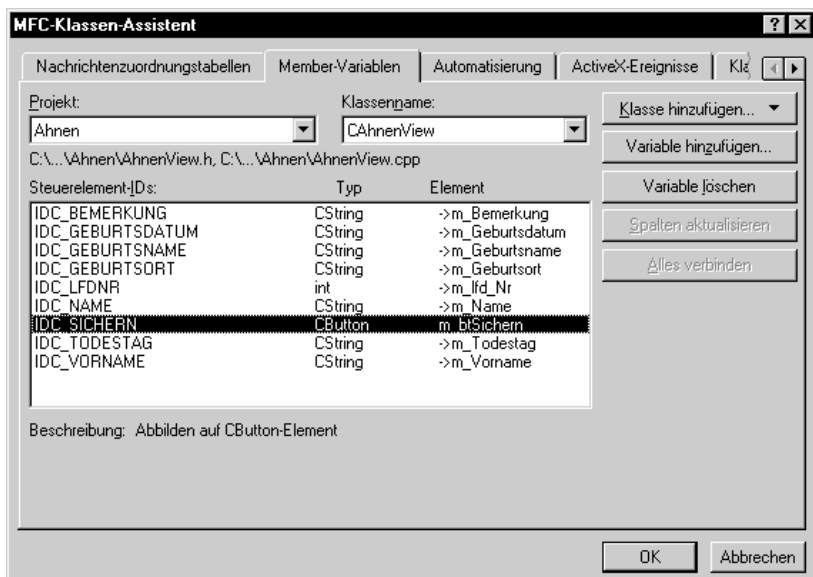


Abbildung 15.16: Variablen hinzufügen

Als erstes verbinden Sie die Steuerung der Dialogvorlage mit den Feldvariablen der Klasse *CAhnenSet*. Dazu rufen Sie den Klassenassistenten auf. Ein besonderer Vorzug des Klassenassistenten ist in diesem Fall die Unterstützung von Fremdvariablen. Dadurch ist es möglich, Member-Variablen

von `CAhnenSet` mit Steuerungen von `CAhnenView` über die Methode *DoDataExchange* zu verbinden. Die MFC stellt entsprechende DDX-Funktionen zur Verfügung. Im Dialog Member-Variablen hinzufügen steht jetzt eine Auswahl aller Member-Variablen der Klasse *CAhnenSet* durch den Zeiger *m_pSet* zur Auswahl. Verbinden Sie alle Steuerungen mit den dazugehörigen Variablen. Zusätzlich wird für die Schaltfläche SICHERN eine Member-Variable angelegt.

Bei den Variablen vom Typ *CString* können Sie auch die Länge der Zeichenkette eingeben. Das Programm prüft dann vor dem Übernehmen der Daten automatisch die eingegebene mit der zulässigen Länge und gib gegebenenfalls eine Fehlermeldung aus.

Das Ergebnis der Verbindung zwischen den Variablen und den Steuerungen sehen Sie wieder in der Methode *DoDataExchange* der Ansichtsklasse.

```
void CAhnenView::DoDataExchange(CDataExchange* pDX)
{
    CRecordView::DoDataExchange(pDX);
    if(!m_pSet->IsOpen())
        return;
    //{{AFX_DATA_MAP(CViewEinzel)
    DDX_Control(pDX, IDC_SICHERN, m_btSichern);
    DDX_FieldText(pDX, IDC_BEMERKUNG,
        m_pSet->m_Bemerkung, m_pSet);
    DDX_FieldText(pDX, IDC_GEBURTSNAME,
        m_pSet->m_Geburtsname, m_pSet);
    DDV_MaxChars(pDX, m_pSet->m_Geburtsname, 20);
    DDX_FieldText(pDX, IDC_GEBURTSDATUM,
        m_pSet->m_Geburtsdatum, m_pSet);
    DDV_MaxChars(pDX, m_pSet->m_Geburtsdatum, 10);
    DDX_FieldText(pDX, IDC_GEBURTSORT,
        m_pSet->m_Geburtsort, m_pSet);
    DDV_MaxChars(pDX, m_pSet->m_Geburtsort, 20);
    DDX_FieldText(pDX, IDC_LFDNR, m_pSet->m_lfd_Nr, m_pSet);
    DDX_FieldText(pDX, IDC_NAME, m_pSet->m_Name, m_pSet);
    DDV_MaxChars(pDX, m_pSet->m_Name, 20);
    DDX_FieldText(pDX, IDC_VORNAME, m_pSet->m_Vorname, m_pSet);
    DDV_MaxChars(pDX, m_pSet->m_Vorname, 40);
    DDX_FieldText(pDX, IDC_TODESTAG,
        m_pSet->m_Todestag, m_pSet);
    DDV_MaxChars(pDX, m_pSet->m_Todestag, 10);
    //}}AFX_DATA_MAP
}
```

Listing: Verbindung zwischen Steuerung und Datenbank

Fügen Sie die Abfrage nach dem Zustand der Datenbankfelder *m_pSet* ->*IsOpen* an den Anfang der Methode ein. Die Klasse *CRecordView* öffnet zwar automatisch in *OnInitialUpdate* die Datenbank, prüft aber nicht ab, ob es auch fehlerfrei funktioniert hat. Wenn dann durch das Programmgerüst *DoDataExchange* aufgerufen wird, bekommen Sie sonst an dieser Stelle Probleme.

Als nächstes ergänzen Sie die Methode *OnInitialUpdate*. Für das Einlesen der Daten in die Strukturansicht müssen die Datensätze nach dem Feld *lfd_nr* sortiert vorliegen. Deshalb muß der Filter entsprechend gesetzt werden.

```
void CAhnenView::OnInitialUpdate()
{
    m_pSet = &GetDocument()->m_ahnenSet;
    m_pSet->m_strSort = "lfd_Nr";
    CRecordView::OnInitialUpdate();
    GetDocument()->m_pRecordView = this;
    GetParentFrame()->RecalcLayout();
    ResizeParentToFit();
}
```

Listing: Erweitern von *OnInitialUpdate*

Außerdem bekommt das Dokument einen Zeiger auf diese Ansichtsklasse. Die Begründung dazu wird weiter unten gegeben.

Zur Steuerung der Anzeige fügen Sie mit dem Klassenassistenten die Methode *OnUpdate* ein. Hier reagiert die Ansichtsklasse darauf, wenn sich der aktuelle Datensatz geändert hat und neue Daten angezeigt werden müssen. Auch hier prüfen Sie zuerst, ob die Datenbank überhaupt geöffnet ist, damit keine unnötigen Fehler in der Anwendung auftreten. In einem zusätzlichen Parameter *lHint* wird mitgeteilt, ob die Ansichtsklasse in den Änderungsmodus wechseln soll. Wenn ja, wird die Datenfeldklasse ebenfalls in den Änderungsmodus gebracht und die Schaltfläche zum Sichern aktiviert.

```
void CAhnenView::OnUpdate(CView* pSender, LPARAM lHint,
                          COject* pHint)
{
    if(!m_pSet->IsOpen())
        return;
    UpdateData(FALSE);
    if(lHint == 2)
    {
```



```

    m_pSet->Edit();
    m_btSichern.EnableWindow();
    m_btSichern.ShowWindow(SW_SHOW);
}
}

```

Listing: Die Methode OnUpdate der Ansichtsklasse

Als letztes müssen Sie noch die Funktionalität zum Sichern implementieren. Dazu legen Sie zur Schaltfläche SICHERN eine Methode *OnSichern* an. Hier werden die Daten aus den Steuerungen übernommen und in die Datenbank durch die Methode *Update* zurückgeschrieben. Die Schaltfläche SICHERN wird wieder versteckt, und der zweiten Ansichtsklasse *CAhnenTreeView* wird über die Methode *UpdateAllViews* des Dokuments mitgeteilt, daß sich die Daten geändert haben.

```

void CAhnenView::OnSichern()
{
    UpdateData();
    m_btSichern.EnableWindow(FALSE);
    m_btSichern.ShowWindow(SW_HIDE);
    if(m_pSet->Update()
        GetDocument()->UpdateAllViews(this, 3);
}

```

Listing: Das Sichern der Daten

Die Funktionalität zum Drucken bleibt in diesem Beispiel unberücksichtigt. Sie können diese jedoch gern erweitern und den aktuellen Datensatz drucken lassen.

Ausbau von CAhnenTreeView

In dieser Ansichtsklasse ist am meisten zu tun, denn die vom Klassenassistenten angelegte Klasse kann noch nicht viel. Als erstes fügen Sie der Klasse einen Zeiger *m_pSet* auf die Klasse *CAhnenSet* hinzu. Um auf die Daten zugreifen zu können, muß die Ansichtsklasse einen Zeiger auf das entsprechende Dokument erhalten. Sie überlagern also die Methode *GetDocument* von *CView*, um einen typisierten Zugriff zu erhalten. Ähnlich wie bei *CAhnenView* legen Sie eine Methode *Inline* in der Schnittstellendatei und eine zum Debuggen in der Implementierungsdatei an.

```

#ifdef _DEBUG // Testversion in AhnenTreeView.cpp
inline CAhnenDoc* CAhnenTreeView::GetDocument()
{ return (CAhnenDoc*)m_pDocument; }
#endif
CAhnenDoc* CAhnenTreeView::GetDocument()

```

```
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CAhnenDoc)));
    return (CAhnenDoc*)m_pDocument;
}
```

Listing: Überlagern der Methode `GetDocument`



Vergessen Sie nicht die, Schnittstellendatei *AhnenDoc.h* von *CAhnenDoc* einzufügen.

Um das Aussehen der eingebetteten Klasse *CTreeView* zu verändern, müssen Sie die Methode *OnCreate* der Ansichtsklasse überlagern. Dort werden die Attribute `TVS_HASLINES|TVS_HASBUTTONS` an die Ansichtsklasse übergeben.

```
int CAhnenTreeView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    lpCreateStruct->style |=TVS_HASLINES|TVS_HASBUTTONS;
    return CTreeView::OnCreate(lpCreateStruct);
}
```

Listing: Anpassen der Strukturansicht

Dann überlagern Sie die Methode *OnInitialUpdate* der Basisklasse, um – wie bei *CAhnenView* – die Initialisierung vorzunehmen. Da *CAhnenView* bereits die Datenbank des Datenbankfeldes *CAhnenSet* geöffnet hat, können Sie bereits auf die Daten zugreifen. Weisen Sie den Zeiger *m_pSet* auf *m:ahnenSet* des Dokuments zu und übergeben den Zeiger der eigenen Ansichtsklasse dem Dokument. Um keine Zugriffe auf eine geschlossene Datenbank durchzuführen, prüfen Sie auch hier, ob das Öffnen fehlerfrei funktioniert hat.

Die Klasse *CImageList* unterstützt die Anzeige von Bitmaps in der Strukturansicht. Mit der Methode *SetImageList* kann der Klasse ein Zeiger auf eine Instanz von *CImageList* übergeben werden.

```
CImageList* SetImageList(CImageList * pImageList,
                          int nImageListType);
```

Legen Sie die Klasse *m_ctlImage* als Member der Ansichtsklasse an. Die Methode *Create* von *CImageList* hat viele Varianten (siehe Kapitel »Steuerungen«). Übergeben Sie den Symbolnamen `IDB_IMAGELIST` des Bitmaps, die Breite der Einzelbilder und die Farbe, die aus dem Hintergrund entfernt werden soll.

Über die Methode *GetTreeCtrl* von *CTreeView* erhalten Sie einen Zeiger auf die eingebettete Klasse *CImageList*. Weisen Sie ihr darüber die Bildliste zu.

```

void CAhnenTreeView::OnInitialUpdate()
{
    CTreeView::OnInitialUpdate();
    m_pSet = &GetDocument()->m_ahnenSet;
    GetDocument()->m_pTreeView = this;
    if(!m_pSet->IsOpen())
        return;
    m_ctlImage.Create(IDB_IMAGELIST,16, 0, RGB(255,0,255));
    m_ctlImage.SetBkColor(GetSysColor(COLOR_WINDOW));
    GetTreeCtrl().SetImageList(&m_ctlImage, TVSIL_NORMAL);
    //...
}

```

Listing: Initialisieren des Listenfeldes

Um den Stammbaum in die Strukturansicht zu übernehmen, sind zwei Varianten möglich. Bei der einen werden jedesmal beim Aufklappen eines Knotens die Daten aus dem Datenbankfeld gelesen und in die Strukturansicht übernommen. Diese dynamische Ansicht hat den Vorteil, daß sie nur die Daten bei Bedarf ermittelt und deshalb sehr schnell ist. Der Windows-Explorer arbeitet auch nach diesem Prinzip. Erst wenn ein Knoten aufgeklappt wird, werden die Daten der darunterliegenden Ebene ermittelt.

Bei der zweiten Variante werden gleich am Anfang alle Knoten und Blätter eingefügt, was eine längere Ladezeit zur Folge haben kann. Dafür gibt es dann beim Aufklappen der Knoten keine Verzögerung. Wenn die Anzahl der Datensätze gering ist, ist diese Methode günstiger. In diesem Beispiel kommt die zweite Variante zum Einsatz.

Die Knoten in der Strukturansicht werden über ein Handle des Elternknotens miteinander verzeigert. Das heißt, jeder Eintrag in dem Baum hat einen Zeiger auf den Elternknoten, außer der Wurzel. Da die Datensätze in der Tabelle nicht in der Reihenfolge vorliegen können, wie sie in der Strukturansicht benötigt werden, müssen Sie sich eine Hilfsstruktur schaffen, die das Feld *lfd_Nr* und das dazugehörige Handle des Eintrages enthält. Dazu eignet sich die Klasse *CMapWordToPtr* der MFC. Diese Map ermöglicht es Ihnen, über die laufende Nummer eines Feldes das dazugehörige Handle zu bestimmen. Die Datensätze werden durch die Sortieranweisung mit aufsteigender laufender Nummer geliefert. Da Sie davon ausgehen können, daß zu jeder Nummer ein Kind existieren muß und dessen Nummer ja kleiner ist, können Sie aus der Map das zugehörige Kind mit dem Handle bestimmen.

In der Struktur *TV_INSERTSTRUCT* werden jedem neuen Element die erforderlichen Daten und Eigenschaften übergeben. In das Feld *lParam* setzen Sie den Wert *lfd_Nr* des aktuellen Datensatzes. Für das Anfügen eines

Blattes in die Struktur legen Sie eine Methode *NeuesBlatt* für die Ansichts-klasse an. Hier werden alle Daten zusammengestellt und mit *InsertItem* der Strukturansicht hinzugefügt.

```
void CAhnenTreeView::NeuesBlatt()
{
    TV_INSERTSTRUCT TreeCtrlItem;
    HTREEITEM hTreeItem;
    CString strName;
    strName = m_pSet->m_Name + ", ";
    strName += m_pSet->m_Vorname;
    TreeCtrlItem.hInsertAfter = TVI_LAST;
    TreeCtrlItem.item.mask = TVIF_TEXT | TVIF_PARAM |
        TVIF_IMAGE | TVIF_SELECTEDIMAGE;
    TreeCtrlItem.hParent =
        (HTREEITEM)m_TreeMap[m_pSet->m_lfd_Nr / 2];
    TreeCtrlItem.item.pszText = (LPTSTR)(LPCTSTR)strName;
    TreeCtrlItem.item.lParam = m_pSet->m_lfd_Nr;
    TreeCtrlItem.item.iImage = m_pSet->m_lfd_Nr % 2 ? 4 : 2;
    TreeCtrlItem.item.iSelectedImage =
        m_pSet->m_lfd_Nr % 2 ? 5 : 3;
    hTreeItem = GetTreeCtrl().InsertItem(&TreeCtrlItem);
    m_TreeMap.SetAt(m_pSet->m_lfd_Nr, hTreeItem);
}
```

Listing: Methode zum Einfügen neuer Elemente in die Strukturansicht

Diese Funktion wird in *OnInitialUpdate* in einer Schleife aufgerufen. Dabei wird der Datensatzzeiger bei jedem Durchlauf weiter gesetzt, bis das Ende des Datenbankfeldes erreicht wird.

```
while (!m_pSet->IsEOF())
{
    NeuesBlatt();
    m_pSet->MoveNext();
}
```

Listing: Schleife über allen Datensätze

Damit ist die Initialisierungsarbeit abgeschlossen. Nun müssen Sie noch eine Methode implementieren, mit der Sie die Ansichts-klasse *CAhnenView* informieren, wenn der Benutzer einen neuen Knoten oder ein neues Blatt auswählt, damit die Daten in der Dialogvorlage angezeigt werden können. Die Methode *OnSelchanged* von *CTreeView* wird vom Programmgerüst jedesmal aufgerufen, wenn ein Eintrag aktiviert bzw. deaktiviert wird. Dabei können Sie unterscheiden, ob das mit der Maus oder über die Tastatur

erfolgt. Sie müssen nur diese Methode überlagern und die Aktion auswerten. Wenn ein anderer Eintrag selektiert wurde, setzen Sie den Filter des Datenfeldes und führen eine erneute Abfrage mit *Requery* durch. Nutzen Sie die Methode *UpdateAllViews* des Dokuments, um die andere Ansichts-klasse zu informieren.

```
void CAhnenTreeView::OnSelchanged(NMHDR* pNMHDR, LRESULT* pResult)
{
    NM_TREEVIEW* pNMTreeView = (NM_TREEVIEW*)pNMHDR;
    if ((pNMTreeView->action == 1) || (pNMTreeView->action == 2))
    {
        m_pSet->m_strFilter.Format("lfd_nr = %d",
                                   pNMTreeView->itemNew.lParam);
        m_pSet->Requery();
        GetDocument()->UpdateAllViews(this);
    }
    *pResult = 0;
}
```

Listing: Ein neuer Eintrag wurde ausgewählt

Über die Methode *OnUpdate* erfährt die Klasse *CAhnenTreeView*, wenn sie ihre Anzeige ändern muß. Dabei sind drei verschiedene Varianten zu berücksichtigen. Die Unterscheidung erfolgt auf Grund des in *lHint* übergebenen Parameters.

1. Der Benutzer hat über das Menü oder die Symbolleiste Vater, Mutter oder Kind gewählt. Das Dokument hat bereits den entsprechenden Datensatz ausgewählt. Sie müssen nur den dazugehörigen Eintrag in der Strukturansicht selektieren.
2. Es wird ein neues Blatt eingefügt oder zum Bearbeiten ausgewählt. Die Ansichts-klasse wird deaktiviert, solange die Klasse *CAhnenView* im Änderungsmodus ist.
3. Der Änderungsmodus wird beendet. Die Anzeige muß aktualisiert werden, daß sich Name und Vorname geändert haben können.

```
void CAhnenTreeView::OnUpdate(CView* pSender,
                              LPARAM lHint, CObject* pHint)
{
    if (lHint == 1)
        GetTreeCtrl().Select((HTREEITEM)m_TreeMap[m_pSet->m_lfd_Nr]
                              ,TVGN_CARET);

    if (lHint == 2)
    {
        EnableWindow(FALSE);
    }
}
```

```

        NeuesBlatt();
    }
    if (lHint == 3)
    {
        EnableWindow();
        CString strName = m_pSet->m_Name + ", ";
        strName += m_pSet->m_Vorname;
        GetTreeCtrl().SetItem(
            (HTREEITEM)m_TreeMap[m_pSet->m_lfd_Nr], TVIF_TEXT,
            (LPTSTR)(LPCTSTR)strName, 0, 0, 0, 0, 0 );
    }
}

```

Listing: Aktualisieren der Ansicht

Jetzt fehlen nur noch die beiden Funktionen, die vom Menü aus einen Knoten expandieren oder zusammenklappen lassen. Legen Sie die beiden Funktionen für die Menüpunkte ANSICHT|EXPANDIEREN und ANSICHT|ZUSAMMEN an und rufen dort die Methode *Expand* der Klasse *CTreeCtrl* auf.

```

void CAhnenTreeView::OnAnsichtExpandieren()
{
    GetTreeCtrl().Expand(
        (HTREEITEM)m_TreeMap[m_pSet->m_lfd_Nr], TVE_EXPAND);
}
void CAhnenTreeView::OnAnsichtZusammen()
{
    GetTreeCtrl().Expand(
        (HTREEITEM)m_TreeMap[m_pSet->m_lfd_Nr], TVE_COLLAPSE);
}

```

Listing: Auf- und Zuklappen eines Knotens

Erweitern von CAhnenDoc

Auch die Dokumentenklasse muß erweitert werden. In diesem Beispiel wird sie vor allem auf Menünachrichten reagieren bzw. die Menüpunkte steuern. Zur Navigation im Stammbaum gibt es drei Richtungen: *zum Vater*, *zur Mutter* und *zum Kind*. Die laufende Nummer des Zieldatensatzes läßt sich über die bekannten Formeln leicht bestimmen. Das Dokument muß dann nur entscheiden, ob der entsprechende Datensatz aktiviert und angezeigt werden soll oder ob es sich um einen neuen Datensatz handelt.

Dafür legen Sie eine Methode *CreateOrActivateRecord* in *CAhnenDoc* an, die als Parameter die neue aktuelle laufende Nummer erwartet. Mit *Requery* prüfen Sie, ob die übergebende Nummer bereits in der Tabelle vorhanden

ist. Wenn nicht, wird ein neuer Datensatz durch die Methode *AddNew* angelegt und das Feld *lfd_Nr* belegt. Anschließend werden die Ansichtsklassen durch *UpdateAllViews* informiert. Dabei werden die oben erwähnten Parameter übergeben. Ist dagegen der Datensatz vorhanden, werden nur die Ansichtsklassen benachrichtigt.

```
void CAhnenDoc::CreateOrActivateRecord(int nLfdNrNeu)
{
    // Filter auf aktuelle laufende Nummer und neue Abfrage
    m_ahnenSet.m_strFilter.Format("lfd_nr = %d", nLfdNrNeu);
    m_ahnenSet.Requery();
    // keinen passenden Datensatz gefunden
    if (m_ahnenSet.IsEOF())
    {
        if(AfxMessageBox("Datensatz nicht vorhanden! \
                          Jetzt anlegen?", MB_YESNO)==IDNO)
            return;
        // Neuanlage möglich
        if(!m_ahnenSet.CanAppend())
        {
            AfxMessageBox("Anhängen nicht möglich!");
            return;
        }
        m_ahnenSet.AddNew();
        m_ahnenSet.m_lfd_Nr = nLfdNrNeu;
        m_ahnenSet.m_Name = "Neu";
        if (!m_ahnenSet.Update())
            AfxMessageBox("Fehler beim Anhängen!");
        else
        {
            m_ahnenSet.Requery();
            UpdateAllViews(NULL, 2);
        }
        return;
    }
    UpdateAllViews(NULL, 1);
}
```

Listing: Aktivieren oder Neuanlegen eines Datensatzes

Nun sind die drei Menüpunkte zur Navigation schnell programmiert. Sie müssen nur die neuen laufenden Nummern berechnen und die Funktion aufrufen.

```

void CAhnenDoc::OnRecordVater()
{
    CreateOrActivateRecord(m_ahnenSet.m_lfd_Nr * 2);
}
void CAhnenDoc::OnRecordMutter()
{
    CreateOrActivateRecord(m_ahnenSet.m_lfd_Nr * 2 + 1);
}
void CAhnenDoc::OnRecordKind()
{
    CreateOrActivateRecord(m_ahnenSet.m_lfd_Nr / 2);
}

```

Listing: Die Navigationsfunktionen

Der Menüpunkt BEARBEITEN|ÄNDERN ist auch nicht kompliziert. Die Strukturansicht muß während des Änderungsmodus deaktiviert werden, und die Klasse *CAhnenView* soll in den Bearbeitungsmodus wechseln. Dazu wird wieder die Methode *UpdateAllViews* verwendet.

```

void CAhnenDoc::OnBearbeitenAendern()
{
    if(!m_ahnenSet.CanUpdate())
    {
        AfxMessageBox("Aendern nicht möglich!");
        return;
    }
    m_pTreeView->EnableWindow(FALSE);
    UpdateAllViews(m_pTreeView, 2);
}

```

Listing: Wechsel in den Bearbeitungsmodus

Eine letzte Steuerungsfunktion soll das Dokument noch übernehmen. Der erste Eintrag in der Tabelle, die Wurzel, darf nicht geändert werden. Dazu soll der betreffende Menüpunkt deaktiviert werden, wenn der Datensatz angezeigt wird. Dazu dient die Nachricht *UPDATE_COMMAND_UI* des Menüpunktes BEARBEITEN|ÄNDERN. In der zugehörigen Funktion wird der Menüpunkt gesteuert.

```

void CAhnenDoc::OnUpdateBearbeitenAendern(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(m_ahnenSet.m_lfd_Nr > 1);
}

```

Listing: Steuerung des Menüpunktes Ändern

15.9.6 Fazit und Anregungen

Das Beispielprogramm stellt noch keinen hohen Standard dar. Es wurden jedoch die wichtigsten Möglichkeiten der Assistentenunterstützung und der MFC-Datenbankklassen beschrieben. Das Thema Ausnahmebehandlung blieb im ganzen Beispiel außen vor. Kerngedanke war die Einführung in die Datenbankprogrammierung. Dieses Thema ist zu umfangreich, um in diesem Beispiel umfassend erklärt zu werden.

Das Beispiel eignet sich sehr gut für Erweiterungen. So können Sie eine Funktion zum Löschen realisieren, die von einem Knoten aus alle darunterliegenden Ebenen löscht. Oder Sie implementieren in jeder Ansichts-klassse eine Druckfunktionalität, die je nachdem nur einen Datensatz oder den gesamten Baum druckt. Versuchen Sie die Strukturansicht für das dynamische Aufbauen umzustellen.

Wie Sie sehen, bleibt noch genügend Arbeit an dem Programm, damit es auch höheren Ansprüchen gerecht wird. Wenn Ihnen das noch nicht reicht, bauen Sie doch noch das Hilfesystem ein.

Den Quellcode für das Programm finden Sie natürlich auf der beiliegenden CD.



15.10 Erweiterte SQL-Funktionen

Die SQL-Abfragesprache ermöglicht nicht nur die einfache Selektion von Datensätzen. Einige Schlüsselwörter erlauben es beispielsweise, numerische Felder der gefundenen Datensätze zu summieren, Datensätze mit identischen Schlüsselfeldern nur einmal in der Auswahl erscheinen zu lassen usw. Alle diese Schlüsselwörter werden in der SELECT-Klausel in der Feldliste verwendet. Folgende numerische Funktionen sind möglich:

Funktion	Beschreibung
SUM	Berechnet innerhalb der Auswahl die Summe eines Feldes.
MIN	Findet in einer Auswahl den Minimalwert eines Feldes, nicht aber den jeweiligen Datensatz.
MAX	Findet in einer Auswahl das Maximum eines Feldwertes, nicht aber den jeweiligen Datensatz.
AVG	Ermittelt in einer Auswahl den Mittelwert eines Feldes.
COUNT	Zählt die Anzahl der gefundenen Datensätze.
DISTINCT	Das Schlüsselwort bewirkt, daß Datensätze mit identischem Schlüssel nur einmal in der Auswahl erscheinen.
FIRST	Lokalisiert den ersten Datensatz, der die Auswahlkriterien erfüllt.
LAST	Findet den letzten Datensatz, der den Auswahlkriterien entspricht.

Tabelle 15.4: Numerische SQL-Funktionen

Sofern der Funktion *Open* der Klasse *CRecordset* keine vollständige SQL-Anweisung als Parameter übergeben werden soll, können Sie die oben erwähnten Schlüsselwörter innerhalb von *DoFieldExchange* benutzen. In der RFX-Übertragungsfunktion wird der Feldname durch den entsprechenden Ausdruck ersetzt, beispielsweise durch `RFX_Long(pFX, "Sum(LagerBestand)", m_nLagerBest)`.

Der RFX-Mechanismus trägt das Ergebnis in die *CRecordset*-Variable `m_nLagerBest` ein. Soll dieser Wert in einer Dialogvorlage angezeigt werden, ist natürlich auch der DDX-Mechanismus zu verwenden: `DDX_FieldText(pDX, IDC_SUMBESTAND, m_pSet->m_nLagerBest, m_pSet);`.

Es können auch gespeicherte Prozeduren der Datenbank ausgeführt werden. Außerdem werden die Zugriffe auf Systemtabellen unterstützt.



Anhang

A

Die folgende Tabelle zeigt den Standard-Windows-Zeichensatz (Code-Page 1252).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0																
16																
32		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
128			,	f	„	...	†	‡	^	‰	Š	<	Œ			
144		'	'	“	”	•	-	-	~	™	š	>	œ			Ÿ
160		ı	ç	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
176	°	±	²	³	'	µ	¶	·	,	¹	º	»	¼	½	¾	¿
192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
208	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
240	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

B

In der folgenden Liste finden Sie eine Gegenüberstellung deutscher und englischer Begriffe, die in Zusammenhang mit Visual C++ verwendet werden. Diese Liste stellt nur eine Auswahl dar und erhebt keinen Anspruch auf Vollständigkeit. Wie Sie vielleicht schon beim Lesen des Buches bemerkt haben, wurde die deutsche Version von Visual C++ zu Grunde gelegt und versucht, die Begriffe der deutschen Version zu verwenden. Sicher ist das nicht an allen Stellen gelungen. Einige Hinweise dazu sind auch im Text zu finden. Hauptsächlich für die Benutzer der englischen Version wurde dieser Abschnitt eingefügt und deshalb auch nach den deutschen Begriffen alphabetisch sortiert.

Deutsch	englisch
abgerundetes Rechteck	Round Rect
Abhängigkeiten	Dependencies
Allgemein	General
Ansicht	View
Anwendungs-Assistent	AppWizard
Arbeitsbereich	(Project) Workspace
Assistentenleiste	WizardBar
Aufrufliste	Call Stack
Ausgabe	Output
Ausnahmen	Exceptions
ausrichten	align
Basisklasse	Base class
Bearbeitungsfeld	Edit Control
Beschleuniger	Accelerator

Deutsch	englisch
Beschriftung	Caption
Besitzerzeichnung	Owner Draw
Bibliothek	Library
Bildlaufleiste	Scrollbar
Bildliste	Imagelist
Bleistift	Pencil
Datenbankansicht	Database View
Datenquelle	Data Source
Deaktiviert	Disable
Drehfeld	Spinbutton
Durchsuchen	Browser
Eigenschaften	Properties
Einstellungen	Settings
Erstellen	Build
erweiterte Formate	extended Styles
Flach	Flat
Formate	Styles
Führungslinien	Layout Guides
Füllen	Fill
gefülltes Rechteck	Filled Rectangle
Gerät	Device
Grau	Grayed
Haltepunkt	Breakpoint
Info über	Aboutbox
Klassen-Assistent	ClassWizard
Kombinationsfeld	Combobox
Kontextabhängig	Context-sensitiv
Kontrollkästchen	Checkbox
Lesezeichen	Bookmark
Löschen	Erase
Mehrzeilig	Multi-line
Nachricht	Message
Nachrichtenzuordnungstabelle	Message Map
Optionsfeld	Radio Button
Pinself	Brush
Programmfaden	Thread

Deutsch	englisch
Rand	Border
Rasterweite	Grid Spacing
Rechteck	Rectangle
Registerkarte	Tab Control
Regler	Spin Button Control
Schaltfläche	Button
Schieberegler	Slider
Schnellüberwachung	QuickWatch
Schnittstellendatei	Header File
Schriftart	Font
Schriftgrad	Font Size
Seitenansicht	Print Preview
Sichtbar	Visible
Slider	Regler
Speicher	Memory
Statusleiste	Statusbar
Statuszeilentext	Prompt
Steuerungen	Controls
Strukturansicht	Tree Control
Symbol	Icon
Symbolbild	Image
Symbolleiste	Toolbar
Tastenbefehl	Hotkey
Teilung	Break
Titelleiste	Titelbar
Trennlinie	Separator
Überwachung	Watch
Verbindung	Connection
Vergrößern	Magnify
Weiter Optionen	Advanced
weitere Formate	More Styles
zu erledigen	Todo
Zugriffstaste	Hotkey

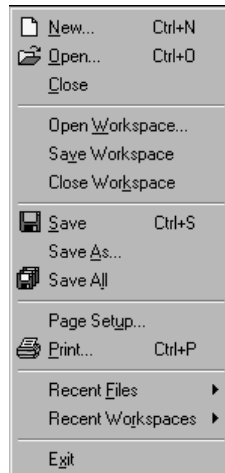


Anhang

C

Nachdem im Anhang B die Begriffe der deutschen und englischen Version gegenüber gestellt wurden, finden Sie in diesem Abschnitt eine Gegenüberstellung der Menüs beider Sprachen.

DATEI – FILE



BEARBEITEN – EDIT

Rückgängig	Ctrl+Z
Wiederherstellen	Ctrl+Y
Ausschneiden	Ctrl+X
Kopieren	Ctrl+C
Einfügen	Ctrl+V
Löschen	Del
Alles markieren	Ctrl+A
Suchen...	Ctrl+F
Suchen in Dateien...	
Ersetzen...	Ctrl+H
Gehe zu...	Ctrl+G
Lesezeichen...	Alt+F2
Weitere Optionen	▶
Haltepunkte...	Alt+F9
Elemente auflisten	Ctrl+Alt+T
Typinfo	Ctrl+T
Parameterinfo	Ctrl+Shift+Space
Wort vervollständigen	Ctrl+Space

Undo	Ctrl+Z
Redo	Ctrl+Y
Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Delete	Del
Select All	Ctrl+A
Find...	Ctrl+F
Find in Files...	
Replace...	Ctrl+H
Go To...	Ctrl+G
Bookmarks...	Alt+F2
Advanced	▶
Breakpoints...	Alt+F9
List Members	Ctrl+Alt+T
Type Info	Ctrl+T
Parameter Info	Ctrl+Shift+Space
Complete Word	Ctrl+Space

WEITERE OPTIONEN – ADVANCED

Inkrementelle Suche	Ctrl+I
Auswahl formatieren	Alt+F8
Tab's statt Leerzeichen	
Leerzeichen statt Tab's	
Auswahl in Großbuchstaben umwandeln	Ctrl+Shift+U
Auswahl in Kleinbuchstaben umwandeln	Ctrl+U
a·b Leerraum einblenden	Ctrl+Shift+8

Incremental Search	Ctrl+I
Format Selection	Alt+F8
Tabify Selection	
Unify Selection	
Make Selection Uppercase	Ctrl+Shift+U
Make Selection Lowercase	Ctrl+U
a·b View Whitespace	Ctrl+Shift+8

ANSICHT – VIEW

Klassen-Assistent...	Ctrl+W
ID= Ressourcensymbole...	
Ressourcen-Includes...	
Ganzer Bildschirm	
Arbeitsbereich	Alt+0
Ausgabe	Alt+2
Debug-Fenster	▶
Aktualisieren	
Eigenschaften	Alt+Enter

Class Wizard...	Ctrl+W
ID= Resource Symbols...	
Resource Includes...	
Full Screen	
Workspace	Alt+0
Output	Alt+2
Debug Windows	▶
Refresh	
Properties	Alt+Enter

DEBUG-FENSTER – DEBUG WINDOWS

Überwachung	Alt+3
Aufrufliste	Alt+7
Speicher	Alt+6
Variablen	Alt+4
Register	Alt+5
Disassemblierung	Alt+8

W <u>atch</u>	Alt+3
C <u>all Stack</u>	Alt+7
M <u>emory</u>	Alt+6
V <u>ariables</u>	Alt+4
R <u>egisters</u>	Alt+5
D <u>isassembly</u>	Alt+8

EINFÜGEN – INSERT

N <u>eue Klasse...</u>
N <u>eu</u> es F <u>ormular...</u>
R <u>essource...</u> Ctrl+R
R <u>essourcen</u> kopie...
 N <u>eu</u> es ATL- <u>O</u> bjekt...

N <u>e</u> w C <u>l</u> ass...
N <u>e</u> w F <u>o</u> rm...
R <u>e</u> source... Ctrl+R
R <u>e</u> source C <u>o</u> py...
 N <u>e</u> w <u>A</u> TL O <u>b</u> ject...

PROJEKT – PROJECT

Aktives Projekt festlegen	▶
Dem Projekt hinzufügen	▶
Q <u>u</u> ellcodeverwaltung	▶
Abhängigkeiten...	
Einstellungen...	Alt+F7
M <u>a</u> kefile exportieren...	
Projekt in den Arbeitsbereich einfügen...	

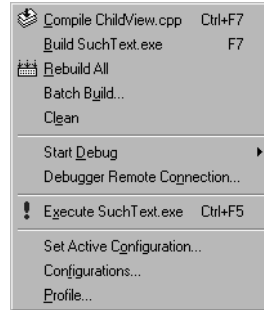
S <u>e</u> t Active Project	▶
A <u>dd</u> To Project	▶
D <u>e</u> pendencies...	
S <u>e</u> tings...	Alt+F7
E <u>x</u> port M <u>a</u> kefile...	
I <u>n</u> sert Project into Workspace...	

DEM PROJEKT HINZUFÜGEN – ADD TO PROJECT

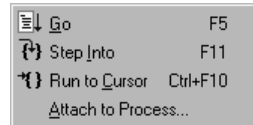
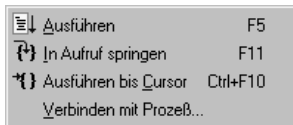
N <u>e</u> u...
 N <u>e</u> uer <u>O</u> rdner...
▶  F <u>e</u> ilen:
▶  D <u>a</u> tenverb <u>in</u> dung...
 K <u>o</u> mponenten und St <u>e</u> uerelemente...

N <u>e</u> u...
 N <u>e</u> w F <u>o</u> lder...
▶  F <u>e</u> iles...
▶  D <u>a</u> ta C <u>o</u> nnection...
 C <u>o</u> mponents and C <u>o</u> ntr <u>o</u> ls...

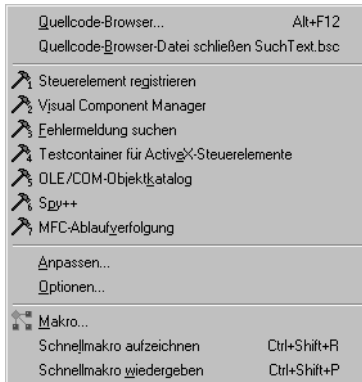
ERSTELLEN – BUILD



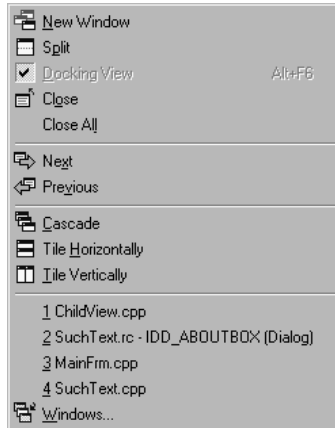
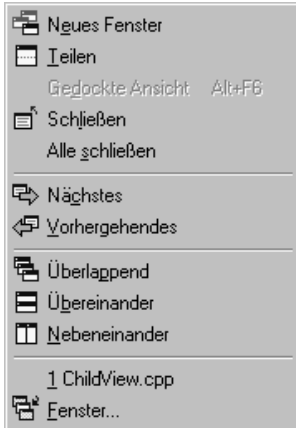
DEBUG STARTEN – START DEBUG



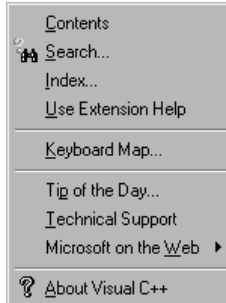
EXTRAS – TOOLS



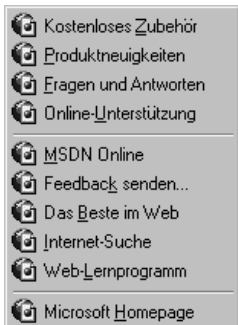
FENSTER – WINDOW



HILFE – HELP



MICROSOFT IM WEB – MICROSOFT ON THE WEB



DEBUG – DEBUG

	Ausführen	F5
	Erneut starten	Ctrl+Shift+F5
	Debug beenden	Shift+F5
	Anhalten	
	Code-Änderungen zuweisen	Alt+F10
	In Aufruf springen	F11
	Aufruf als ein Schritt	F10
	Ausführen bis Rücksprung	Shift+F11
	Ausführen bis Cursor	Ctrl+F10
	Bestimmte Funktion anspringen	
	Ausnahmen...	
	Threads...	
	Module...	
	Nächste Anweisung anzeigen	Alt+Num *
	Schnellüberwachung...	Shift+F9

	Go	F5
	Restart	Ctrl+Shift+F5
	Stop Debugging	Shift+F5
	Break	
	Step Into	F11
	Step Over	F10
	Step Out	Shift+F11
	Run to Cursor	Ctrl+F10
	Step Into Specific Function	
	Exceptions...	
	Threads...	
	Show Next Statement	Alt+Num *
	QuickWatch...	Shift+F9

LAYOUT – LAYOUT

	Ausrichten	▶
	Gleichmäßig verteilen	▶
	Gleiche Größe	▶
	Schaltflächen anordnen	▶
	Im Dialogfeld zentrieren	▶
	Größe an Inhalt anpassen	Shift+F7
	Auto-Größe	▶
	Kippen	
	Tabulator-Reihenfolge	Ctrl+D
	Einstellungen für Führungslinien...	
	Testen	Ctrl+T

	Align	▶
	Space Evenly	▶
	Make Same Size	▶
	Arrange Buttons	▶
	Center in Dialog	▶
	Size to Content	Shift+F7
	Auto Size	▶
	Flip	
	Tab Order	Ctrl+D
	Guide Settings...	
	Test	Ctrl+T

BILD – IMAGE

	Farben invertieren	
	Horizontal kippen	X
	Vertikal kippen	Y
	Drehen 90 Grad	Z
	Farben einstellen...	
	Palette laden...	
	Palette speichern...	
	Deckend zeichnen	O
	Symboleisten-Editor...	
	Rastereinstellungen...	
	Neues Geräteabbild...	Ins
	Geräteabbild öffnen...	
	Geräteabbild löschen	

	Invert Colors	
	Flip Horizontal	X
	Flip Vertical	Y
	Rotate 90 degrees	Z
	Adjust Colors...	
	Load Palette...	
	Save Palette...	
	Draw Opaque	O
	Toolbar Editor	
	Grid Settings...	
	New Device Image...	Ins
	Open Device Image...	
	Delete Device Image	



Anhang

D

Dieser Abschnitt zeigt Ihnen alle in Visual C++ vorhandenen Symbolleisten und soll Ihnen Helfen, schnell die für Ihre Arbeit richtige Leiste zu finden.

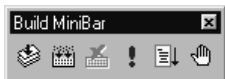
Standard



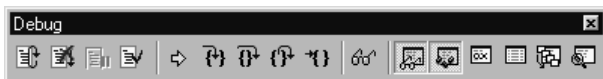
Erstellen



Erstellen mini



Debug



Ressource



Assistentenleiste



Bearbeiten



Browse



ATL





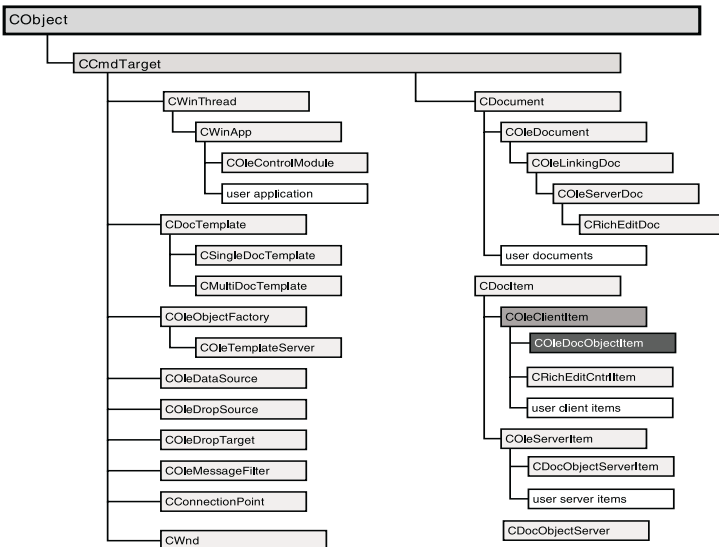
Anhang

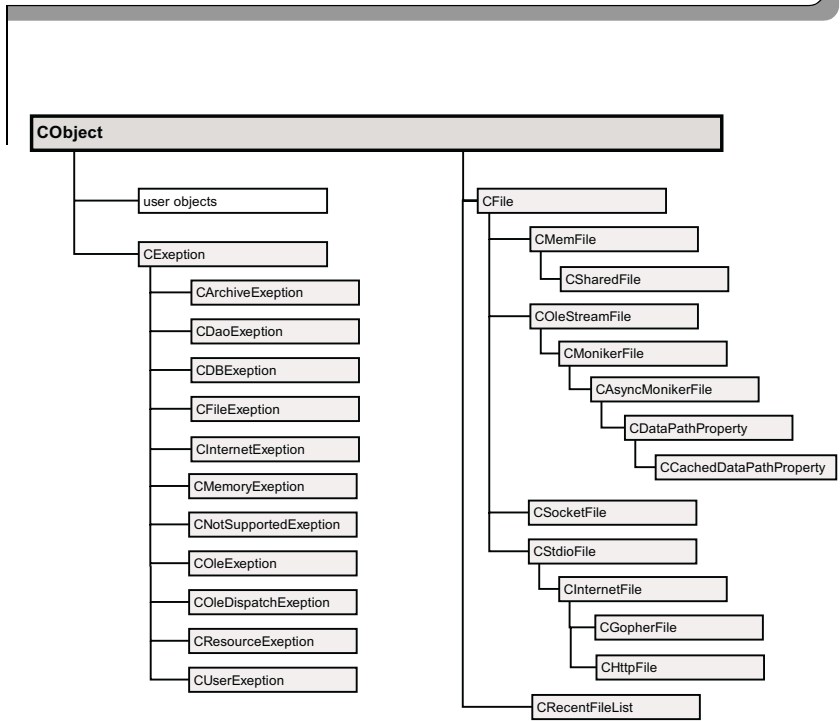
E

Auf den folgenden Seiten finden Sie eine grafische Darstellung der MFC-Klassenbibliothek der Version 6.0. Alle geänderten Klassen gegenüber der Vorgängerversion der MFC werden in einem dunkleren Grauton dargestellt. Neu hinzugekommene Klassen werden durch weiße Schrift auf dunklem Hintergrund gekennzeichnet.

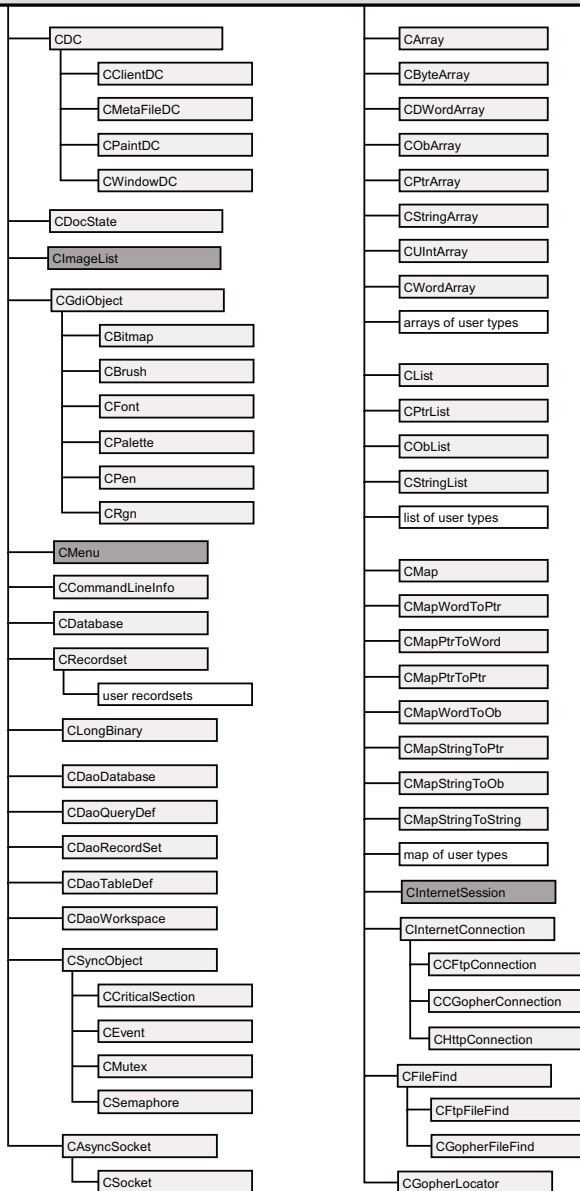
Neue Klasse

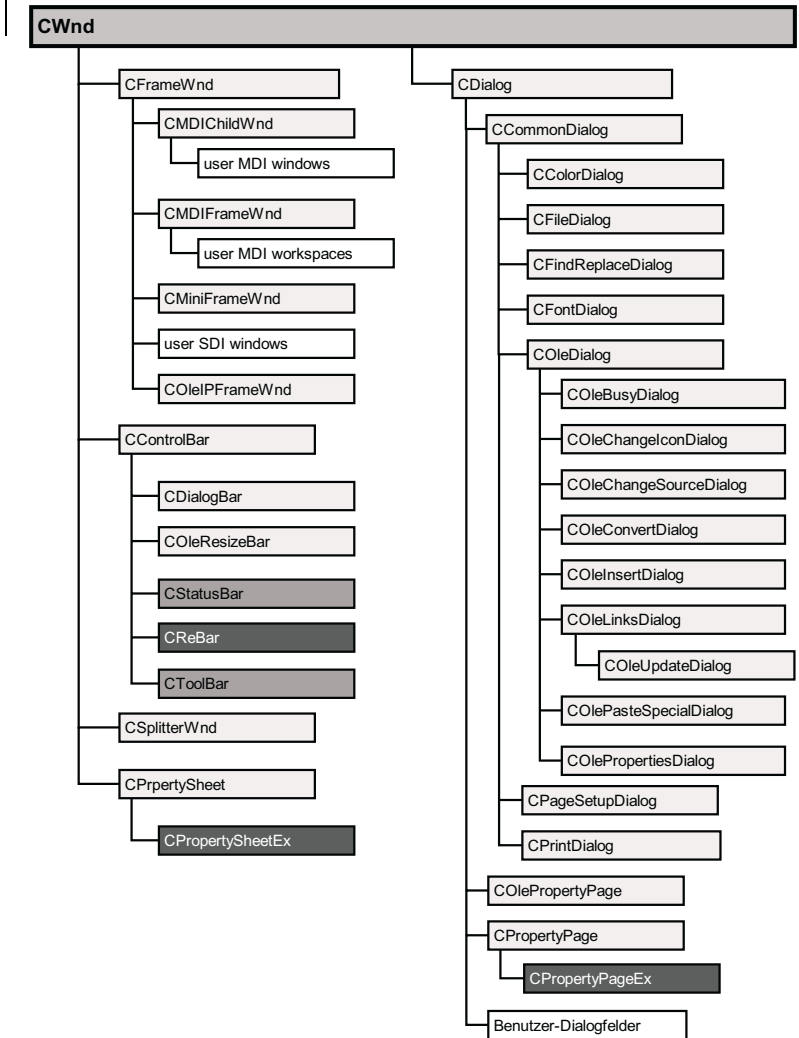
Aktualisierte Klasse





CObject





CWnd

CView

CCtrlView

CEditView

CListView

CRichEditView

CTreeView

CScrollView

user scroll views

CFormView

user form views

CDaoRecordView

CHtmlView

COleDBRecordView

CRecordView

user record views

CAnimateControl

CButton

CBitmapButton

CComboBox

CComboBoxEx

CDateTimeCtrl

CEdit

CHeaderCtrl

CHotKeyCtrl

CIPAddressCtrl

CListBox

CCheckListBox

CDragListBox

CListCtrl

CMonthCalCtrl

COleCtrl

CProgressCtrl

CReBarCtrl

CRichEditCtrl

CScrollBar

CSliderCtrl

CSpinButtonCtrl

CStatic

CStatusBarCtrl

CTabCtrl

CToolbarCtrl

CTooltipCtrl

CTreeCtrl

Klassen, die nicht von CObject abgeleitet sind

CArchive
CDumpContext
CRuntimeClass
CPoint
CRect
CSize
CString
CTime
CTimeSpan

CCreateContext
CMemoryState
COleSafeArray
CPrintInfo

CMultiLock
CSingleLock

CHtmlStream
CHttpFilter
CHttpFilterContext
CHttpServer
CHttpServerContext

CcmdUI
COleCmdUI
CDaoFieldExchange
CDataExchange
CDBVariant
CFieldExchange
COleDataObject
COleDispatchDriver
CPropExchange
CRectTracker
CWaitCursor

CTypedPtrArray
CTypedPtrList
CTypedPtrMap

CFontHolder
CPictureHolder

COleCurrency
COleDateTime
COleDateTimeSpa
COleVariant



Stichwortverzeichnis

A

Accelerator-Table 171
ActiveX 121
AddDocTemplate 517
AddString 317
AFX_IDI_STD_FRAME 82, 100
AFX_IDI_STD_MDIFRAME 100
Afx_msg 96, 97
AfxDump 252
AfxGetApp 76, 80, 556
AfxGetAppName 80
AfxGetMainWnd 80, 198
AfxMessageBox 579
AfxRegisterWndClass 99
AFXRES.H 100
AFXWIN.H 78, 98
AfxWinTerm 106
AfxWndProc 96
AktivX Steuerelemente 468
ALT+F4 181
Andere Nachrichten 95
ANSI_FIXED_FONT 214
ANSI_VAR_FONT 214
Anwendungs-Assistent 20, 72, 118,
459, 468, 505, 513, 522, 523, 551,
558, 564, 586, 616
Anwendungsobjekt 73
AppendMenu 163
AppWizard 20, 118

Arbeitsbereich 40, 119
Arial-Schriftart 213
Arrays 406
Assembler-Code 261
ASSERT-Makro 263
Assistenten 118
Assistentenleiste 135
Aufrufhierarchie 242
Aufrufliste 261
Ausnahmebehandlung 428, 619
Ausnahmeklasse 428

B

Basic Styles 274
Basisklasse 566
BEGIN_MESSAGE_MAP 103
BeginPaint 101, 202
BeginWaitCursor 100
Beispielprogramme 32
Beschleuniger-Ressource 172
Beschleunigertasten 171
Bildlaufleisten 230, 554
Bitmap 489
BN_CLICKED 279
Breakpoint 253
Browser 63, 243
Abgeleitete Klassen und Elemente
245
Basisklassen und Elemente 246
-Datenbank 243

- Definitionen und Referenzen 248
- Diagramm Aufrufe 246
- Diagramm Aufrufende Funktionen 247
- Symbolleiste 249
- BS_3STATE 278
- BS_AUTO3STATE 278
- BS_AUTOCHECKBOX 278
- BS_AUTORADIOBUTTON 279
- BS_LEFTTEXT 278, 279
- BS_OWNERDRAW 277
- BSC-Datei 243
- Buchinhalt 20
- Buttons 270
- C**
- CArchive 406, 423, 500, 501, 568
 - Read 501
 - Write 501
- catch 429, 619
- CB_ERR 317
- CB_ERRSPACE 317
- CBitmap 215
- CBN_CLOSEUP 312, 357
- CBN_DBLCLK 104, 312
- CBN_DROPDOWN 312
- CBN_EDITCHANGE 313
- CBN_EDITUPDATE 313
- CBN_ERRSPACE 313
- CBN_KILLFOCUS 313
- CBN_SELCHANGE 313
- CBN_SETFOCUS 313
- CBrush 215, 545
 - CreateSolidBrush 545
- CBS_DROPDOWN 311
- CBS_DROPDOWNLIST 311
- CBS_NOINTEGRALHEIGHT 311
- CBS_SIMPLE 311
- CClientDC 227
- CCommandTarget 96, 100, 497, 517, 564, 618
- CCommandUI 548
- CComboBox
 - SetItemData 481
- CCommandLineInfo 87, 406
- CDaoErrorInfo 620
- CDaoException 620
 - m_pErrorInfo 620
- CDaoRecordset 618
- CDatabase 607, 608, 610
 - BeginTrans 609
 - Close 609
 - CommitTrans 610
 - ExecuteSQL 609
 - Open 608
 - OpenEx 609
 - Rollback 610
- CDBException
 - m_strError 620
- CDBExceptions 620
- CDC 101, 202, 510, 514, 515
 - FillRect 545
 - IsPrinting 514
 - LineTo 544
 - MoveTo 544
 - SelectObject 543
 - SetBkColor 515
 - SetBkMode 515
- CDialog 440, 541, 550
 - Create 487
 - CreateIndirect 441
 - DoModal 453
 - EndDialog 454
 - InitDialog 541
 - InitModalIndirect 441
 - OnCancel 454, 457
 - OnInitDialog 454, 455, 456, 482
 - OnOK 454, 457, 541
- CDocManager 517
- CDocTemplate 517
 - CreateNewDocument 518
 - CreateNewFrame 518, 556, 582
 - InitialUpdateFrame 518, 556, 582
 - OpenDocumentFile 581
- CDocument 497, 529, 533, 607
 - AddView 499
 - DeleteContents 503, 529, 534
 - GetFirstViewPosition 499, 582
 - GetNextView 499, 582
 - IsModified 502
 - m_bAutoDelete 503
 - OnCloseDocument 501, 534
 - OnFileSendMail 504
 - OnNewDocument 501, 534, 575
 - OnOpenDocument 501, 534
 - OnSaveDocument 501

- OnUpdateFileSendMail 504
- RemoveView 499
- SaveModified 502
- SetModifiedFlag 502, 530, 575, 580
- UpdateAllViews 498, 509, 530, 531, 536, 580, 633
- CEdit 295
- CFile 406, 422
 - Abort 426
 - Close 426
 - m_cause 429
 - Open 424
 - Read 426
 - ReadHuge 426
 - Seek 427
 - Write 427
 - WriteHuge 427
- CFileDialog 188, 434
- CFileException 428
- CFindReplaceDialog 191
- CFont 215
- CFormView 549, 558, 564, 613
- CFrameWnd 73, 97, 174, 551, 582
 - ActivateFrame 582
 - Create 98
 - GetActiveDocument 582
- CGdiObject 215, 219
- CheckMenuItem 159
- CHOOSECOLOR 185
- CHOOSEFONT 185
- CImageList 333, 406, 412, 628, 634
 - Add 414
 - Create 634
 - ExtractIcon 414
 - Remove 414
 - Replace 414
- ClassView 37, 48, 51
- ClassWizard 20
- Clipping-Region 205
- CListBox 430, 572
 - AddString 432
 - GetText 432
 - ResetContent 433
 - SetTabStops 435
- CMainFrame
 - OnCreateClient 625
- CMapWordToPtr 635
- CMDIChildWnd 551, 555, 558
- CMDIFrameWnd 558, 582
 - MDIGetActiv 582
- CMemFile 423
- CMenu 142, 154, 161
- CMultiDocTemplate 517, 555
- CObArray 410
 - Add 410
 - GetAt 411
 - InsertAt 410
 - RemoveAll 411
 - RemoveAt 411
- CObject 406, 497, 566, 567
 - IsKindOf 432
- COleStreamFile 423
- COMBOBOX 104
 - Ressource 311
- Command Messages 95
- COMMDLG.DLL 184
- common dialogs 183
- Component Gallery 47
- Console Application 43
- Copy 298
- Courier New 213
- CPaintDC 101, 202
- CPalette 215
- CPen 215, 542
 - CreatePen 542
- Create 97, 148
 - CFrameWnd 98
- CreateFont 217
- CreateOrActivateFrame 583
- CreatePopupMenu 165
- CRecordSet
 - Open 642
 - Requery 637
 - Update 633
- CRecordset 610, 613, 618
 - AddNew 612
 - Edit 612
 - GetDefaultConnect 611
 - GetDefaultSQL 611
 - GetSQL 612

- m_strFilter 612
 - m_strSort 612
 - Update 612
- CRecordView 613, 618, 626, 630
 - OnGetRecordset 614
 - OnInitialUpdate 614
 - OnMove 613
- CRect 102, 204
- CRegion 215
- CRuntimeClass 406, 519
- CScrollBar 235, 325
- CScrollView 549, 572
 - ResizeParentToFit 572
- CSingleDocTemplate 517
- CSize 222, 554
- CSliderCtrl 329
- CSocketFile 423
- CSpinButtonCtrl 339
- CSplitterWnd 520, 551
 - CreateView 554
 - OnCreateClient 551
- CStatic 303
- CStdioFile 188, 423, 430
 - ReadString 433
 - WriteString 433
- CString 196, 226, 406, 500, 567, 569
 - Find 410
 - GetAt 408
 - GetLength 408
 - Left 409
 - MakeLower 409
 - MakeUpper 409
 - Mid 409
 - Operator + 408
 - Operator += 408
 - Operator = 407, 408
 - ReverseFind 410
 - Right 409
 - SetAt 408
 - TrimLeft 409
 - TrimRight 409
 - Vergleichsoperatoren 408
- CStringArray 410
- CTEXT 304
- CTime 621
- CToolBar 406
- CTreeCtrl 624, 634
 - Expand 638
 - InsertItem 636
 - SetImageList 634
- CTreeView 624
 - GetTreeCtrl 634
 - OnSelchanged 636
- CUIIntArray 410
- Cursor 100
 - verändern 84
- Custom AppWizard 43
- Cut 298
- CView 497, 507, 514, 516, 549, 554
 - GetDocument 508, 510, 577, 633
 - OnBeginPrinting 515
 - OnDraw 510, 513, 546
 - OnEndPrinting 515
 - OnInitialUpdate 509, 551, 572
 - OnPreparePrinting 514
 - OnPrint 515
 - OnUpdate 499, 509, 530, 551, 577
- CWinApp 73, 75, 83, 85, 517, 589, 607
 - AddDocTemplate 505, 580
 - InitInstance 463, 505, 519, 555, 580, 581
 - OnFileNew 518, 520, 556, 575, 581
 - OnFileOpen 581
 - OpenDocumentFile 518
 - ProcessShellCommand 518, 556, 581
 - WinHelp 591
- CWindowDC 228
- CWinThread 92
- CWnd 75, 95
 - ClientToScreen 432
 - DestroyWindow 487, 488
 - DoDataExchange 432, 473
 - EnableWindow 473
 - GetCapture 431
 - GetDlgItem 472
 - GetParentFrame 582
 - GotoDlgCtrl 549
 - Invalidate 499
 - OnClose 493
 - OnCtlColor 458, 480
 - OnLButtonDown 431

- OnLButtonUp 431
- OnPaint 510
- OnSysCommand 464
- PostNcDestroy 491, 493
- SetFocus 473
- ShowWindow 485
- SubclassDlgItem 433
- UpdateData 577, 579
- WindowFromPoint 432
- CWordArray 410
- D**
- DAO 605
- DAO-Recordsets 605
- Dateien-Fenster 468, 523
- Datenbankkomponenten 606
- Datenbank-Unterstützung 120
- Datensatzgruppe 610, 618
- Debugger 35, 38, 39, 252, 255, 261
- Debug-Version 42, 61, 78, 128, 257, 263
- DECLARE_MESSAGE_MAP 97, 103
- DECLARE_SERIAL 568
- DEFAULT_PITCH 218
- DEFPUSHBUTTON 277
- DefWindowProc 75
- Delete 488, 574
- DeleteItem 336
- DeleteMenu 163
- DeleteObject 219, 543
- DeleteString 318
- DestroyIcon 83
- DestroyWindow 105, 180
- Developer Studio 20, 23, 34, 39, 43, 64, 81, 172
 - Aufbau 35
- Development Library-CD 35
- DEVICE_DEFAULT_FONT 214
- Device-Kontext 101, 202
 - außerhalb von OnPaint 227
- Dialog Data Exchange 284
- dialogbasierende Anwendung 462
- Dialogbox 440
 - einheiten 300
- Dialogeditor 549, 572
- Dialogfeld Eigenschaften 446, 450

- Dialogfeldbasierend 468
- Dialogfeldklasse 455, 541
- Dialogfeldrahmen 448
- Dialogfeldsymbolleiste 445
- Dialogfeldvorlage 550
- Dialogvorlage 441, 446, 452, 470, 483, 485, 507, 526, 561, 564, 613
- Dithering 209
- Document-View-Architektur 468
- DoDataExchange 541, 631
- Dokument-Ansicht 496, 514, 517
- Dokument-Ansicht-Konzept 526
- Dokumentvorlage 517, 556, 580
- Drag & Drop 430
- DrawIcon 83
- DrawMenuBar 164
- DrawText 226
- Druckauftrag 514
- Druckausgabe 511
- Druckvorschau 513
- DSN 615, 622
- DSN-Datei 612, 616
- DTN_FORMATQUERY 369
- dynamisch geteilte Fenster 551

- E**
- Editfeld 288
- EDITTEXT 290
- Eigenschaften 446
- Eigenschaften-Fenster 146, 485, 486
- Eingabefokus 293, 494
- Einsteiger Edition 26
- Einzelschrittmodus 253
- EN_CHANGE 292, 293
- EN_ERRSPACE 293
- EN_HSCROLL 292
- EN_KILLFOCUS 292
- EN_MAXTEXT 293
- EN_SETFOCUS 292
- EN_UPDATE 292, 293
- EN_VSCROLL 293
- EnableMenuItem 156
- END_MESSAGE_MAP 105
- Endebehandlung 502, 534

EndPaint 101, 202
 EndWaitCursor 100
 Enterprise Edition 26
 Ereignisse auslösen 270
 ES_AUTOHSCROLL 291
 ES_AUTOVSCROLL 291
 ES_CENTER 291
 ES_LOWERCASE 290
 ES_MULTILINE 290
 ES_PASSWORD 290
 ES_READONLY 290
 ES_RIGHT 291
 ES_UPPERCASE 290
 ETO_CLIPPED 223
 ETO_OPAQUE 223
 ExcludeClipRect 207
 ExitInstance 79, 106

F

Fehlersuche 242
 Fenster
 Arbeitsbereich 36, 47
 Ausgabe 36
 Editor 36
 Funktion 95
 neu zeichnen 200
 Objekt 95
 FF_DECORATIVE 218
 FF_DONTCARE 218
 FF_MODERN 218
 FF_ROMAN 218
 FF_SCRIPT 218
 FF_SWISS 218
 FileView 37, 48
 FINDMSGSTRING 193
 FindNext 192
 FINDREPLACE 185, 191
 FindResource 80
 FIXED_PITCH 218
 FlashWindow 483
 Fokus 479
 Font 482
 Format-Eigenschaften 447, 448
 Formularvorlage 549
 Frame-Document-Ansicht-Architektur 22

Fremdobjekte 618
 Fremdvariablen 619
 Führungslinien 445
 Funktionstasten 39, 40

G

Garbage Collection
 der MFC 155
 temporärer Objekte 284
 GDI
 Funktionen 202
 Objekte 208
 Objekte im Device-Kontext 215
 GetBuddy 339
 GetButtonStyle 287
 GetCheck 287
 GetClientRect 102, 203
 GetClipboard 208
 GetCount 321
 GetCurrentMessage 153, 280
 GetCurSel 320
 GetDC 227, 228
 GetDlgItem 283
 GetDlgItemText 285
 GetItemData 319, 320, 336
 GetItemText 336
 GetLine 297
 GetMenu 154
 GetMenuState 160
 GetRange 329, 339
 GetSelectedItem 336
 GetSellItems 321
 GetSubMenu 154
 GetSysColor 211
 GetSystemMenu 169
 GetTextColor 210
 GetTextExtent 222
 GetTextMetrics 220
 GetUpdateRect 205
 GetWindowDC 228
 GetWindowRect 166, 203
 GetWindowText 285, 296

H

Haltepunkt 253
 Hauptfenster 462

Hauptmenü,
 Zugriff auf 154
 Header-Datei schützen 76
 Help Viewer 596
 Helv-Schriftarten 213
 Hervorhebung eines Buttons 287
 Hilfe-Compiler 588
 Hilfe-Workshop 587
 hPrevInstance 78
 HTML Help 594

I
 ICON
 -Ressource 305
 Icon 486
 in Dialogboxen 305
 IDC_ARROW 100
 IDCANCEL 178
 IDNO 178
 IDOK 178
 IDR_MAINFRAME 149
 IDYES 178
 IMPLEMENT_SERIAL 568
 Info über-Dialog 178
 InfoView 41, 48
 INI-Datei 129
 InitApplication 76
 InitInstance 75, 76, 78, 85, 110, 128,
 131
 InsertItem 336
 Installation 26
 Methode 28
 Standard 25
 Instanzen eines Programms 76
 Internet-Server-Erweiterungen 23
 InvalidateRect 112, 205, 301

J
 Jet-Datenbankmodul 605

K
 Klassen-Assistent 20, 39, 127, 132,
 150, 459, 471, 478, 490, 516, 536,
 564, 566, 570, 618
 Klassen-Fenster 564

Klassenhierarchie 245
 Kombinationsfeld 479
 Kommandozeile 85
 Kompilieren 60
 Komponenten und Steuerelemente 118
 Kontextabhängige Hilfe 66, 450, 586,
 597
 Kontext-Menü 165

L
 Laufleisten 323
 LB_ERR 317
 LB_ERRSPACE 317
 LBN_DBLCLK 313
 LBN_ERRSPACE 313
 LBN_KILLFOCUS 313
 LBN_SELCANCEL 312
 LBN_SELCHANGE 313
 LBN_SETFOCUS 313
 LBS_EXTENDEDSEL 310, 356
 LBS_MULTIPLESEL 310, 356
 LBS_NOINTEGRALHEIGHT 310, 356
 LBS_NOTIFY 310, 355
 LBS_SORT 309, 311, 355
 LBS_USETABSTOPS 310
 LimitText 298
 LineFromChar 303
 Linken 60
 LISTBOX 307
 Ressource 309, 355, 367, 369, 370,
 379, 395
 LoadAccelTable 173
 LoadCursor 84, 100
 LoadIcon 83, 306
 LoadOEMCursor 84
 LoadOEMIcon 83
 LoadStandardCursor 84
 LoadStandardIcon 83
 LoadStdProfileSettings 129
 Löschen temporärer Objekte 155
 logische Schriftart 214, 217
 LPCSTR 406
 LTEXT 304

M

- m_hAccelTable 174
- m_hWnd 189
- m_lpCmdLine 86
- m_pszAppName 85
- Mail 498
- Mailsystem 504
- MAKEINTRESOURCE 149
- Makros 57
- MAPI 503
- Mapping-Modi 213
- Maustaste 522
- Maximieren 486
- MB_CANCEL 178
- MB_ICONEXCLAMATION 179
- MB_ICONSTOP 179
- MB_OK 178
- MB_YESNO 178
- MDI 519, 533, 550, 620
- MDI-Anwendung 555, 623
- Member-Funktion 516
- Member-Variable 539, 567
- Memory Leaks 264
- Memory-Device-Kontext 215
- Menü 142
 - dynamisch 161
 - Popup 165
 - Systemmenü 169
- Menü-Editor 143, 527, 563
- Menüpunkt
 - markieren 159
 - Zustand abfragen 160
- MessageBeep 483
- MessageBox 178, 181, 483
- Message-Map 73, 74, 93, 95, 103, 132, 150, 151, 165, 170, 174, 209, 216, 234, 270, 279, 313
- MF_BYCOMMAND 157
- MF_BYPOSITION 157
- MF_CHECKED 159
- MF_DISABLED 157
- MF_ENABLED 157
- MF_GRAYED 157
- MF_POPUP 164
- MF_SEPARATOR 164

- MF_STRING 164
- MF_UNCHECKED 159
- Microsoft Foundation Classes 15, 20, 72, 73, 75, 95, 173, 242
- Minimieren 486
- modal 441, 452, 484
- MSG-Struktur 153
- Multiple Document Interface 97
- Multitasking 88
 - non-preemptiv 88
 - OnIdle 89
 - preemptiv 88
 - Threads 92

N

- Name einer Schriftart 213
- Namenskonvention 70
- Neue Klasse 460, 564
- new 488, 574
- nicht-modal 441, 485
- Notification Messages 95
- nSBCode 325

O

- OCX 121
- ODBC 604
 - Datenquelle 614
 - Datenquellen-Administrator 615, 616
 - Driver Manager 614
 - Komponenten 606
 - Manager 612
 - Server 604
 - Treiber 606, 614, 621
- OEM_FIXED_FONT 214
- Old English-Schrift 213
- OLE 121
- ON_COMMAND 150
- ON_UPDATE_COMMAND_UI 548
- ON_WM_HSCROLL 234
- ON_WM_RBUTTONDOWN 165
- ON_WM_SYSCOMMAND 170
- ON_WM_VSCROLL 234
- OnClose 105, 180
- OnCmdMsg 96
- OnFileOpen 607

OnFind 194
 OnGetRecordset 626
 OnHScroll 234, 324
 OnIdle 89, 155
 OnLButtonDown 85, 112
 Online-Hilfe 64
 OnNcDestroy 106
 OnPaint 100, 111, 201
 OnSetCursor 113
 OnSysCommand 170
 OnVScroll 234, 324
 OPAQUE 211
 Open 607
 OpenEx 607
 OPENFILENAME 185, 187
 Optionen
 Compiler und Linker 62
 Projekt 62
 OutputDebugString 252
 Owner-Draw-Button 277

P

PAINTSTRUCT 101, 202
 Paradigma
 dokumentorientiertes 21
 Paste 298
 PeekMessage 538
 PostQuitMessage 106
 Precompiled Header Files 78
 PreCreateWindow 232
 PreTranslateMessage 174
 PRINTDLG 185
 Professional Edition 26
 Programmcursor 100
 Programmgerüst 72, 118
 Programm-Icon 81
 Progress Control 341
 Projekt 40, 119, 128
 bearbeiten 46
 erstellen 44
 Projekttyp 42
 Projektverzeichnis 40, 41, 119
 Property Pages 342
 Property Sheets 66, 342
 Proportionalschriften 213

Prozesse 92
 public 567
 Punkt
 Größenangabe für Schriften 213

Q

Quelltexte der MFC 67
 Quick-Watch-Fenster 260

R

Radio-Button 272
 rectDefault 99
 RECT-Struktur 203
 Regelbereich von Bildlaufleisten 233
 Register-Fenster 261
 Registerkarte 479, 482
 Registrieren von Fensterklassen 77
 Registrierung 505
 Registrierungsdatei 130
 ReleaseDC 227
 Release-Version 38, 42, 61, 78, 128, 263
 ReplaceSel 297
 ResetContent 318
 RESOURCE.H 148
 ResourceView 37, 48, 83, 144, 561
 Ressource 47, 142, 146
 Datei 81
 Fenster 442, 524, 526
 Symbolleiste 81
 Vorlagen 50
 RGB
 -Farbwert 208
 -Makro 208
 Rich Text Format 586
 Rollbalken 230
 RTEXT 304, 305
 RTF 586
 Run 76

S

SB_BOTH 231
 SB_BOTTOM 235
 SB_ENDSCROLL 235
 SB_HORZ 231
 SB_LINEDOWN 235, 325

- SB_LINEUP 235, 325
 - SB_PAGEDOWN 235, 325
 - SB_PAGEUP 235, 325
 - SB_THUMBPOSITION 235, 325
 - SB_THUMBTRACK 235, 325
 - SB_TOP 235
 - SB_VERT 231
 - SBR-Datei 243
 - Schaltflächen 522
 - Schieber 230
 - Schieberegler 230, 328
 - Schliessen eines Fensters 105
 - Schriftfamilie 213
 - Script-Schriftart 213
 - Scrollbar 230
 - SDI 81, 519, 533, 550, 620
 - SDI-Anwendung 503, 519, 523
 - Seitenansicht 511
 - Seitenzahlen 512
 - SELECT 641
 - SelectClipRgn 207
 - SelectObject 217
 - SelectStockObject 214
 - Serialisation 422, 500, 558, 567, 606
 - Serialize 500, 576
 - serifenlose Schriften 213
 - SetBkColor 210
 - SetBkMode 211
 - SetBuddy 339
 - SetButtonStyle 287
 - SetCheck 286
 - SetCurSel 320
 - SetCursor 84, 100, 431
 - SetDefaultTitle 197
 - SetDlgItemText 285
 - SetIcon 305
 - SetItemData 319, 336
 - SetItemText 336
 - SetModify 302
 - SetRange 329, 339
 - SetScrollPos 233, 327
 - SetScrollRange 233, 327
 - SetSel 299, 321
 - SetState 287
 - SetTabStops 300
 - SetTextAlign 102, 222
 - SetTextColor 102, 208
 - SetTic 330
 - SetTicFreq 330
 - SetWindowText 197, 285, 296, 322
 - ShowScrollBar 230
 - Sichtbar 485
 - Simonyi, Charles 71
 - Sleep 547
 - Slider-Control 328
 - Speicher-Fenster 261
 - SS_NOPREFIX 305
 - Standard-Dialog 512
 - static 491
 - statisch geteilte Fenster 551
 - statische Variable 490
 - Statuszeile 123
 - STDAFX.H 121
 - Steuerungen 443, 472
 - Suchen 52
 - in Dateien 53
 - inkrementell 53
 - Symboleiste 37, 123, 522, 524, 562
 - Debugger 257
 - Symboleisten-Editor 524
 - Syntax-Coloring 35, 51
 - SYSTEM_FIXED_FONT 214
 - SYSTEM_FONT 214
 - Systemmenü 466, 494
- T**
- TA_BASELINE 223
 - TA_BOTTOM 223
 - TA_CENTER 223
 - TA_LEFT 223
 - TA_RIGHT 223
 - TA_TOP 223
 - TabbedTextOut 226
 - Tabulator-Reihenfolge 450
 - Taschenrechner 463
 - Teilfarben eines RGB-Farbwertes 212
 - Testmodus 452

Text ausgeben
 mit Statics 304
 TEXTMETRIC 221
 TextOut 102, 225
 Threads 80, 89, 92
 user-interface 93
 worker 93
 Timer 491
 Times New Roman-Schriftart 213
 Titelleiste 197
 tmAscent 221
 tmAveCharWidth 221
 tmDescent 221
 tmExternalLeading 221
 tmHeight 221
 tmMaxCharWidth 221
 Tool Tips 67
 TPM_CENTERALIGN 166
 TPM_LEFTALIGN 166
 TPM_LEFTBUTTON 166
 TPM_RIGHTALIGN 166
 TPM_RIGHTBUTTON 166
 TRACE-Makro 112, 263
 TrackPopupMenu 165
 TRANSPARENT 211
 Tree Control 331
 try 429, 619
 Turtle-Grafik 543
 TV_INSERTSTRUCT 332, 635
 TV_ITEM-Struktur 333

U

Ungarische Namenskonvention 70
 Untermenü,
 Zugriff auf 154
 Update-Region 205
 UpdateWindow 205

V

VARIABLE_PITCH 218
 Variablen-Fenster 259
 VBScript 57
 Vererbungshierarchie 242

VERIFY-Makro 264
 Visual C++
 Environment 41
 NMAKE 42
 Visual Sourcesafe 32
 Visual Studio 34
 Voraussetzungen für Visual C++ 24

W

Watch-Fenster 259
 Windows-Drucker 512
 Windows-Fenster 95
 Window-Styles 232
 WinHelp 586, 589
 WinMain 73
 WM_CLOSE 105
 WM_COMMAND 104
 WM_CREATE 99
 WM_DESTROY 106
 WM_ERASEBKGND-Nachricht 206
 WM_HSCROLL 324, 329
 WM_NCDESTROY 106
 WM_PAINT 99, 100, 104, 200
 WM_QUERYENDSESSION 182
 WM_QUIT 106
 WM_SETTEXT 197
 WM_VSCROLL 324
 WS_BORDER 290, 309, 355, 367, 370,
 379
 WS_DISABLED 274, 275, 276
 WS_GROUP 274, 275, 276
 WS_HSCROLL 230, 290
 WS_OVERLAPPEDWINDOW 99
 WS_TABSTOP 274, 275
 WS_VISIBLE 274, 275
 WS_VSCROLL 230, 290, 310, 311, 355

Z

Zugriffstasten 519
 Zustände speichern mit Buttons 270
 Zwischenablage
 Schnittstelle zu Editfeldern 295



Copyright

Daten, Texte, Design und Grafiken dieses eBooks, sowie die eventuell angebotenen eBook-Zusatzdaten sind urheberrechtlich geschützt.

Dieses eBook stellen wir lediglich als **Einzelplatz-Lizenz** zur Verfügung!

Jede andere Verwendung dieses eBooks oder zugehöriger Materialien und Informationen, einschliesslich der Reproduktion, der Weitergabe, des Weitervertriebs, der Platzierung im Internet, in Intranets, in Extranets anderen Websites, der Veränderung, des Weiterverkaufs und der Veröffentlichung bedarf der schriftlichen Genehmigung des Verlags.

Bei Fragen zu diesem Thema wenden Sie sich bitte an:

<mailto:info@pearson.de>

Zusatzdaten

Möglicherweise liegt dem gedruckten Buch eine CD-ROM mit Zusatzdaten bei. Die Zurverfügungstellung dieser Daten auf der Website ist eine freiwillige Leistung des Verlags. Der Rechtsweg ist ausgeschlossen.

Hinweis

Dieses und andere eBooks können Sie rund um die Uhr und legal auf unserer Website



(<http://www.informit.de>)

herunterladen