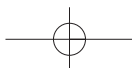
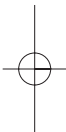
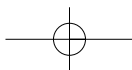
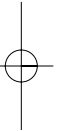
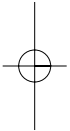
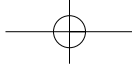


C++ *Wochenend Crashkurs*





C++ *Wochenend Crashkurs*

Stephen R. Davis

*Übersetzung aus dem Amerikanischen
von Dr. Thorsten Graf*



Die Deutsche Bibliothek - CIP-Einheitsaufnahme:

Davis, Stephen R.:

C++ Wochenend Crashkurs

Übersetzung aus dem Amerikanischen von Dr. Thorsten Graf

Bonn : MITP-Verlag, 2001

Einheitssacht.: C++ Wochenend Crashkurs

ISBN 3-8266-0692-2

ISBN 3-8266-0692-2

Alle Rechte, auch die der Übersetzung, vorbehalten. Kein Teil des Werkes darf in irgendeiner Form (Druck, Fotokopie, Mikrofilm oder einem anderen Verfahren) ohne schriftliche Genehmigung des Verlages reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

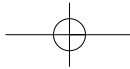
Übersetzung der amerikanischen Originalausgabe:

Stephen R. Davis: C++ Weekend Crash Course

Copyright © by mitp-Verlag,
ein Geschäftsbereich der verlag moderne industrie Buch AG & Co.KG, Landsberg
Original English language edition text and art copyright © 2000 by IDG Books Worldwide, Inc.
All rights reserved including the right of reproduction in whole or in part in any form.
This edition published by arrangement with the original publisher IDG Books Worldwide, Inc..
Foster City, California, USA.

Printed in Germany

Lektorat: Christine Wöltche
Korrektur: Michael Eckloff
Herstellung: Dieter Schulz
Druck: Media-Print, Paderborn
Satz und Layout: Eva Kraskes, Köln

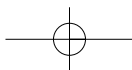
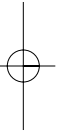
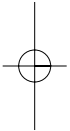
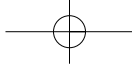


Über den Autor

Stephen R. Davis

Wenn der 43-jährige Vater und Hausmann nicht gerade Fahrrad fährt oder seinen Sohn zu einem Taekwondo-Wettkampf begleitet, arbeitet und lebt Stephen R. Davis in Greenville, Texas als Programmierer mit Leib und Seele.

Beim MITP-Verlag ist neben dem *C++ Wochenend Crashkurs* auch sein Buch *C++ für Dummies* erschienen.



Vorwort

Mit dem C++ *Wochenend Crashkurs* erlernen Sie an einem einzigen – zugegebenermaßen arbeitsreichen – Wochenende die Programmiersprache C++: in 30 Sitzungen à 30 Minuten, also 15 Stunden – von Freitagabend bis Sonntagnachmittag.

Am Ende jeden Teils bekommen Sie Gelegenheit für eine Pause und eine Rekapitulation dessen, was Sie erlernt haben. Viel Glück!

1.1 Was ist C++?

C++ ist heute die populärste Programmiersprache. C++ wird in Anwendungen eingesetzt, die vom Mikroprogramm, das in Ihrer Mikrowelle, in Ihrer Waschmaschine oder in Ihrem Fernseher läuft, bis hin zu Programmen zur Steuerung von Atomraketen oder Marsraketen reichen.

In den späten achtziger Jahren kam C allmählich in die Jahre. Das lag unter anderem daran, dass C keine objektorientierte Programmierung unterstützt. Zu dieser Zeit hat die objektorientierte Welle die Welt im Sturm erobert. Objektorientierten Programmierern wurde das Geld nachgeworfen. Sie brauchten im Gespräch nur »neues Paradigma« zu sagen und hatten sofort eine Menge Bewunderer.

Das Problem war, dass jedes vernünftige Programm in C geschrieben war (es gab einige Programme, die in PASCAL geschrieben waren, wie frühere Versionen von Windows, aber die zählen nicht – wenn Sie die frühen Versionen von Windows kennen, wissen Sie, warum nicht). Die Firmen würden nicht einfach alle existierenden Programme neu schreiben, nur um auf der objektorientierten Welle zu reiten.

Objektorientierte Konzepte wurden in die existierende Programmiersprache C integriert. Das Ergebnis wurde C++ genannt.

C++ ist eine Obermenge von C. Jedes korrekt geschriebene C-Programm kann unter C++ erzeugt werden. Dadurch konnten die Firmen ihre Software stückweise upgraden. Existierender Code konnte weiterhin in C geschrieben sein, wohingegen neuer Code die Features von C++ nutzte.

VIII Vorwort

Zu unserem Glück ist C++ eine standardisierte Sprache. Das American National Standards Institute (ANSI) und die International Standards Organisation (ISO) sind sich einig darüber, was C++ ist. Sie haben eine detaillierte Beschreibung der Programmiersprache C++ erstellt. Diese standardisierte Sprache ist unter dem Namen ANSI oder ISO C++, oder einfach Standard-C++, bekannt.

Standard-C++ unterliegt nicht der Kontrolle einer einzelnen Firma, wie z.B. Microsoft oder Sun. Die Gemeinschaft der Programmierer, die Standard-C++ verwenden, ist nicht abhängig von einem Software Giganten. Außerdem halten sich die Firmen an den Standard, selbst Microsoft's Visual C++ hält sich streng an den C++-Standard.

Die Programme im C++ *Wochenend Crashkurs* können mit jeder Implementierung von Standard-C++ übersetzt werden.

1.2 Das objektorientierte Paradigma

Objektorientierte Programmierung ist nicht nur ein Modetrend. Objektorientierte Programmierung ist eine Methode der Programmierung, die sich sehr von ihren Vorgängern unterscheidet. Objektorientierte Programme können leichter geschrieben und gepflegt werden. Objektorientierte Module können leichter wiederverwendet werden als die Module, die unter einem anderen Paradigma erstellt wurden.

Der C++ *Wochenend Crashkurs* ist mehr als nur eine Einführung in C++. Sie müssen das objektorientierte Paradigma erlernen, um C++ voll nutzen zu können. Der C++ *Wochenend Crashkurs* verwendet Beispiele in C++, um Ihnen die objektorientierte Sicht auf die Welt zu vermitteln. Jeder, der behauptet, in C++ zu programmieren, ohne die objektorientierten Konzepte verstanden zu haben, verwendet C++ als »besseres C«.

1.3 Wer

Der C++ *Wochenend Crashkurs* richtet sich an Anfänger bis hin zu Lesern auf mittlerem Level.

Es werden keine Vorkenntnisse im Bereich Programmierung und Programmierkonzepte beim Leser vorausgesetzt. Die ersten Sitzungen erklären anhand realer Beispiele auf nicht-technische Weise, was Programmierung ist.

Dieses Buch ist auch gut geeignet für den Hobbyprogrammierer. Die vielen Beispiele demonstrieren Programmiertechniken, die in modernen Programmen eingesetzt werden.

Der ernsthafte Programmierer oder Student muss C++ in seinem Köcher der Programmierfähigkeiten haben. Fundiertes Wissen in C++ zu haben, kann den Unterschied machen, ob man einen bestimmten Job bekommt oder nicht.

1.4 Was

Der C++ *Wochenend Crashkurs* ist mehr als nur ein Buch. Er ist ein vollständiges Entwicklungspaket. Eine CD-ROM enthält die berühmte GNU C++-Umgebung.

Sie benötigen ein Textprogramm, wie z.B. Microsoft Word, um Texte bearbeiten zu können. Und sie brauchen eine C++-Entwicklungsumgebung, um Programme in C++ zu erzeugen und auszuführen.

Viele Leser werden bereits eine eigene Entwicklungsumgebung besitzen, wie z.B. Microsoft's Visual C++. Für die Leser, die noch über keine Entwicklungsumgebung verfügen, enthält der C++ *Wochenend Crashkurs* das GNU C++.

GNU C++ ist kein abgespecktes oder laufzeitbeschränktes Programm. Das GNU C++-Paket ist eine vollwertige Entwicklungsumgebung. Der C++ *Wochenend Crashkurs* enthält vollständige Anleitungen zur Installation von GNU C++ und Visual C++.

1.5 Wie

Der C++ *Wochenend Crashkurs* ist für ein Wochenende gedacht. Fangen Sie am Freitagabend an, dann sind Sie am Sonntagnachmittag fertig.

Dieses Ein-Wochen-Format ist

- ideal für Studenten, die mit ihren Mitstudenten gleichziehen möchten,
- ideal für den Programmierer, der seine Fähigkeiten erweitern will, und
- ideal für jeden, der C++ lernen möchte, während die Kinder bei der Oma sind.

Natürlich können Sie das Buch auch etwas langsamer durcharbeiten, wenn Sie das lieber tun. Jeder Teil von 4 bis 6 Sitzungen kann separat gelesen werden.

Der Leser sollte jede der 30 Sitzungen innerhalb von 30 Minuten durcharbeiten können. Zeitmarkierungen helfen, die Zeit im Auge zu behalten.

Am Ende jeder Sitzung befinden sich Fragen, die dem Leser zur Selbsteinschätzung dienen sollen. Eine Menge schwierigerer Fragen, die helfen sollen, das Erlernte zu festigen, befindet sich am Ende jeden Buchteils.

1.6 Überblick

Der C++ *Wochenend Crashkurs* präsentiert seine Sitzungen in Gruppen von 4 bis 6 Kapiteln, die in 6 Buchteile organisiert sind.

1.6.1 Freitagabend – Einführung in die Programmierung

Dieser Teil führt Programmierkonzepte ein und führt Sie durch Ihr erstes Programm.

1.6.2 Samstagmorgen – Einstieg in C++

Dieser Teil behandelt Themen wie Anweisungssyntax, Operatoren und elementare Funktionen.

1.6.3 Samstagnachmittag – Strukturen und Zeiger

Hier beschäftigt sich der Leser mit dem etwas komplizierteren Thema der Zeigervariablen, zusammen mit ihrem Einsatz in verketteten Listen, Arrays und Objekten.

1.6.4 Samstagabend – Einführung in die objektorientierte Programmierung

Das ist ein Punkt mit Schlüsselcharakter – Themen wie C++-Strukturen, die Grundlage der objektorientierten Programmierung sind, werden besprochen.

1.6.5 Sonntagmorgen – Objektorientierte Programmierung

Hier ist die Hauptschlagader. Dieser Teil taucht in die Syntax und die Bedeutung der objektorientierten Programmierung ein.

1.6.6 Sonntagnachmittag – Abschluss

Dieser Teil stellt einige fortgeschrittene Themen dar, wie Fehlerbehandlung und das Überladen von Operatoren.

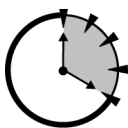
Jeder Teil endet mit einer Diskussion von Debug-Techniken, um die offensichtlichen Fehler in Ihren Programmen zu finden und zu entfernen. Die Komplexität dieser Techniken ist den Fähigkeiten angepasst, die der Leser in der Sitzung erlernt hat.

Der Anhang enthält weiterführende Programmierprobleme zu jeder Sitzung.

1.7 Layout und Features

Niemand sollte versuchen, sich ohne Pause durch das Material durchzuschlagen. Nach jeder Sitzung und am Ende eines jeden Teils finden Sie einige Fragen, um Ihr Wissen zu überprüfen und Ihre neu erworbenen Fähigkeiten auszuprobieren. Machen Sie eine Pause, holen Sie sich einen Snack, trinken Sie einen Kaffee und gehen Sie dann in die nächste Sitzung.

Entlang Ihres Weges finden Sie Markierungen, die Ihnen bei der Orientierung helfen sollen. Sie sagen Ihnen, wo Sie sich gerade befinden und weisen Sie auf interessante Punkte hin, die Sie nicht verpassen sollten. Wenn Sie eine Sitzung durcharbeiten, halten Sie nach den folgenden Zeichen Ausschau:



Dieses und ähnliche Icons zeigen Ihnen, wie weit Sie bereits in der Sitzung gekommen sind.

20 Min.

Es gibt eine Reihe von Icons, die Sie auf spezielle Informationen hinweisen sollen:



Dieses Zeichen weist auf Informationen hin, die Sie im Gedächtnis behalten sollten. Sie werden Ihnen später noch nützlich sein.



Hier erhalten Sie hilfreiche Hinweise darauf, wie Sie eine Sache am besten ausführen oder erfahren eine Technik, die Ihre Programmierung einfacher macht.



Tun Sie das niemals!



Dieses Zeichen weist auf Informationen hin, die Sie auf der CD-ROM finden, die diesem Buch beiliegt.

1.8 Konventionen in diesem Buch

Abgesehen von den Zeichen, die Sie gerade gesehen haben, gibt es nur zwei Konventionen, die in diesem Buch verwendet werden:

- Programmcode, der im normalen Text verwendet wird, erscheint in einem speziellen Font, wie in folgendem Beispiel zu sehen ist:
Wenn ich die Funktion `main( )` schreibe, kann ich mich auf den Wert konzentrieren, der von der Funktion `sumSequence( )` zurückgegeben wird, ohne darüber nachzudenken, wie diese Funktion intern arbeitet.
- Programmbeispiele, die sich nicht im normalen Text befinden, werden wie folgt dargestellt:

```
float fVariable1 = 10.0;  
float fVariable2 = (10 / 3) * 3;  
fVariable1 == fVariable2; // sind die beiden gleich?
```

1.9 Was fehlt noch?

Nichts. Öffnen Sie die erste Seite Ihres Arbeitsbuches und halten Sie die Uhr bereit. Es ist Freitagabend, und Sie haben zwei Tage Zeit.

Inhalt

Freitag

Teil 1 – Freitagabend	2
Lektion 1 – Was ist Programmierung?	3
1.1 Ein menschliches Programm	4
1.1.1 Der Algorithmus	4
1.2 Der Prozessor	4
1.3 Das Programm	5
1.4 Computerprozessoren	7
Zusammenfassung	7
Selbsttest	7
Lektion 2 – Ihr erstes Programm in Visual C++	8
2.1 Installation von Visual C++	8
2.2 Ihr erstes Programm	9
2.3 Erzeugen Ihres Programms	10
2.4 Ausführen Ihres Programms	14
2.5 Abschluss	15
2.5.1 Programmausgabe	16
2.5.2 Visual C++-Hilfe	16
Zusammenfassung	17
Selbsttest	17
Lektion 3 – Ihr erstes C++-Programm mit GNU C++	18
3.1 Installation von GNU C++	18
3.2 Ihr erstes Programm	20
3.2.1 Eingabe des C++-Codes	20
3.3 Erzeugen Ihres Programms	23
3.4 Ausführen Ihres Programms	26

3.5	Abschluss	27
3.5.1	Programmausgabe	27
3.5.2	GNU C++-Hilfe	27
	Zusammenfassung	28
	Selbsttest	28
Lektion 4 – C++-Instruktionen		29
4.1	Das Programm	29
4.2	Das C++-Programm erklärt	30
4.2.1	Der grundlegende Programmaufbau	30
4.2.2	Kommentare	31
4.2.3	Noch mal der Rahmen	32
4.2.4	Anweisungen	32
4.2.5	Deklarationen	32
4.2.6	Eingabe/Ausgabe	33
4.2.7	Ausdrücke	33
4.2.8	Zuweisung	34
4.2.9	Ausdrücke (Fortsetzung)	34
	Zusammenfassung	34
	Selbsttest	35
	Freitagabend – Zusammenfassung	36

Samstag

Teil 2 – Samstagmorgen	40

Lektion 5 – Variablentypen	41	
5.1	Dezimalzahlen	42
5.1.1	Begrenzungen von int in C++	42
5.1.2	Lösen des Abschneideproblems	45
5.1.3	Grenzen von Gleitkommazahlen	46
5.2	Andere Variablentypen	47
5.2.1	Typen von Konstanten	48
5.2.2	Sonderzeichen	48
5.3	Gemischte Ausdrücke	50
	Zusammenfassung	51
	Selbsttest	52
Lektion 6 – Mathematische Operationen	53	
6.1	Arithmetische Operatoren	54
6.2	Ausdrücke	54
6.3	Vorrang von Operatoren	55
6.4	Unäre Operatoren	56
6.5	Zuweisungsoperatoren	57
	Zusammenfassung	58
	Selbsttest	58

XIV Inhalt

Lektion 7 – Logische Operationen	.59
7.1 Einfache logische Operatoren	.59
7.1.1 Kurze Schaltkreise und C++	.61
7.1.2 Logische Variablentypen	.62
7.2 Binäre Zahlen	.62
7.3 Bitweise logische Operationen	.63
7.3.1 Die Einzelbit-Operatoren	.63
7.3.2 Die bitweisen Operatoren	.64
7.3.3 Ein einfacher Test	.65
7.3.4 Warum?	.66
Zusammenfassung	.67
Selbsttest	.68
Lektion 8 – Kommandos zur Flusskontrolle	.69
8.1 Das Verzweigungskommando	.69
8.2 Schleifenkommandos	.71
8.2.1 Die while-Schleife	.71
8.2.2 Die for-Schleife	.74
8.2.3 Spezielle Schleifenkontrolle	.75
8.3 Geschachtelte Kontrollkommandos	.78
8.4 Können wir switchen?	.79
Zusammenfassung	.80
Selbsttest	.80
Lektion 9 – Funktionen	.81
9.1 Code einer Sammelfunktion	.81
9.1.1 Sammelcode	.82
9.2 Funktion	.84
9.2.1 Warum Funktionen?	.84
9.2.2 Einfache Funktionen	.85
9.2.3 Funktionen mit Argumenten	.85
9.2.4 Mehrere Funktionen mit gleichem Namen	.88
9.3 Funktionsprototypen	.89
9.4 Verschiedene Speichertypen	.90
Zusammenfassung	.91
Selbsttest	.91
Lektion 10 – Debuggen	.92
10.1 Fehlertypen	.92
10.2 Die Technik der Ausgabeanweisungen	.93
10.3 Abfangen von Bug Nr. 1	.94
10.3.1 Visual C++	.95
10.3.2 GNU C++	.96
10.4 Abfangen von Bug Nr. 2	.97
Zusammenfassung	.100
Selbsttest	.100
Samstagmorgen – Zusammenfassung	.101

Teil 3 – Samstagnachmittag	104
Lektion 11 – Das Array	105
11.1 Was ist ein Array?	105
11.1.2 Ein Array in der Praxis	108
11.1.3 Initialisierung eines Array	110
11.1.4 Warum Arrays benutzen?	110
11.1.5 Arrays von Arrays	111
11.2 Arrays von Zeichen	111
11.3 Manipulation von Zeichenketten	113
11.3.1 Unsere eigene Verbindungsfunktion	114
11.3.2 Funktionen für C++-Zeichenketten	116
11.3.3 Wide Character	117
11.4 Obsolete Ausgabefunktionen	117
Zusammenfassung	118
Selbsttest	118
Lektion 12 – Einführung in Klassen	119
12.1 Gruppieren von Daten	119
12.1.1 Ein Beispiel	120
12.1.2 Das Problem	122
12.2 Die Klasse	122
12.2.2 Beispielprogramm	124
12.2.3 Vorteile	126
Zusammenfassung	126
Selbsttest	126
Lektion 13 – Einstieg C++-Zeiger	127
13.1 Was ist deine Adresse?	128
13.2 Einführung in Zeigervariablen	129
13.3 Typen von Zeigern	132
13.4 Übergabe von Zeigern an Funktionen	133
13.4.1 Wertübergabe	133
13.4.2 Übergabe von Zeigerwerten	134
13.4.3 Referenzübergabe	135
13.5 Heap-Speicher	135
13.5.1 Geltungsbereich	135
13.5.2 Das Geltungsbereichsproblem	137
13.5.3 Die Heap-Lösung	137
Zusammenfassung	138
Selbsttest	139
Lektion 14 – Mehr über Zeiger	140
14.1 Zeiger und Arrays	140
14.1.1 Operationen auf Zeigern	141
14.1.2 Zeichenarrays	145
14.1.3 Operationen auf unterschiedlichen Zeigertypen	148

XVI Inhalt

14.2	Argumente eines Programms	150
14.2.1	Arrays von Zeigern	150
14.2.2	Arrays von Zeichenketten	151
14.2.3	Die Argumente von <code>main()</code>	152
	Zusammenfassung	154
	Selbsttest	155
Lektion 15 – Zeiger auf Objekte		156
15.1	Zeiger auf Objekte	156
15.1.1	Übergabe von Objekten	157
15.1.2	Referenzen	159
15.1.3	Rückgabe an den Heap	159
15.2	Die Datenstruktur Array	160
15.3	Verkettete Listen	160
15.3.1	Anfügen am Kopf der verketteten Liste	161
15.3.2	Andere Operationen auf verketteten Listen	162
15.3.3	Eigenschaften verketteter Listen	164
15.4	Ein Programm mit verkettetem <code>NameData</code>	165
15.5	Andere Container	168
	Zusammenfassung	168
	Selbsttest	168
Lektion 16 – Debuggen II		169
16.1	Welcher Debugger?	170
16.2	Das Testprogramm	170
16.3	Einzelschritte durch ein Programm	172
16.4	Einzelschritte in eine Funktion hinein	173
16.5	Verwendung von Haltepunkten	175
16.6	Ansehen und Modifizieren von Variablen	176
16.7	Verwendung des Visual C++-Debuggers	179
	Zusammenfassung	181
	Selbsttest	182
	Samstagnachmittag – Zusammenfassung	183
Teil 4 – Samstagabend		186
Lektion 17 – Objektprogrammierung		187
17.1	Abstraktion und Mikrowellen	187
17.1.1	Funktionale Nachos	188
17.1.2	Objektorientierte Nachos	188
17.2	Klassifizierung und Mikrowellen	189
17.2.1	Warum solche Objekte bilden?	189
17.2.2	Selbstenthaltende Klassen	190
	Zusammenfassung	191
	Selbsttest	191

Lektion 18 – Aktive Klassen	192
18.1 Klassenrückblick	192
18.2 Grenzen von struct	193
18.2.1 Eine funktionale Lösung	194
18.3 Definition einer aktiven Klasse	195
18.3.1 Namensgebung für Elementfunktionen	196
18.4 Definition einer Elementfunktion einer Klasse	197
18.5 Schreiben von Elementfunktionen außerhalb einer Klasse	198
18.5.1 Include-Dateien	199
18.6 Aufruf einer Elementfunktion	200
18.6.1 Aufruf einer Elementfunktion über einen Zeiger	201
18.6.2 Zugriff auf andere Elemente von einer Elementfunktion aus	202
18.7 Überladen von Elementfunktionen	204
Zusammenfassung	205
Selbsttest	205
Lektion 19 – Erhalten der Klassenintegrität	206
19.1 Erzeugen und Vernichten von Objekten	206
19.1.1 Der Konstruktor	207
19.1.2 Der Destruktor	212
19.2 Zugriffskontrolle	214
19.2.1 Das Schlüsselwort protected	214
19.2.2 Statische Datenelemente	217
Zusammenfassung	218
Selbsttest	219
Lektion 20 – Klassenkonstruktoren II	220
20.1 Konstruktoren mit Argumenten	220
20.2 Konstruktion von Klassenelementen	223
20.3 Reihenfolge der Konstruktion	228
20.3.1 Lokale Objekte werden in der Reihenfolge konstruiert	229
20.3.2 Statische Objekte werden nur einmal angelegt	229
20.3.3 Alle globalen Variablen werden vor main() erzeugt	229
20.3.4 Keine bestimmte Reihenfolge für globale Objekte	229
20.3.5 Elemente werden in der Reihenfolge ihrer Deklaration konstruiert	231
20.3.6 Destruktoren in umgekehrter Reihenfolge wie Konstruktoren	231
20.4 Der Kopierkonstruktor	231
20.4.1 Flache Kopie gegen tiefe Kopie	233
20.4.2 Ein Fallback-Kopierkonstruktor	234
Zusammenfassung	235
Selbsttest	236
Samstagabend – Zusammenfassung	237

XVIII Inhalt**Sonntag****Teil 5 – Sonntagmorgen** 240**Lektion 21 – Vererbung** 241

21.1	Vorteile der Vererbung	241
21.2	Faktorisieren von Klassen	242
21.3	Implementierung von Vererbung in C++	243
21.4	Unterklassen konstruieren	246
21.5	Die Beziehung HAS_A	249
	Zusammenfassung	251
	Selbsttest	251

Lektion 22 – Polymorphie 252

22.1	Elementfunktionen überschreiben	252
22.2	Einstieg in Polymorphie	254
22.3	Polymorphie und objektorientierte Programmierung	254
22.4	Wie funktioniert Polymorphie?	257
22.5	Was ist eine virtuelle Funktion nicht?	259
22.6	Überlegungen zu virtual	261
	Zusammenfassung	262
	Selbsttest	262

Lektion 23 – Abstrakte Klassen und Faktorisieren 263

23.1	Faktorisieren	263
23.2	Abstrakte Klassen	267
23.2.1	Deklaration einer abstrakten Klasse	267
23.2.2	Erzeugung einer konkreten Klasse aus einer abstrakten Klasse	269
23.2.3	Warum ist eine Unterklasse abstrakt?	271
23.2.4	Ein abstraktes Objekt an eine Funktion übergeben	272
23.2.5	Warum werden rein virtuelle Funktionen benötigt?	273
	Zusammenfassung	274
	Selbsttest	275

Lektion 24 – Mehrfachvererbung 276

24.1	Wie funktioniert Mehrfachvererbung?	276
24.2	Uneindeutigkeiten bei der Vererbung	278
24.3	Virtuelle Vererbung	279
24.4	Konstruktion von Objekten bei Mehrfachnennung	285
24.5	Eine Meinung dagegen	285
	Zusammenfassung	286
	Selbsttest	287

Lektion 25 – Große Programme	288
25.1 Warum Programme aufteilen?	288
25.2 Trennung von Klassendefinition und Anwendungsprogramm	289
25.2.1 Aufteilen des Programms	289
25.2.2 Die #include-Direktive	290
25.2.3 Anwendungscode aufteilen	292
25.2.4 Projektdatei	293
25.2.5 Erneute Betrachtung des Standard-Programm-Templates	295
25.2.6 Handhabung von Outline-Elementfunktionen	296
Zusammenfassung	297
Selbsttest	297
Lektion 26 – C++-Präprozessor	298
26.1 Der C++-Präprozessor	298
26.2 Die #include-Direktive	299
26.3 Die Direktive #define	299
26.3.1 Definition von Makros	300
26.3.2 Häufige Fehler bei der Verwendung von Makros	300
26.4 Compiler-Kontrolle	302
26.4.1 Die #if-Direktive	302
26.4.2 Die #ifdef-Direktive	303
Zusammenfassung	305
Selbsttest	306
Sonntagmorgen – Zusammenfassung	307
Teil 6 – Sonntagnachmittag	310
Lektion 27 – Überladen von Operatoren	311
27.1 Warum sollte ich Operatoren überladen?	312
27.2 Was ist die Beziehung zwischen Operatoren und Funktionen?	312
27.3 Wie funktioniert das Überladen von Operatoren?	313
27.3.1 Spezielle Überlegungen	316
27.4 Ein detaillierterer Blick	316
27.5 Operatoren als Elementfunktionen	318
27.6 Eine weitere Irritation durch Überladen	320
27.7 Wann sollte ein Operator ein Element sein?	321
27.8 Cast-Operator	321
Zusammenfassung	324
Selbsttest	324

Lektion 28 – Der Zuweisungsoperator	325
28.1 Warum ist das Überladen des Zuweisungsoperators kritisch?	325
28.1.1 Vergleich mit dem Kopierkonstruktor	326
28.2 Wie den Zuweisungsoperator überladen?	326
28.2.1 Zwei weitere Details des Zuweisungsoperators	329
28.3 Ein Schlupfloch	330
Zusammenfassung	331
Selbsttest	331
Lektion 29 – Stream-I/O	332
29.1 Wie funktioniert Stream-I/O?	332
29.2 Die Unterklassen fstream	333
29.3 Die Unterklassen strstream	337
29.3.1 Vergleich von Techniken der Zeichenkettenverarbeitung	338
29.4 Manipulatoren	341
29.5 Benutzerdefinierte Inserter	343
29.6 Schlaue Inserter	344
29.7 Aber warum die Shift-Operatoren?	346
Zusammenfassung	347
Selbsttest	347
Lektion 30 – Ausnahmen	348
30.1 Konventionelle Fehlerbehandlung	348
30.2 Warum benötigen wir einen neuen Fehlermechanismus?	349
30.3 Wie arbeiten Ausnahmen?	350
30.3.1 Warum ist der Ausnahmemechanismus eine Verbesserung?	352
30.4 Abfangen von Details, die für mich bestimmt sind	352
30.4.1 Was kann ich »werfen«?	354
30.5 Verketteten von catch-Blöcken	357
Zusammenfassung	358
Selbsttest	359
Sonntagnachmittag – Zusammenfassung	360
Anhang A: Antworten auf die Wiederholungsfragen	363
Anhang B: Ergänzende Probleme	381
Anhang C: Was ist auf der CD-Rom	393
Index	395
GNU General Public License	421



Freitag



Samstag



Sonntag

Freitagabend

Teil 1

Lektion 1

Was ist Programmierung?

Lektion 2

Ihr erstes Programm in Visual C++

Lektion 3

Ihr erstes C++-Programm mit GNU C++

Lektion 4

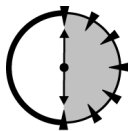
C++-Instruktionen

Was ist Programmierung?



Checkliste

- ☒ Die Prinzipien des Programmierens erlernen
- ☒ Lernen, ein menschlicher Prozessor zu sein
- ☒ Lernen, einen Reifen zu wechseln



30 Min.

Der Webster's New World College Dictionary bietet mehrere Definitionen des Substantivs »Programm« an. Die erste lautet »eine Proklamation, ein Prospekt, oder eine Inhaltsangabe«. Das trifft das, worum es uns geht, nicht so richtig. Erst die sechste Definition passt besser: »eine logische Sequenz codierter Instruktionen, die Operationen beschreiben, die von einem Computer ausgeführt werden sollen, um ein Problem zu lösen oder Daten zu verarbeiten«.

Nach kurzem Nachdenken habe ich festgestellt, dass diese Definition ein wenig restriktiv ist. Zum einen weiß man in der Phrase »eine logische Sequenz codierter Instruktionen ...«, nicht, ob die Instruktionen gekryptet, d.h. kodiert sind oder nicht, und der Begriff »logisch« ist sehr einschränkend. Ich selber habe schon Programme geschrieben, die nicht sehr viel gemacht haben, bevor sie abgestürzt sind. In der Tat stürzen die meisten meiner Programme ab, bevor sie irgend etwas tun. Das scheint nicht sehr logisch zu sein. Zum anderen »... um ein Problem zu lösen oder Daten zu verarbeiten«: Was ist mit dem Steuerungscomputer meiner Klimaanlage im Auto? Er löst kein Problem, das mir bewusst ist. Ich mag die Klimaanlage so wie sie ist – anschalten und ausschalten auf Knopfdruck.

Das größte Problem mit Webster's Definition ist die Phrase »die von einem Computer ausgeführt werden sollen ...«. Ein Programm hat nicht unbedingt etwas mit Computern zu tun. (Es sei denn, Sie zählen das konfuse Etwas zwischen Ihrem Stereokopfhörer dazu. In diesem Fall können Sie behaupten, dass alles, was Sie tun, etwas mit Computern zu tun hat.) Ein Programm kann ein Leitfaden sein, für etwas, das wenigstens ein Körnchen Intelligenz besitzt – sogar für mich. (Vorausgesetzt, ich liege nicht unterhalb dieser Körnchengrenze.) Lassen Sie uns betrachten, wie wir ein Programm schreiben würden, um ein menschliches Verhalten anzuleiten.

1.1 Ein menschliches Programm

Ein Programm für einen Menschen zu schreiben ist viel einfacher, als ein Programm für eine Maschine zu schreiben. Uns sind Menschen vertraut – wir verstehen Menschen gut. Ein besonders wichtiger Aspekt dieser Vertrautheit ist die gemeinsame Sprache. In diesem Abschnitt schreiben wir ein »menschliches Programm« und studieren seine Teile. Lassen Sie uns das Problem des Reifenwechsels betrachten.

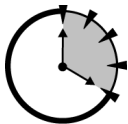
1.1.1 Der Algorithmus

Das Wechseln eines Reifens ist recht einfach. Die Schritte gehen etwa so:

1. Heben Sie den Wagen mit dem Wagenheber an.
2. Entfernen Sie die Radmutter, die den Reifen am Wagen befestigen.
3. Entfernen Sie den platten Reifen.
4. Setzen Sie den Ersatzreifen ein.
5. Schrauben Sie ihn fest.
6. Lassen Sie Ihren Wagen wieder herunter.

(Ich weiß, dass Rad und Reifen nicht das Gleiche sind – Sie entfernen nicht den Reifen vom Auto, sondern das Rad. Zwischen diesen beiden Begriffen hin und her zu springen, ist verwirrend. Nehmen Sie daher einfach an, dass das Wort »Reifen« in diesem Beispiel synonym zu »Rad« verwendet wird.)

Das ist das grundlegende Programm. Ich könnte mit diesen Anweisungen alle meine platten Reifen ersetzen, die ich bisher hatte. Um genauer zu sein, ist dies ein *Algorithmus*. Ein Algorithmus ist eine Beschreibung von auszuführenden Schritten, in der Regel auf einem hohen Abstraktionsniveau. Ein Algorithmus ist für ein Programm wie eine Beschreibung der Prinzipien des Fernsehens für die Schaltkreise in einem Fernseher.



20 Min.

1.2 Der Prozessor

Um etwas ans Laufen zu bekommen, muss ein Algorithmus mit etwas kombiniert werden, das ihn ausführt: einem *Prozessor*. Unser Programm zum Wechseln des Reifens setzt z.B. voraus, dass ein Mann (hups, ich meine, eine *Person*) vorhanden ist, um den Wagen anzuheben, die Schrauben zu lösen und den neuen Reifen an seinen Platz zu heben. Die erwähnten Objekte – Auto, Reifen und Schrauben – sind nicht in der Lage, sich alleine zu bewegen.

Lassen Sie uns annehmen, dass unser Prozessor nur einige Wörter versteht, und diese sehr wörtlich. Lassen Sie uns weiter annehmen, dass unser Prozessor die folgenden Substantive versteht, die im Reifenwechselgeschäft geläufig sind:

Auto
Reifen
Radmutter
Wagenheber
Schraubenschlüssel

Lektion 1 – Was ist Programmierung?**5**

(Die beiden letzten Objekte wurden in unserem Algorithmus zum Reifenwechseln nicht erwähnt, aber sie waren in Phrasen wie »ersetzen Sie den Reifen« enthalten. Das ist das Problem mit Algorithmen – so vieles ist nicht explizit ausgesprochen).

Lassen Sie uns weiterhin annehmen, dass unser Prozessor folgende Verben versteht:

```
greifen  
bewegen  
loslassen  
drehen
```

Schließlich muss unsere Prozessorperson zählen und einfache Entscheidungen treffen können.

Das ist alles, was unsere Prozessorperson fürs Reifenwechseln versteht. Alle weiteren Anweisungen rufen bei ihr nur einen ratlosen Blick hervor.

1.3 Das Programm

Mit dem gegebenen Vokabular ist es unserem Prozessor nicht möglich, Anweisungen der Form »entferne die Radmutter vom Auto« auszuführen. Das Wort »entfernen« kommt nicht im Vokabular des Prozessors vor. Außerdem wird der Schraubenschlüssel, mit dem die Radmutter gelöst werden müssen, nicht erwähnt. (Das sind die Dinge, die in der normalen Sprache auch unausgesprochen funktionieren.)

Die folgenden Schritte beschreiben den Vorgang »Radmutter entfernen« unter Verwendung von Begriffen, die der Prozessor versteht:

1. Schraubenschlüssel greifen
2. Schraubenschlüssel auf Radmutter bewegen
3. Schraubenschlüssel fünfmal gegen Uhrzeigersinn drehen
4. Schraubenschlüssel von Radmutter entfernen
5. Schraubenschlüssel loslassen

Lassen Sie uns jeden Schritt einzeln ansehen.

Der Prozessor beginnt mit Schritt 1 und arbeitet jeden Schritt einzeln ab, bis er bei Schritt 5 angekommen ist. In der Ausdrucksweise der Programmierung sagen wir, dass das Programm von Schritt 1 nach Schritt 5 fließt, obwohl das Programm nirgends hinget – die Prozessorperson tut dies.

In Schritt 1 nimmt die Prozessorperson den Schraubenschlüssel. Es kann sein, dass der Prozessor den Schraubenschlüssel bereits in der Hand hat, aber wir können nicht davon ausgehen. In Schritt 2 wird der Schraubenschlüssel auf der Radmutter platziert. Schritt 3 löst die Radmutter. Die Schritte 4 und 5 schließlich geben den Schraubenschlüssel zurück.

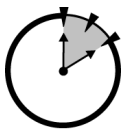
Ein Problem mit diesem Algorithmus ist ganz offensichtlich. Wie können wir wissen, dass fünf Umdrehungen ausreichen, um die Radmutter zu lösen? Wir könnten die Anzahl der Umdrehungen einfach so groß machen, dass sie ausreicht, um jede beliebige Radmutter zu lösen. Diese Lösung wäre nicht nur Verschwendung, sondern kann auch mal nicht funktionieren. Was wird unser Prozessor tun, wenn die Radmutter herunterfällt und der Prozessor die Anweisung erhält, den Schraubenschlüssel erneut zu drehen? Wird das den Prozessor verwirren und zum Anhalten bewegen?

Das folgende Programm nutzt die eingeschränkten Fähigkeiten des Prozessors, einfache Entscheidungen zu treffen, um eine Radmutter korrekt zu entfernen:

6

Freitagabend

1. Schraubenschlüssel greifen
2. Schraubenschlüssel auf Radmutter bewegen
3. Solange Radmutter nicht gelöst
4. <
5. Schraubenschlüssel gegen Uhrzeigersinn drehen
6. >
7. Schraubenschlüssel von Radmutter entfernen
8. Schraubenschlüssel loslassen



10 Min.

Das Programm durchläuft die Schritte 1 und 2 wie zuvor. Schritt 3 ist vollständig anders. Der Prozessor wird angewiesen, die Schritte, die in den auf Schritt 3 folgenden Klammern eingeschlossen sind, so lange auszuführen, bis eine bestimmte Bedingung erfüllt ist. In diesem Fall, bis die Radmutter gelöst ist. Sobald die Radmutter gelöst ist, fährt die Prozessorperson bei Schritt 7 fort. Die Schritte 3 bis 6 werden als *Schleife* bezeichnet, da der Prozessor sie wie einen Kreis durchläuft.

Diese Lösung ist viel besser, da sie keine Annahmen in Bezug auf die Anzahl der Umdrehungen macht, die benötigt werden, um eine Radmutter zu lösen. Das Programm ist außerdem nicht verschwenderisch, da keine unnötigen Umdrehungen ausgeführt werden. Und das Programm weist den Prozessor nie an, eine Schraube zu drehen, die nicht mehr da ist.

So schön das Programm ist, hat es doch noch ein Problem: Es entfernt nur eine einzige Radmutter. Die meisten Autos haben fünf Radmutter pro Reifen. Wir können die Schritte 2 bis 7 fünfmal wiederholen, einmal für jede Radmutter. Immer fünf Radmutter zu entfernen, funktioniert aber auch nicht. Kleinwagen haben manchmal nur vier Radmutter, größere Autos und kleine Lastwagen haben meist sechs Radmutter.

Das folgende Programm erweitert die letzte Lösung auf alle Radmutter eines Reifens, unabhängig von der Anzahl der Radmutter.

1. Schraubenschlüssel greifen
2. Für jede Radmutter
3. <
4. Schraubenschlüssel auf Radmutter bewegen
5. Solange Radmutter nicht gelöst
6. <
7. Schraubenschlüssel gegen Uhrzeigersinn drehen
8. >
9. Schraubenschlüssel von Radmutter entfernen
10. >
11. Schraubenschlüssel loslassen

Das Programm fängt wie immer mit dem Greifen des Schraubenschlüssels an. Danach durchläuft das Programm eine Schleife zwischen den Schritten 2 bis 10 über alle Radmutter. Schritt 9 entfernt den Schraubenschlüssel von der Radmutter, bevor in Schritt 2 mit der nächsten Radmutter fortgefahren wird.

Beachten Sie, wie die Schritte 5 bis 8 wiederholt werden, bis die Radmutter gelöst ist. Die Schritte 5 bis 8 werden als *innere Schleife* bezeichnet, während die Schritte 2 bis 10 als *äußere Schleife* bezeichnet werden.

Das gesamte Programm besteht aus einer Kombination gleichartiger Lösungen für jeden der sechs Schritte im ursprünglichen Programm.

1.4 Computerprozessoren

Ein Computerprozessor arbeitet sehr ähnlich wie ein menschlicher Prozessor. Ein Computerprozessor folgt wörtlich einer Kette von Kommandos, die mit einem endlichen Vokabular erzeugt wurde.

Einen Reifen von einem Auto zu entfernen, scheint eine einfache Aufgabe zu sein, und unsere Prozessorperson benötigt 11 Anweisungen, um ein einzelnen Reifen zu wechseln. Wie viele Anweisungen werden benötigt, um die vielen tausend Pixel auf dem Bildschirm zu bewegen, wenn der Benutzer die Maus bewegt?



0 Min.

Im Gegensatz zu einem menschlichen Prozessor sind Prozessoren aus Silizium extrem schnell. Ein Pentium III-Prozessor kann einige 100 Millionen Schritte pro Sekunde ausführen. Es bedarf einiger Millionen Anweisungen, um ein Fenster zu bewegen, aber weil der Computerprozessor so schnell ist, bewegt sich das Fenster flüssig über den Bildschirm.

Zusammenfassung

Dieses Kapitel hat grundlegende Prinzipien der Programmierung eingeführt anhand eines Beispiels, um einen sehr dummen, aber außerordentlich folgsamen Mechaniker in die Kunst des Reifenwechsels einzuweisen.

- Computer tun das, was Sie ihnen sagen – nicht weniger, aber natürlich auch nicht mehr.
- Computerprozessoren haben ein kleines, aber wohldefiniertes Vokabular.
- Computerprozessoren sind clever genug, einfache Entscheidungen zu treffen.

Selbsttest

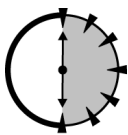
1. Nennen Sie Substantive, die ein »menschlicher Prozessor« verstehen müsste, um Geschirr abzuwaschen.
2. Nenne Sie einige Verben.
3. Welche Art von Entscheidungen müsste ein Prozessor treffen können?



Ihr erstes Programm in Visual C++

Checkliste

- ☒ Ihr erstes C++-Programm in Visual C++ schreiben
- ☒ Aus Ihrem C++-Code ein ausführbares Programm erzeugen
- ☒ Ihr Programm ausführen
- ☒ Hilfe bei der Programmierung bekommen



30 Min.

Kapitel 1 handelte von Programmen für Menschen. Dieses und das nächste Kapitel beschreiben, wie Sie einen Computer in C++ programmieren. Dieses Kapitel behandelt die Programmierung mit Visual C++, während sich das nächste Kapitel mit dem frei verfügbaren GNU C++ befasst, das auch auf der beiliegenden CD-ROM zu finden ist.



Haben Sie keine Angst vor den Bezeichnungen Visual C++ und GNU C++. Beide Compiler stellen Implementierungen des C++-Standards dar. Jeder der Compiler kann jedes Programm in diesem Buch übersetzen.

Das Programm, das wir schreiben wollen, konvertiert eine Temperatur, die der Benutzer in Grad Celsius eingibt, in Grad Fahrenheit.

2.1 Installation von Visual C++

Sie müssen Visual C++ auf Ihrem Rechner installieren, bevor Sie ein Visual C++-Programm schreiben können. Das Paket Visual C++ wird benutzt, um C++-Programme zu schreiben, und daraus .EXE-Programme zu erzeugen, die der Computer versteht.



Visual C++ ist nicht auf der beiliegenden CD-ROM enthalten. Sie müssen Visual C++ separat erwerben, entweder als Bestandteil von Visual Studio, oder als Einzelprodukt. Den sehr guten GNU C++-Compiler finden Sie auf der CD-ROM.

2.2 Ihr erstes Programm

Ein C++-Programm beginnt sein Leben als Textdatei, die C++-Anweisungen enthält. Ich werde Sie Schritt für Schritt durch das erste Programm führen.

Starten Sie Visual C++. Für Visual Studio 6.0, klicken Sie auf »Start«, gefolgt von den Menüoptionen »Programme« und »Microsoft Visual Studio 6.0«. Von dort wählen Sie »Microsoft Visual C++ 6.0« aus.

Visual C++ sollte zwei leere Fenster zeigen, die mit Ausgabe und Arbeitsbereich bezeichnet sind. Wenn noch andere Fenster gezeigt werden, oder die beiden genannten Fenster nicht leer sind, dann hat jemand bereits Visual C++ auf Ihrem Computer benutzt. Um all dies zu schließen, wählen Sie »Datei« gefolgt von »Arbeitsbereich schließen« aus.

Erzeugen Sie eine leere Textdatei durch Klicken auf das kleine Icon ganz links in der Leiste, wie in Abbildung 2.1 zu sehen ist.

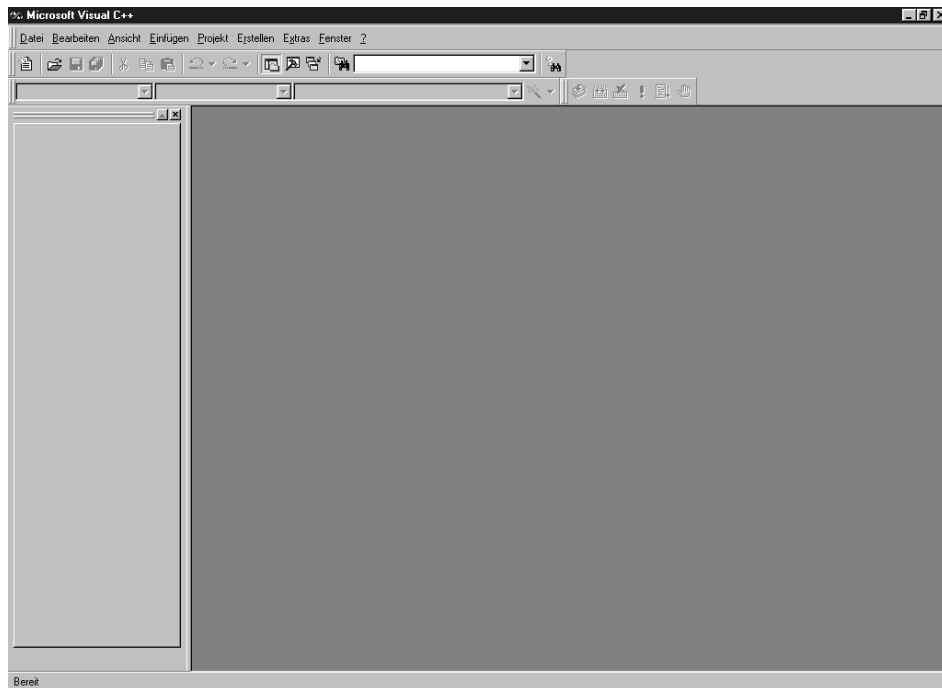


Abbildung 2.1: Sie beginnen damit, ein C++-Programm zu schreiben, indem Sie eine neue Textdatei anlegen.



Machen Sie sich keine Gedanken über das Einrücken – es kommt nicht darauf an, ob eine Zeile zwei oder drei Zeichen eingerückt ist. Groß- und Kleinschreibung sind jedoch wichtig. Für C++ sind »Betrügen« und »betrügen« nicht gleich.



Sie können sich das Programm *Conversion.cpp* von der beiliegenden CD-ROM herunterladen.

Geben Sie das folgende Programm genau wie hier abgedruckt ein. (Oder kopieren Sie es sich von der CD-ROM.)

```
//
// Programm konvertiert Temperaturen von Grad Celsius
// nach Grad Fahrenheit
// Fahrenheit = Celsius * (212 - 32)/100 + 32
//
#include <stdio.h>
#include <iostream.h>
int main(int nNumberOfArgs, char* pszArgs[])
<
    // Eingabe der Temperatur in Grad Celsius
    int nCelsius;
    cout << »Temperatur in Grad Celsius:<<;
    cin >> nCelsius;

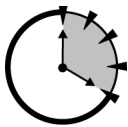
    // berechne Umrechnungsfaktor von Celsius
    // nach Fahrenheit
    int nFactor;
    nFactor = 212 - 32;

    // verwende Umrechnungsfaktor zur Konvertierung
    // von Celsius in Fahrenheit
    int nFahrenheit;
    nFahrenheit = nFactor * nCelsius/100 + 32;

    // Ausgabe des Ergebnisses
    cout << »Fahrenheit Wert ist:<<;
    cout << nFahrenheit;

    return 0;
>
```

Speichern Sie die Datei unter dem Namen *Conversion.cpp*. Das Standardverzeichnis ist eines der Verzeichnisse von Visual Studio. Ich bevorzuge es, in ein von mir selbst generiertes Verzeichnis zu wechseln, bevor ich die Datei speichere.



20 Min.

2.3 Erzeugen Ihres Programms

Wir haben in Sitzung 1 eine begrenzte Anzahl von Anweisungen verwendet, um den menschlichen Computer anzuweisen, einen Reifen zu wechseln. Obwohl sehr eingeschränkt, werden Sie vom Durchschnittsmenschen verstanden (zumindest von Deutsch Sprechenden).

Das Programm *Conversion.cpp*, das Sie gerade eingegeben haben, enthält C++-Anweisungen, eine Sprache, die Sie in keiner Tageszeitung finden werden. So kryptisch und grob diese C++-Anweisungen auch auf Sie wirken, versteht der Computer eine Sprache, die noch viel elementarer ist als C++. Die Sprache, die Ihr Computer versteht, wird als *Maschinensprache* bezeichnet.

Lektion 2 – Ihr erstes Programm in Visual C++

11

Der C++-Compiler übersetzt Ihr C++-Programm in die Maschinensprache Ihrer Mikroprozessor-CPU in Ihrem PC. Programme, die Sie von der Option »Programme« des Menüs »Start« aus aufrufen können, Visual C++ eingeschlossen, sind nichts anderes als Dateien, die Maschinenanweisungen enthalten.



Hinweis

Es ist möglich, Programme direkt in Maschinensprache zu schreiben. Dies ist aber viel schwieriger, als das gleiche Programm in C++ zu schreiben.

Die wichtigste Aufgabe von Visual C++ ist, Ihr C++-Programm in eine ausführbare Datei zu übersetzen. Der Vorgang der Übersetzung in eine ausführbare .EXE-Datei wird als *Erzeugen* bezeichnet. Dieser Prozess wird manchmal auch als *Kompilieren* bezeichnet (es gibt einen Unterschied dieser beiden Begriffe, der aber hier nicht relevant ist). Der Teil des C++-Paketes, der die Übersetzung des Programms ausführt, wird als *Compiler* bezeichnet.

Um Ihr Programm `Conversion.cpp` zu erzeugen, klicken Sie auf »Erstellen« im Menü »Erstellen«. (Nein, ich habe nicht gestottert.) Visual C++ antwortet darauf mit der Warnung, dass Sie noch keinen Arbeitsbereich angelegt haben, was immer das ist. Dies ist in Abbildung 2.2 zu sehen.

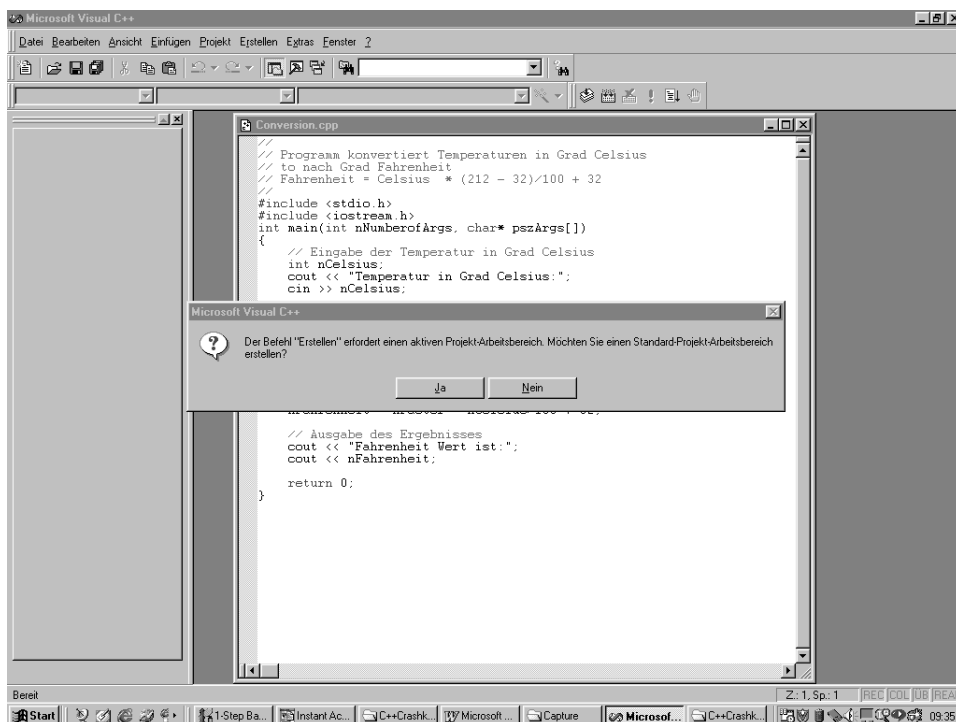


Abbildung 2.2: Ein Arbeitsbereich wird benötigt, bevor Visual C++ Ihr Programm erzeugen kann.

12 Freitagabend

Klicken Sie auf »Ja«, um eine Arbeitsbereichsdatei zu erzeugen und mit dem Erzeugungsprozess fortzufahren.



Die .cpp-Quelldatei ist nichts anderes, als eine Textdatei, ähnlich zu dem, was Sie etwa mit WordPad erzeugen würden. Der Arbeitsbereich Conversion.pwd, der von Visual C++ angelegt wird, ist eine Datei, in der Visual C++ spezielle Informationen über Ihr Programm speichern kann, Informationen, die in die Datei Conversion.cpp nicht hinein gehören.

Nach einigen Minuten Festplattenaktivität antwortet Visual C++ mit einem zufriedenen Klingelzeichen, das anzeigt, dass der Erzeugungsprozess abgeschlossen ist. Das Ausgabefenster sollte eine Meldung ähnlich zu Abbildung 2.3 enthalten, die anzeigt, dass die Datei Conversion.exe ohne Fehler und ohne Warnungen erzeugt wurde.

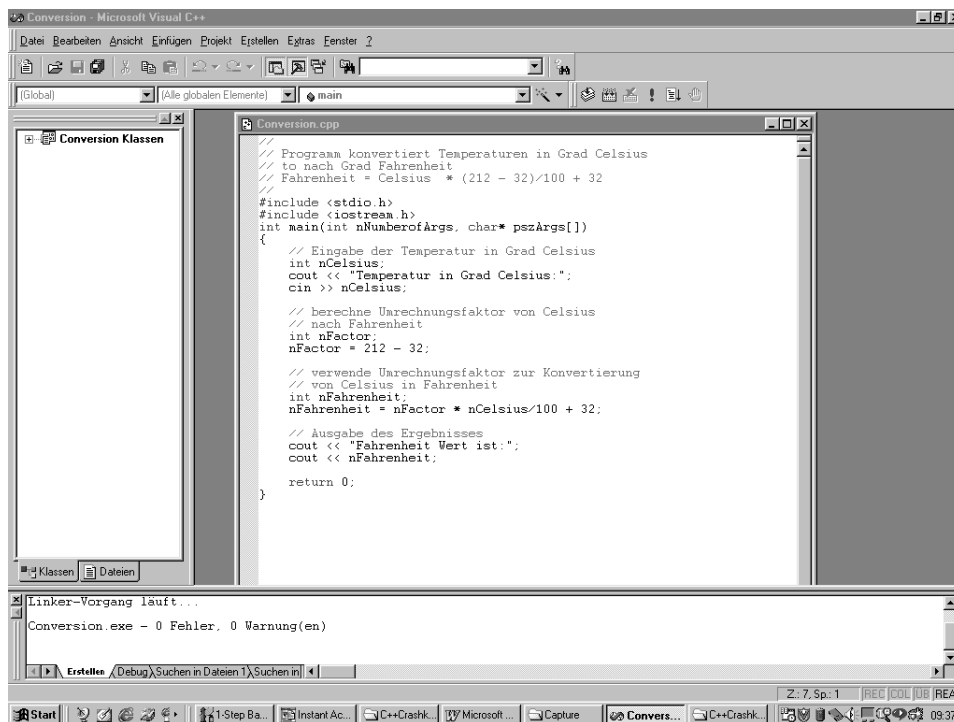


Abbildung 2.3: Keine Fehler und keine Warnungen – das Programm wurde erfolgreich erzeugt.

Visual C++ erzeugt einen unangenehmen Ton, wenn es während des Erzeugungsprozesses auf einen Fehler stößt (wenigstens denkt Microsoft, dass er unangenehm ist – ich habe ihn schon so oft gehört, dass er fast ein Teil von mir geworden ist). Zusätzlich enthält das Ausgabefenster eine Erklärung, welchen Fehler Visual C++ gefunden hat.

Ich habe ein Semikolon am Ende einer Zeile im Programm entfernt und habe das Programm neu kompiliert, nur um die Fehlermeldung zu demonstrieren. Das Ergebnis finden Sie in Abbildung 2.4.

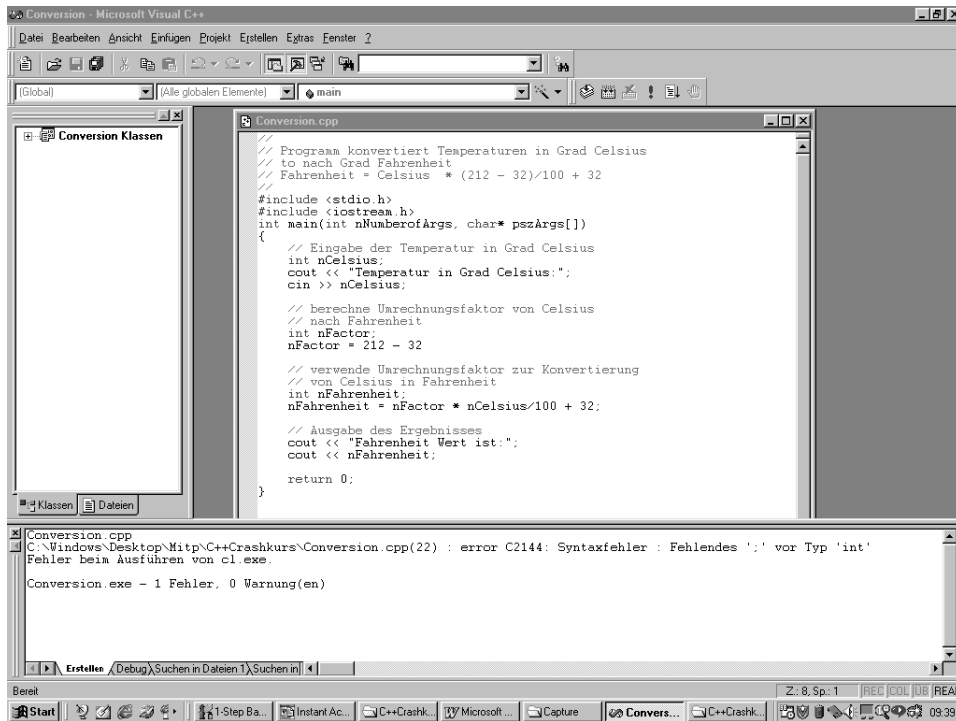


Abbildung 2.4: Visual C++ gibt während des Erzeugungsprozesses eine Fehlermeldung aus.

Die Fehlermeldung in Abbildung 2.4 ist tatsächlich sehr ausführlich. Sie beschreibt das Problem und den Ort des Fehlers (Zeile 22 in der Datei Conversion.cpp). Ich habe das Semikolon wieder eingefügt und das Programm neu kompiliert, um das Problem zu beheben.



Nicht alle Fehlermeldungen sind so klar wie diese. Oft kann ein einziger Fehler mehrere Fehlermeldungen erzeugen. Am Anfang können diese Fehlermeldungen verwirrend sein. Mit der Zeit bekommen Sie ein Gefühl dafür, was Visual C++ während des Erzeugungsprozesses denkt, und was Visual C++ verwirrt haben könnte.



Sie werden ohne Zweifel den unangenehmen Ton eines Fehlers hören, der von Visual C++ entdeckt wurden, bevor Sie das Programm Conversion.cpp fehlerfrei eingegeben haben. Wenn Sie es gar nicht schaffen, den Code so einzugeben, dass Visual C++ damit zufrieden ist, kopieren Sie die Datei Conversion.cpp aus wecc\Programs\lesson02\Conversion.cpp auf der beiliegenden CD-ROM.

C++-Fehlermeldungen

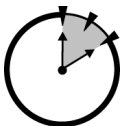
Warum sind alle C++-Pakete, Visual C++ eingeschlossen, so pingelig, wenn es um die Syntax von C++ geht? Wenn Visual C++ erkennt, dass ich ein Semikolon vergessen habe, warum kann es dieses Problem nicht einfach selber lösen und fortfahren?

Die Antwort ist einfach aber profund. Visual C++ denkt, dass Sie ein Semikolon vergessen haben. Ich könnte beliebig viele andere Fehler eingebaut haben, die Visual C++ als Fehlen eines Semikolons fehldiagnostiziert haben könnte. Wenn der Compiler einfach das Problem durch Einfügen eines Semikolons behebt, würde Visual C++ möglicherweise dadurch das eigentliche Problem verschleiern.

Wie Sie sehen werden, ist das Auffinden eines Fehlers in einem Programm, das ohne Probleme den Erzeugungsprozess durchläuft, schwierig und zeitaufwendig. Es ist besser, den Compiler Fehler finden zu lassen, wenn möglich.

Diese Lektion war hart zu Beginn. In den frühen Tagen des Computers versuchten Compiler alle möglichen Fehler zu erkennen und selber zu korrigieren. Dies hatte manchmal lächerliche Züge. Meine Freunde und ich machten uns einen Spaß daraus, einen »freundlichen« Compiler damit zu quälen, indem wir ein Programm eingaben, das nichts als die existenzielle Frage IF enthielt. (Rückschauend waren meine Freunde und ich ein wenig verrückt). Durch eine Reihe schmerzhafter Drehungen hat der besagte Compiler aus diesem einen Wort eine Kommandozeile generiert, die sich ohne Fehler übersetzen ließ. Ich weiß, dass der Compiler meine Absicht mit dem Wort IF missverstanden haben muss, weil ich nichts damit beabsichtigt hatte.

Meine Erfahrung ist, dass jedes Mal, wenn der Compiler versucht hat, ein Problem in einem Programm zu beheben, das Ergebnis falsch war. Trotz Fehlinformation war es keine Schwierigkeit, das Problem zu beheben, wenn der Compiler den Fehler gemeldet hat, bevor er ihn versuchte zu beheben. Compiler, die Fehler behoben haben, ohne entsprechende Fehlermeldungen auszugeben, haben mehr Schaden angerichtet als dass sie geholfen haben.



10 Min.

2.4 Ausführen Ihres Programms

Sie können das erfolgreich erzeugte Programm Conversion.exe durch Klicken auf das Icon »Ausführen« von Conversion.exe unter dem Menü »Erzeugen« ausführen. Alternativ können Sie Ctrl-F5 drücken.



Vermeiden Sie das Ausführen-Menü-Kommando oder die äquivalente F5-Taste fürs Erste.

Visual C++ öffnet ein Programmfenster ähnlich zu dem in Abbildung 2.5, das die Eingabe einer Temperatur in Grad Celsius erwartet.

Geben Sie eine Temperatur ein, z.B. 100 Grad Celsius. Nach Drücken der Enter-Taste gibt das Programm die äquivalente Temperatur in Grad Fahrenheit aus, wie in Abbildung 2.6 zu sehen ist. Die »Press any key to continue«-Meldung, die vielleicht hinter die Temperatúrausgabe gequetscht ist, ist ästhetisch nicht zufriedenstellend, aber die konvertierte Temperatur ist unmissverständlich – wir beheben das in Kapitel 5.



Abbildung 2.5: Das Programm Conversion.exe beginnt mit der Frage nach einer Temperatur.

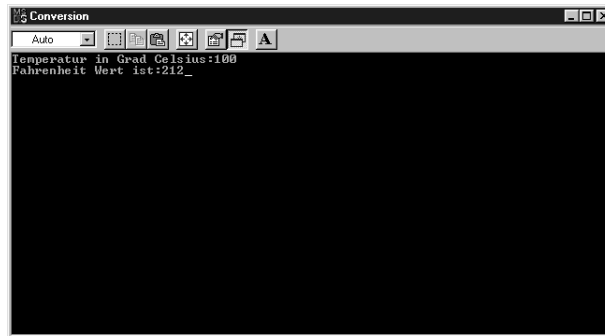


Abbildung 2.6: Das Programm Conversion.exe wartet auf eine Eingabe, nachdem das Programm beendet ist.



Die »Press any key to continue«-Meldung gibt dem Benutzer Zeit, die Ausgabe des Programms anzusehen, bevor das Fenster nach Beendigung des Programms geschlossen wird. Diese Meldung erscheint nicht, wenn Sie das Go-Kommando oder die F5-Taste verwenden.

Glückwunsch! Sie haben Ihr erstes Programm eingegeben, erzeugt und ausgeführt.

2.5 Abschluss

Es gibt noch zwei Punkte, die erwähnt werden sollten, bevor wir weitergehen. Zum einen könnte Sie die Ausgabe des Programms Conversion.exe überraschen. Zum anderen bietet Visual C++ viel mehr Hilfe an als nur Fehlermeldungen.

2.5.1 Programmausgabe

Windows-Programme haben eine visuell ausgerichtete, Fenster-basierte Ausgabe. Conversion.exe ist ein 32-Bit-Programm, das unter Windows ausgeführt wird, ist aber kein Windows-Programm im visuellen Sinne.



Wenn Sie nicht wissen, was die Phrase »32-Bit-Programm« bedeutet, brauchen Sie sich keine Sorgen zu machen.

Wie ich bereits in der Einleitung erläutert habe, ist dies kein Buch über das Schreiben von Windows-Programmen. Die C++-Programme, die Sie in diesem Buch schreiben, haben ein Kommandozeilen-Interface, das innerhalb einer DOS-Box ausgeführt wird. Angehende Windows-Programmierer sollten nicht verzweifeln – Sie haben Ihr Geld nicht umsonst ausgegeben. Das Erlernen von C++ ist Grundvoraussetzung für das Schreiben von Windows-Programmen mit C++.



0 Min.

2.5.2 Visual C++-Hilfe

Visual C++ bietet ein Hilfesystem an, das den C++-Programmierer signifikant unterstützt. Um zu sehen, wie diese Hilfe funktioniert, führen Sie einen Doppelklick auf dem Wort `#include` aus, bis es vollständig selektiert ist. Jetzt drücken Sie F1.

Visual C++ antwortet darauf mit dem Öffnen der MSDN-Bibliothek und der Anzeige einer ganzen Seite von Informationen über `#include`, wie in Abbildung 1.7 zu sehen ist (Sie verstehen vielleicht nicht alles, was da steht, sie werden es aber später verstehen).

Sie können die gleiche Information über die Auswahl von Index... unter dem Hilfe-Menü finden. Geben Sie `#include` im Indexfenster ein, was die gleichen Informationen liefert.

Wenn Sie als C++-Programmierer erfahrener werden, werden Sie sich mehr und mehr auf das Hilfesystem der MSDN-Bibliothek stützen.

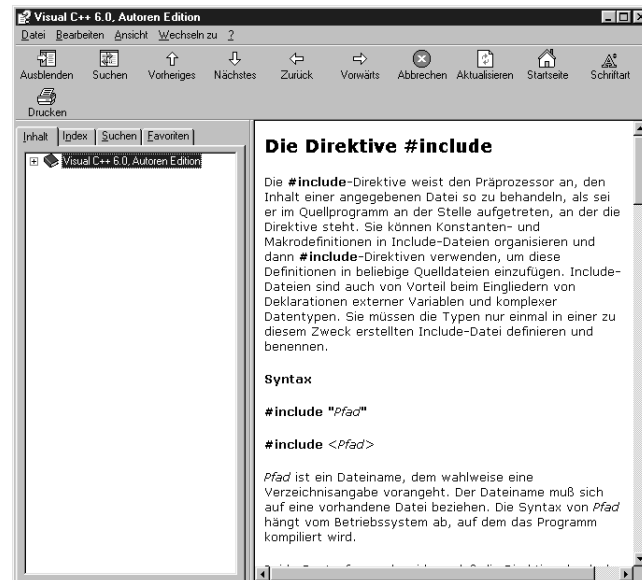


Abbildung 2.7: Die Taste F1 stellt dem C++-Programmierer Hilfe zur Verfügung.

Zusammenfassung

Visual C++ 6.0 hat eine benutzerfreundliche Umgebung, in der Sie Programme erzeugen und testen können. Sie verwenden den Editor von Visual C++, um Ihren Quellcode einzugeben. Einmal eingegeben, werden die C++-Anweisungen durch den Erzeugungsprozess in eine ausführbare Datei überführt. Schließlich können Sie Ihr fertiges Programm von Visual C++ aus ausführen.

Im nächsten Kapitel sehen Sie, wie Sie das gleiche Programm mit dem GNU C++-Compiler erzeugen können, den Sie auf der beiliegenden CD-ROM finden. Wenn Sie absolut überzeugt sind von Visual C++, können Sie zu Sitzung 4 springen, die erklärt, wie das Programm funktioniert, das Sie gerade eingegeben haben.

Selbsttest

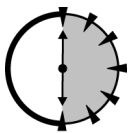
1. Was für eine Datei ist ein C++-Quellprogramm? (Ist es eine Word-Datei? Eine Excel-Datei? Eine Textdatei?) (Sehen Sie sich den ersten Abschnitt von »Ihr erstes Programm« an.)
2. Beachtet C++ die Einrückung? Achtet es auf Groß- und Kleinschreibung? (Siehe »Ihr erstes Programm«)
3. Was bedeutet »Erzeugen Ihres Programms«? (Siehe »Erzeugen Ihres Programms«)
4. Warum erzeugt C++ Fehlermeldungen? Warum versucht C++ nicht einfach, daraus schlau zu werden, was ich eingebe? (Siehe Kasten »C++-Fehlermeldungen«)



Ihr erstes C++-Programm mit GNU C++

Checkliste

- ☒ GNU C++ von der beiliegenden CD-ROM installieren
- ☒ Ihr erstes C++-Programm in GNU C++ schreiben
- ☒ Aus ihrem C++-Code ein ausführbares Programm erzeugen
- ☒ Ihr Programm ausführen
- ☒ Hilfe bei der Programmierung bekommen



30 Min.

Kapitel 2 behandelte das Schreiben, Erzeugen und Ausführen von C++-Programmen mit Visual C++. Viele Leser des *C++ Wochenende Crashkurses* haben keinen Zugang zu Visual C++. Für diese Leser enthält dieses Buch den frei verfügbaren GNU C++-Compiler, der auf der CD-ROM zu finden ist.



GNU wird »guh-new« gesprochen. GNU steht für die Ringdefinition »GNU is Not Unix«. Dieser Witz geht auf die Anfänge von C++ zurück. Nehmen Sie es einfach wie es ist. GNU ist eine Reihe von Werkzeugen der Free Software Foundation. Diese Werkzeuge stehen der Öffentlichkeit zur Verfügung, mit einigen Nutzungseinschränkungen, aber kostenlos.

Dieses Kapitel zeigt Schritt für Schritt, wie das gleiche Programm `Conversion.cpp` aus Kapitel 2 mit GNU C++ in ein ausführbares Programm verwandelt werden kann. Das Programm, um das es gehen soll, konvertiert eine Temperatureingabe von Grad Celsius nach Grad Fahrenheit.

3.1 Installation von GNU C++

Die diesem Buch beiliegende CD-ROM enthält die zum Zeitpunkt der Drucklegung neueste Version der GNU C++-Umgebung. Die Installationsanweisungen sind in Anhang C zu finden; diese Sitzung enthält Anweisungen zum Herunterladen und Installieren von GNU C++ vom Web aus.

Lektion 3 – Ihr erstes C++-Programm mit GNU C++**19**Teil 1 – Freitagabend
Lektion 3

Die GNU Umgebung wird von einer Reihe freiwilliger Programmierer gepflegt. Wenn Sie dies bevorzugen, können Sie die allerneueste Version von GNU C++ vom Web herunterladen.

Die GNU Entwicklungsumgebung ist ein sehr großes Paket. GNU enthält eine Anzahl von Hilfsprogrammen und anderen Programmiersprachen außer C++. GNU C++ selber unterstützt eine Vielzahl von Computerprozessoren und Betriebssystemen. Glücklicherweise müssen Sie nicht alles von GNU herunterladen, wenn Sie nur C++-Programme entwickeln wollen. Die GNU Entwicklungsumgebung ist auf verschiedene ZIP-Dateien aufgeteilt. Das Hilfsprogramm "ZIP-Picker", das auf der Website von Delorie-Software zu finden ist, teilt Ihnen mit, welche ZIP-Dateien Sie herunterladen müssen, basierend auf Ihren Antworten auf eine Reihe einfacherer Fragen.

So installieren Sie GNU C++ vom Web:

1. Gehen Sie zur Webseite <http://www.delorie.com/djgpp/zip-picker.html>.
2. Die Site zeigt Ihnen die Fragen, die wir im Folgenden wiedergeben. Beantworten Sie die Fragen des Programms wie fett gedruckt, um eine minimale Konfiguration zu erhalten.

FTP Site

Select a suitable FTP site: **Pick one for me**

Basic Functionality

Pick one of the following: **Build and run programs with DJGPP**

Which operating system will you be using? **<Ihr Betriebssystem>**

Do you want to be able to read the on-line documentation? **No**

Which programming languages will you be using? **<Klicken Sie C++>**

Integrated Development Environments and Tools

Which IDE(s) would you like? **<Klicken Sie auf RHIDE. Lassen Sie die emacs-options unausgewählt>**

Would you like gdb, the text mode GNU debugger? **No**

Extra Stuff

Please Check off each extra thing that you want. **Don't check anything in this list.**

3. Dann klicken Sie auf »Tell me what files I need«. Der ZIP-Picker antwortet mit einigen einfachen Installationsanweisungen und einer Liste von ZIP-Dateien, die Sie brauchen werden. Das Listing unten zeigt die Dateien, die für eine minimale Installation, wie sie hier beschrieben wird, benötigt werden – die Dateinamen, die Sie erhalten, können davon verschieden sein, weil sie eine aktuellere Version anzeigen.

Read the file README.1ST before you do anything else with DJGPP! It has important installation and usage instructions.

v2/djdev202.zip	DJGPP Basic Development Kit	1.4 mb
v2/faq211b.zip	Frequently Asked Questions	551 kb
v2apps/rhide14b.zip	RHIDE	1.6 mb
v2gnu/bnu281b.zip	Basic assembler, linker	1.8 mb
v2gnu/gcc2952b.zip	Basic GCC compiler	1.7 mb
v2gnu/gpp2952b.zip	C++ compiler	1.6 mb
v2gnu/lgpp295b.zip	C++ libraries	484 kb
v2gnu/mak377b.zip	Make (processes makefiles)	242 kb

Total bytes to download: 9,812,732

20

Freitagabend

4. Legen Sie ein Verzeichnis \DJGPP an.
5. Laden Sie in dieses Verzeichnis alle ZIP-Dateien, die Ihnen der ZIP-Picker aufgelistet hat, indem Sie auf den jeweiligen Dateinamen klicken.
6. Entpacken Sie die Dateien im Ordner DJGPP.
7. Fügen Sie die folgenden Kommandos in die Datei AUTOEXEC.BAT ein:

```
set PATH=C:\DJGPP\BIN;%PATH%
set DJGPP=C:\DJGPP\DJGPP.ENV
```

Beachten Sie, dass die obigen Zeilen davon ausgehen, dass ihr Ordner DJGPP direkt unterhalb von C:\ steht. Wenn Sie Ihren Ordner an einer anderen Stelle platziert haben, ersetzen Sie bitte den Pfad in den obigen Kommandos entsprechend.

8. Booten Sie neu, um die Installation abzuschließen.

Der Ordner \BIN enthält die ausführbaren Dateien der GNU-Werkzeuge. Die Datei DJGPP.ENV setzt eine Reihe von Optionen, um die GNU C++-»Umgebung« von Windows zu beschreiben.

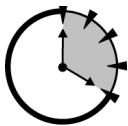


Bevor Sie GNU C++ benutzen, sehen Sie in der Datei DJGPP.ENV nach, ob der Support langer Dateinamen angeschaltet ist. Diese Option ausgeschaltet zu haben ist der meist begangene Fehler bei der Installation von GNU C++.

Öffnen Sie die Datei DJGPP.ENV mit einem Texteditor, z.B. mit Microsoft WordPad. Erschrecken Sie nicht, wenn Sie nur eine lange Zeichenkette sehen, die von kleinen schwarzen Kästchen unterbrochen ist. Unix verwendet ein anderes Zeichen für Zeilenumbrüche als Windows. Suchen Sie nach der Phrase "LFN=y" oder "LFN=Y" (Groß- oder Kleinschreibung spielen also keine Rolle). Wenn Sie stattdessen "LFN=n" finden (oder "LFN" überhaupt nicht vorkommt), ändern Sie das "n" in "y". Speichern Sie die Datei. (Stellen Sie sicher, dass Sie die Datei als Textdatei speichern und nicht in einem anderen Format, z.B. als Word .DOC-Datei.)

Fügen Sie die folgende Zeile in die Datei DJGPP.ENV im Block [rhide] ein, um rhide in deutscher Sprache zu verwenden:

```
LANGUAGE=DE
```



20 Min.

3.2 Ihr erstes Programm

Das Herz des GNU C++-Paketes ist ein Hilfsprogramm, das als `rhide` bekannt ist. Im Wesentlichen ist `rhide` ein Editor, der die verbleibenden Teile von GNU C++ zusammenbindet zu einer integrierten Umgebung, ähnlich wie Visual C++.

3.2.1 Eingabe des C++-Codes

Öffnen Sie ein MS-DOS-Fenster, indem Sie auf das MS-DOS-Icon unter dem Menü »Programme« klicken. Erzeugen Sie ein Verzeichnis, in dem Sie Ihr Programm erzeugen möchten. Ich habe ein Verzeichnis `c:\wecc\programs\lesson03\` erzeugt. In diesem Verzeichnis führen Sie das Kommando `rhide` aus.

Das rhide-Interface

Das Interface von `rhide` ist grundsätzlich anders als das von Windows-Programmen. Windows-Programme »zeichnen« ihre Ausgaben auf den Bildschirm, was ihnen ein feineres Aussehen verschafft.

Im Vergleich dazu arbeitet das Interface von `rhide` zeichenbasiert. `rhide` verwendet eine Reihe von Block-Zeichen, die auf dem PC zur Verfügung stehen, um ein Windows-Interface zu simulieren, was `rhide` weniger elegant aussehen lässt. Z.B. lässt `rhide` es nicht zu, das Fenster auf eine andere Größe zu bringen als die Standardeinstellung von 80x25 Zeichen, was Standard für MS-DOS-Programme ist.



Für die unter Ihnen, die alt genug sind, um sich daran zu erinnern, sieht das `rhide`-Interface sehr ähnlich aus wie das Interface der Borland-Suite von Programmierwerkzeugen, die es heute nicht mehr gibt.

Wie dem aus sei, das Interface von `rhide` funktioniert und gibt bequemen Zugriff auf die übrigen Werkzeuge von GNU C++.

Erzeugen Sie eine leere Datei, indem Sie »Neu« aus dem Datei-Menü auswählen. Geben Sie das Programm genau so ein, wie Sie es hier vorfinden.



Machen Sie sich keine Gedanken über das Einrücken – es kommt nicht darauf an, ob eine Zeile zwei oder drei Zeichen eingerückt ist oder ob zwei Worte durch ein oder zwei Leerzeichen getrennt sind.



Sie können natürlich mogeln und die Datei `Conversion.cpp` von der beiliegenden CD-ROM kopieren.

```
//
// Programm konvertiert Temperaturen in Grad Celsius
// nach Grad Fahrenheit
// Fahrenheit = Celsius * (212 - 32)/100 + 32
//
#include <stdio.h>
#include <iostream.h>
int main(int nNumberOfArgs, char* pszArgs[])
{
    // Eingabe der Temperatur in Grad Celsius
    int nCelsius;
    cout << »Temperatur in Grad Celsius:<<
    cin >> nCelsius;

    // berechne Umrechnungsfaktor von Celsius
    // nach Fahrenheit
    int nFactor;
```

22 Freitagabend

```

nFactor = 212 - 32;

// verwende Umrechnungsfaktor zur Konvertierung
// von Celsius in Fahrenheit
int nFahrenheit;
nFahrenheit = nFactor * nCelsius/100 + 32;

// Ausgabe des Ergebnisses
cout << »Fahrenheit Wert ist:<<
cout << nFahrenheit;

return 0;
}

```

Wenn Sie damit fertig sind, sollte Ihr rhide-Fenster wie in Abbildung 3.1 aussehen.

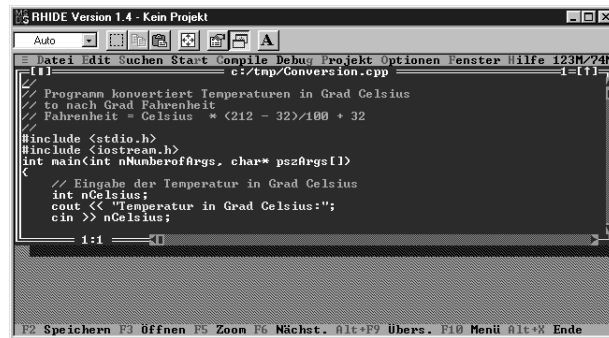


Abbildung 3.1: rhide stellt ein zeichenbasiertes Interface bereit, um C++-Programme zu erzeugen.

Wählen Sie »Speichern als...« im Datei-Menü aus, wie in Abbildung 3.2 zu sehen ist, und speichern Sie die Datei unter dem Namen Conversion.cpp.



Abbildung 3.2: Mit dem Kommando »Save As...« kann der Benutzer eine C++-Datei erzeugen.

3.3 Erzeugen Ihres Programms

Wir haben in Sitzung 1 eine begrenzte Anzahl von Anweisungen verwendet, um den menschlichen Computer anzuweisen, einen Reifen zu wechseln. Obwohl sehr eingeschränkt, werden Sie vom Durchschnittsmenschen verstanden (zumindest von Deutsch Sprechenden).

Das Programm `Conversion.cpp`, das Sie gerade eingegeben haben, enthält C++-Anweisungen, eine Sprache, die Sie in keiner Tageszeitung finden werden. So kryptisch und grob diese C++-Anweisungen auch auf Sie wirken, versteht der Computer eine Sprache, die noch viel elementarer ist als C++. Die Sprache, die Ihr Computer versteht, wird als Maschinensprache bezeichnet.

Der C++-Compiler übersetzt Ihr C++-Programm in die Maschinensprache Ihrer Mikroprozessor-CPU in Ihrem PC. Programme, die Sie unter Windows aufrufen können, GNU C++ eingeschlossen, sind nichts anderes als Dateien, die Maschinenanweisungen enthalten.



Hinweis

Es ist möglich, Programme direkt in Maschinensprache zu schreiben. Dies ist aber viel schwieriger, als das gleiche Programm in C++ zu schreiben.

Die wichtigste Aufgabe von GNU C++ ist, Ihr C++-Programm in eine ausführbare Datei zu übersetzen. Der Vorgang der Übersetzung in eine ausführbare .EXE-Datei wird als *Erzeugen* bezeichnet. Dieser Prozess wird manchmal auch als *Kompilieren* bezeichnet (es gibt einen Unterschied zwischen diesen beiden Begriffen, der aber hier nicht relevant ist). Der Teil des C++-Paketes, der die Übersetzung des Programms ausführt, wird als Compiler bezeichnet.

Um Ihr Programm `Conversion.cpp` zu erzeugen, klicken Sie auf »Compile« und dann auf »Make«, oder drücken Sie F9. `rhide` öffnet ein kleines Fenster am unteren Rand des Fensters, um den Fortschritt des Prozesses anzuzeigen. Wenn alles gut geht, erscheint die Meldung »Creating: Conversion.exe« gefolgt von »no errors« wie in Abbildung 3.3 zu sehen ist.

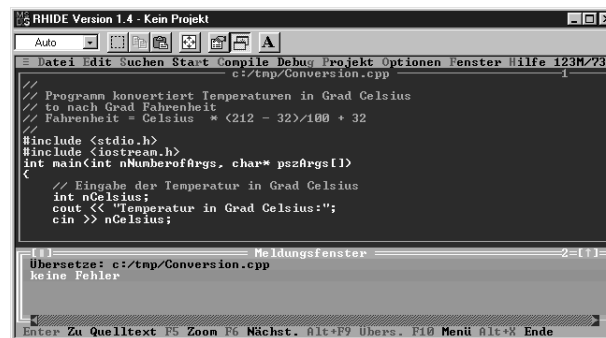


Abbildung 3.3: `rhide` zeigt »no errors« an, wenn kein Fehler aufgetreten ist.

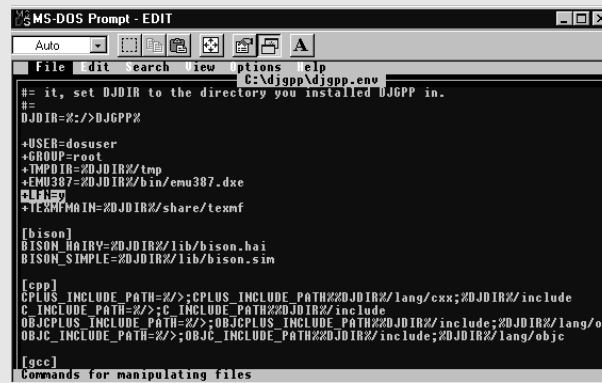
24 Freitagabend

GNU C++-Installationsfehler

Einige häufig gemachte Fehler bei der Installation können Ihnen den Spaß an Ihrer ersten Programmiererfahrung verderben.

Die Meldung »Bad command or file name« bedeutet, dass MS-DOS die Datei gcc.exe nicht finden kann, d.h. den GNU C++-Compiler. Entweder haben Sie GNU C++ nicht richtig installiert oder Ihr Pfad enthält nicht c:\djgpp\bin, wo gcc.exe zu finden ist. Versuchen Sie, GNU C++ erneut zu installieren und stellen Sie sicher, dass das Kommando SET PATH=c:\djgpp\bin;%PATH% in Ihrer Datei autoexec.bat vorkommt. Nachdem Sie GNU C++ erneut installiert haben, starten Sie den Rechner neu.

Die Meldung »gcc.exe: Conversion.cpp: No such file or directory (ENOENT)« zeigt an, dass gcc nicht weiß, dass Sie lange Dateinamen verwenden (im Gegensatz zu den alten MS-DOS 8.3 Dateinamen). Um dies zu korrigieren, editieren Sie die Datei c:\djgpp\djgpp.env. Setzen Sie die Eigenschaft LFN auf Y, wie in der Abbildung zu sehen.



```

MS-DOS Prompt - EDIT
File Edit Search View Options Help
C:\djgpp\djgpp.env
# = it, set DJDIR to the directory you installed DJGPP in.
# =
DJDIR=:~/DJGPPZ

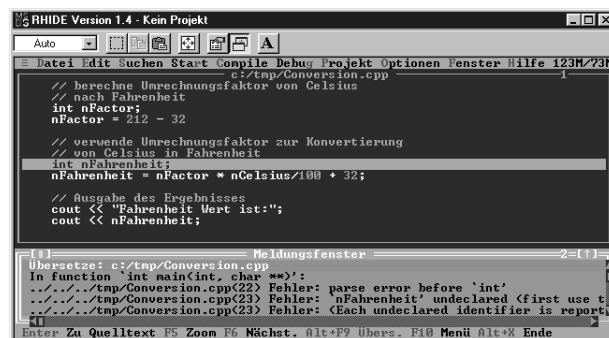
+USER=dosuser
+GROUP=root
+TMPDIR=~/DJDIR/tmp
+EMU387=~/DJDIR/bin/emu387.dxe
+LFN=Y
+TEXTMAIN=~/DJDIR/share/textf

[bison]
BISON_HAIRY=~/DJDIR/lib/bison.hai
BISON_SIMPLE=~/DJDIR/lib/bison.sim

[cpp]
INCLUDE_PATH=~/;CPLUS_INCLUDE_PATH~/lang/cxx;~/include
C_INCLUDE_PATH=~/;C_INCLUDE_PATH~/include
OBJCPLUS_INCLUDE_PATH=~/;OBJCPLUS_INCLUDE_PATH~/include;~/lang/o
OBJC_INCLUDE_PATH=~/;OBJC_INCLUDE_PATH~/include;~/lang/objc

[gcc]
Commands for manipulating files
  
```

GNU C++ erzeugt eine Fehlermeldung, wenn es einen Fehler in Ihrem C++-Programm findet. Um diesen Prozess des Fehlermeldens zu demonstrieren, habe ich ein Semikolon am Ende einer Zeile entfernt und das Programm neu kompiliert. Das Ergebnis finden Sie in Abbildung 3.4.



```

RHIDE Version 1.4 - Kein Projekt
Datei Edit Suchen Start Compile Debug Projekt Optionen Fenster Hilfe 123M/23M
c:/tmp/Conversion.cpp
// berechne Umrechnungsfaktor von Celsius
// nach Fahrenheit
int nFactor;
nFactor = 212 - 32

// verwende Umrechnungsfaktor zur Konvertierung
// von Celsius in Fahrenheit
int nFahrenheit;
nFahrenheit = nFactor * nCelsius/100 + 32;

// Ausgabe des Ergebnisses
cout << "Fahrenheit Wert ist:";
cout << nFahrenheit;

[ ] Messagesfenster
Übersetze: c:/tmp/Conversion.cpp
In function 'int main(int, char **)':
./../tmp/Conversion.cpp(22) Fehler: 'nFahrenheit' undeclared (first use t
./../tmp/Conversion.cpp(23) Fehler: 'Each undeclared identifier is report
  
```

Abbildung 3.4: GNU C++ gibt Fehlermeldungen während des Erzeugungsprozesses aus.

Die Fehlermeldung in Abbildung 3.4 ist ein wenig imposant; sie ist aber einigermaßen ausdrucksvoll, wenn Sie sie Zeile für Zeile betrachten.

Die erste Zeile zeigt an, dass der Fehler entdeckt wurde, während der Code innerhalb von `main( )` analysiert wurde, d.h. der Code zwischen der öffnenden und schließenden Klammer, die auf das Schlüsselwort `main( )` folgen.

Die zweite Zeile zeigt an, dass nicht verstanden wurde, wie `int` in Zeile 22 da passt. Natürlich passt `int` nicht, aber ohne das Semikolon hat GNU C++ gedacht, dass die Zeilen 18 und 22 eine einzige Anweisung sind. Die übrigen Fehler stammen daher, dass Zeile 22 nicht verstanden werden konnte.

Um das Problem zu beheben, habe ich zuerst Zeile 22 analysiert (beachten Sie die Zeile 22:5 unten links im Code-Fenster – der Cursor ist in Spalte 5 von Zeile 22). Da Zeile 22 in Ordnung zu sein scheint, gehe ich zu Zeile 18 zurück und stelle fest, dass ein Semikolon fehlt. Ich füge das Semikolon ein und kompiliere neu. Diesmal kompiliert GNU C++ ohne Schwierigkeiten.

C++ Fehlermeldungen

Warum sind alle C++-Pakete so pingelig, wenn es um die Syntax von C++ geht? GNU C++ war in der Lage, das Fehlen des Semikolons in obigem Beispiel zu erkennen. Wenn ein C++-Compiler erkennt, dass ich ein Semikolon vergessen habe, warum kann es dieses Problem nicht einfach selber lösen und fortfahren?

Die Antwort ist einfach aber profund. GNU C++ *denkt*, dass ich ein Semikolon vergessen habe. Ich könnte beliebig viele andere Fehler eingebaut haben, die GNU C++ als Fehlen eines Semikolons fehldiagnostiziert haben könnte. Wenn der Compiler einfach das Problem durch Einfügen eines Semikolons behebt, würde GNU C++ möglicherweise dadurch das eigentliche Problem verschleiern.

Wie Sie sehen werden, ist das Auffinden eines Fehlers in einem Programm, das ohne Probleme den Erzeugungsprozess durchläuft, schwierig und zeitaufwendig. Es ist besser, den Compiler Fehler finden zu lassen, wenn möglich.

Diese Lektion war hart zu Beginn. In den frühen Tagen des Computers versuchten Compiler, alle möglichen Fehler zu erkennen und selber zu korrigieren. Dies hatte manchmal lächerliche Züge. Meine Freunde und ich machten uns einen Spaß daraus, einen »freundlichen« Compiler damit zu quälen, indem wir ein Programm eingaben, das nichts als die existenzielle Frage IF enthielt. (Rückschauend waren meine Freunde und ich ein wenig verrückt). Durch eine Reihe schmerzhafter Drehungen hat der besagte Compiler aus diesem einen Wort eine Kommandozeile generiert, die sich ohne Fehler übersetzen ließ. Ich weiß, dass der Compiler meine Absicht mit dem Wort IF missverstanden haben muss, weil ich nichts damit beabsichtigt hatte.

Meine Erfahrung ist, dass jedes Mal, wenn der Compiler versucht hat, ein Problem in einem Programm zu beheben, das Ergebnis falsch war. Trotz Fehlinformation war es keine Schwierigkeit, das Problem zu beheben, wenn der Compiler den Fehler gemeldet hat, bevor er ihn versuchte zu beheben. Compiler, die Fehler behoben haben, ohne entsprechende Fehlermeldungen auszugeben, haben mehr Schaden angerichtet als genützt.

26

Freitagabend



10 Min.

3.4 Ausführen Ihres Programms

Um das Programm Conversion auszuführen, klicken Sie auf Start und wieder Start, oder drücken Sie Ctrl+F9 wie in Abbildung 3.5 zu sehen ist.

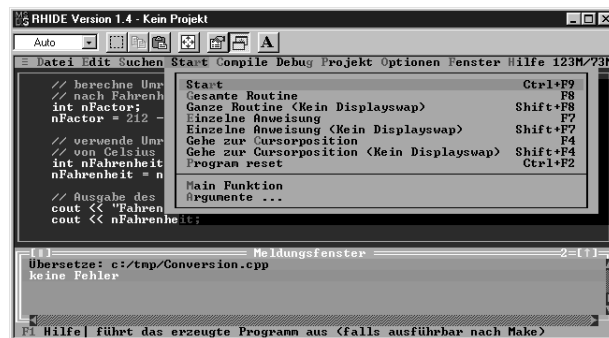


Abbildung 3.5: rhide öffnet ein Fenster, in dem das Programm ausgeführt wird.

Sofort erscheint ein Fenster, in dem das Programm die Eingabe einer Temperatur in Grad Celsius erwartet, wie in Abbildung 3.6.



Abbildung 3.6: Die Temperatur wird in Fahrenheit angezeigt.

Geben Sie eine bekannte Temperatur ein, z.B. 100 Grad. Nach Drücken der Return-Taste gibt das Programm die äquivalente Temperatur von 212 Grad Fahrenheit zurück. Weil rhide das Fenster des Programms sofort schließt, wenn das Programm beendet ist, haben Sie keine Chance, den Inhalt des Fensters zu lesen, bevor es geschlossen wird. rhide öffnet einen Dialog mit der Meldung, dass das Programm mit einem Fehlercode von Null beendet wurde. Abgesehen von der Bezeichnung »Fehlercode« bedeutet Null, dass kein Fehler aufgetreten ist.

Um die Ausgabe des bereits beendeten Programms einzusehen, klicken Sie auf den Menüpunkt »Nutzerbildschirm« des Fenster-Menüs oder drücken Sie Alt+F5. Dieses Fenster zeigt das aktuelle

MS-DOS-Fenster. In diesem Fenster sehen Sie die letzten 25 Zeilen der Ausgabe Ihres Programms, die auch die Ausgabe der berechneten Temperatur in Fahrenheit enthält, wie in Abbildung 3.6 zu sehen ist.

Glückwunsch! Sie haben Ihr erstes Programm mit GNU C++ eingegeben, erzeugt und ausgeführt.

3.5 Abschluss

Es gibt zwei Punkte, auf die hingewiesen werden sollte. Erstens ist GNU C++ nicht dafür gedacht, Windows-Programme zu schreiben. Theoretisch könnten Sie mit GNU C++ ein Windows-Programm schreiben, das wäre aber nicht einfach, ohne die Bibliotheken von Visual C++ zu verwenden. Zweitens bietet GNU C++ eine Art Hilfe an, die sehr nützlich sein kann.

3.5.1 Programmausgabe

Windows-Programme haben eine sehr visuell ausgerichtete, Fenster-basierte Ausgabe. Conversion.exe ist ein 32-Bit-Programm, das unter Windows ausgeführt wird, aber kein Windows-Programm im eigentlichen Sinne ist.



Wenn Sie nicht wissen, was die Phrase »32-Bit-Programm« bedeutet, brauchen Sie sich keine Sorgen zu machen.

Tipp

Wie ich bereits in der Einleitung erläutert habe, ist dies kein Buch über das Schreiben von Windows-Programmen. Die C++-Programme, die Sie in diesem Buch schreiben, haben ein Kommandozeilen-Interface, das innerhalb einer DOS-Box ausgeführt wird. Angehende Windows-Programmierer sollten nicht verzweifeln – Sie haben Ihr Geld nicht umsonst ausgegeben. Das Erlernen von C++ ist Grundvoraussetzung für das Schreiben von Windows-Programmen in C++.



0 Min.

3.5.2 GNU C++ Hilfe

GNU C++ stellt über das Benutzerinterface von `rhide` ein Hilfesystem bereit. Platzieren Sie den Cursor auf ein Konstrukt, das Sie nicht verstehen, und drücken Sie F1; ein Fenster erscheint wie in Abbildung 3.7. Alternativ können Sie auch »Index« im Hilfe-Menü auswählen, um sich eine Liste von Hilfefunkten anzeigen zu lassen. Klicken Sie auf den Hilfefunkt, den Sie angezeigt haben möchten.

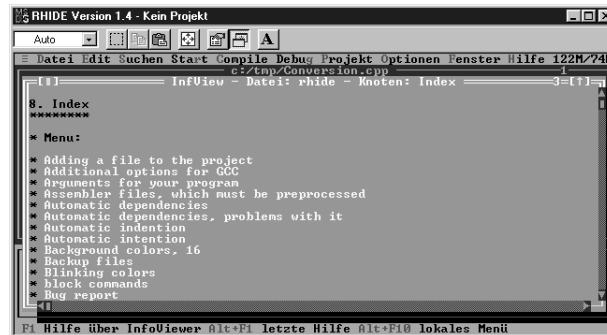


Abbildung 3.7: rhide stellt Hilfe über F1 und einen Index bereit.



Die Hilfe von GNU C++ ist nicht so umfangreich, wie die von Visual C++. Wenn Sie z.B. mit dem Cursor auf `int` gehen und F1 drücken, erscheint ein Fenster, das den Editor beschreibt. Nicht gerade das, was ich haben wollte. Die Hilfe von GNU C++ konzentriert sich auf Bibliotheksfunktionen und Compileroptionen. Glücklicherweise ist die Hilfe von GNU C++ in den meisten Fällen ausreichend, wenn Sie erst einmal C++ beherrschen.

Zusammenfassung

GNU C++ stellt eine benutzerfreundliche Umgebung bereit, in der Sie Programme mit Hilfe des Hilfsprogramms `rhide` erzeugen und testen können. Sie können `rhide` in ähnlicher Weise wie Visual C++ benutzen. Sie können den Editor von `rhide` verwenden, um den Code einzugeben, und den `rhide`-Erzeuger, um den Quelltext in Maschinencode zu überführen. Schließlich ermöglicht es `rhide`, das fertige Programm in derselben Umgebung laufen zu lassen.

Das nächste Kapitel geht Schritt für Schritt durch das C++-Programm.

Selbsttest

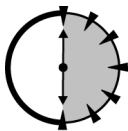
1. Was für eine Datei ist ein C++-Quellprogramm? (Ist es eine Word-Datei? Eine Excel-Datei? Eine Textdatei?) (Sehen Sie sich den ersten Abschnitt von »Ihr erstes Programm« an)
2. Beachtet C++ das Einrücken? Achtet es auf Groß- und Kleinschreibung? (Siehe »Ihr erstes Programm«)
3. Was bedeutet »Erzeugen Ihres Programms«? (Siehe »Erzeugen Ihres Programms«)
4. Warum erzeugt C++ Fehlermeldungen? Warum versucht C++ nicht einfach, daraus schlau zu werden, was ich eingebe? (Siehe Kasten »C++-Fehlermeldungen«)

C++-Instruktionen



Checkliste

- ☒ Programm »Conversion« aus Sitzungen 2 und 3 erneut betrachten
- ☒ Die Teile eines C++-Programms verstehen
- ☒ Häufig verwendete C++-Kommandos einführen



30 Min.

In den Sitzungen 2 und 3 haben Sie ein C++-Programm eingegeben. Die Idee dahinter war, die C++-Umgebung kennen zu lernen (welche Umgebung auch immer Sie gewählt haben) und weniger, das Programmieren dabei zu erlernen. In dieser Sitzung wird das Programm `Conversion.cpp` genauer analysiert. Sie werden sehen, was genau jeder Teil des Programms tut, und wie jeder Programmteil seinen Beitrag zur Lösung leistet.

4.1 Das Programm

Listing 4-1 ist (wieder) das Programm `Conversion.cpp`, außer, dass es nun mit Kommentaren versehen ist, die wir im Rest der Sitzung behandeln wollen.



Es gibt mehrere Aspekte des Programms, die Sie erst einmal glauben müssen. Seien Sie geduldig. Jede Struktur des Programms wird zu ihrer Zeit erklärt werden.

Diese Version hat Extra-Kommentare ...

Listing 4-1

```
//  
// Conversion konvertiert Temperaturen  
// in Grad Celsius nach Grad Fahrenheit:  
// Fahrenheit = Celsius * (212 - 32)/100 + 32  
//  
#include <stdio.h>           // Rahmen  
#include <iostream.h>  
int main(int nNumberOfArgs, char* pszArgs[])  
{  
    // hier ist unsere erste Anweisung -  
    // es ist eine Deklaration  
    int nCelsius;  
  
    // unsere ersten I/O-Anweisungen  
    cout << »Temperatur in Grad Celsius:<<;  
    cin >> nCelsius;  
  
    // die Zuweisung enthält eine Berechnung  
    int nFactor;  
    nFactor = 212 - 32;  
  
    // eine Zuweisung, die einen Ausdruck enthält,  
    // in dem eine Variable vorkommt  
    int nFahrenheit;  
    nFahrenheit = nFactor * nCelsius/100 + 32;  
  
    // Ausgabe des Ergebnisses  
    cout << »Fahrenheit Wert ist:<<;  
    cout << nFahrenheit;  
  
    return 0;  
}
```

4.2 Das C++-Programm erklärt

Unser menschliches Programm in Sitzung 1 bestand aus einer Folge von Anweisungen. In gleicher Weise besteht ein C++-Programm aus einer Folge von C++-Anweisungen, die der Computer in dieser Reihenfolge verarbeitet. Diese Anweisungen lassen sich in eine Reihe von Typen einteilen. Jeder Typ wird hier beschrieben.

4.2.1 Der grundlegende Programmaufbau

Jedes Programm, das in C++ geschrieben wird, beginnt mit dem gleichen grundlegenden Aufbau:

```
#include <stdio.h>  
#include <iostream.h>  
int main(int nNumberOfArgs, char* pzArgs[])  
{  
    ... Ihr Code steht hier ...  
    return 0;  
}
```

Sie brauchen sich über die Details dieses Aufbaus keine Gedanken zu machen – die Details werden später behandelt – aber Sie sollten so einen ersten Eindruck haben, wie sie aussehen. Die ersten beiden Zeilen werden **Include-Anweisungen** genannt, weil Sie dafür sorgen, dass der Inhalt der bezeichneten Datei an diesem Punkt in das Programm eingefügt wird. Wir verstehen sie zu diesem Zeitpunkt einfach als Magie.

Die nächste Anweisung in dem Programmrahmen ist die Anweisung `int main(...)`. Diese wird gefolgt von einer öffnenden und einer schließenden Klammer. Ihr Programm steht zwischen diesen beiden Klammern. Die Ausführung des Programms beginnt nach der öffnenden Klammer und endet bei der `return`-Anweisung, die unmittelbar vor der schließenden Klammer steht.

Unglücklicherweise müssen wir eine detailliertere Beschreibung des Programmrahmens auf spätere Kapitel verschieben. Machen Sie sich keine Sorgen ... wir kommen noch dazu, bevor das Wochenende vorbei ist.

4.2.2 Kommentare

Die ersten Programmzeilen scheinen frei formuliert zu sein. Entweder ist dieser »Code« für das menschliche Auge gedacht, oder der Computer ist doch schlauer, als wir immer gedacht haben. Die ersten sechs Zeilen werden als Kommentare bezeichnet. Ein **Kommentar** ist eine Zeile oder ein Teil einer Zeile, die vom C++-Compiler ignoriert wird. Kommentare ermöglichen es dem Programmierer, zu erklären, was er oder sie beim Schreiben des Codes gedacht hat.

Ein C++-Kommentar beginnt mit einem Doppelslash (`»//«`) und endet mit einer neuen Zeile. Sie können beliebige Zeichen in Ihren Kommentaren verwenden. Kommentare können so lang sein wie Sie wollen, aber es ist übersichtlicher, wenn sie nicht länger als ca. 80 Zeichen werden; das ist die Länge, die vom Bildschirm dargestellt werden kann.

Ein **Zeilenumbruch** würde in den frühen Tagen der Schreibmaschine als »Carriage Return« bekannt geworden sein, als der Vorgang der Zeicheneingabe in eine Maschine als »Tippen« und nicht als »Keyboarding« bezeichnet wurde. Ein Zeilenumbruch ist das Zeichen, das eine Kommandozeile beendet.

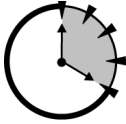
C++ erlaubt eine zweite Art Kommentar, in der alles nach `/*` und vor dem nächsten `*/` ignoriert wird; diese Art des Kommentars wird in C++ normalerweise nicht mehr verwendet.



Es kommt Ihnen vielleicht komisch vor, dass es in C++ oder einer anderen Programmiersprache ein Kommando gibt, das vom Computer ignoriert wird. Jede Programmiersprache hat so etwas in irgendeiner Form. Es ist wichtig, dass der Programmierer erklärt, was ihm oder ihr durch den Kopf gegangen ist, als der Code geschrieben wurde. Für jemanden, der das Programm nimmt, um es zu benutzen oder zu modifizieren, muss das sonst nicht offensichtlich sein. In der Tat kann die Idee hinter dem Programm selbst für den Programmierer nach einigen Monaten nicht mehr offensichtlich sein.



Verwenden Sie Kommentare früh und oft.



20 Min.

4.2.3 Noch mal der Rahmen

Die nächsten vier Zeilen beschreiben den Programmrahmen, den ich schon früher einmal erwähnt habe. Erinnern Sie sich daran, dass die Programmausführung bei der ersten Anweisung nach der öffnenden Klammer beginnt.

4.2.4 Anweisungen

Die erste Zeile, die kein Kommentar ist, ist eine C++-Anweisung. Eine **Anweisung** ist eine einzelne Menge von Kommandos. Alle Anweisungen, die keine Kommentare sind, enden mit einem Semikolon (;). (Es gibt einen Grund, weshalb Kommentare dies nicht tun, aber er ist obskur. Meiner Meinung nach sollten auch Kommentare mit einem Semikolon enden und wenn es nur wegen der Konsistenz ist.)

Wenn Sie sich das Programm ansehen, werden Sie bemerken, dass es Leerzeichen, Tabulatorzeichen und Zeilenumbrüche enthält. Und tatsächlich habe ich jeder Anweisung im Programm einen Zeilenumbruch folgen lassen. Diese Zeichen werden unter dem Sammelbegriff **Leerraum** zusammengefasst, weil Sie keines dieser Zeichen auf dem Bildschirm sehen können. Ein **Leerraum** ist ein Leerzeichen, ein Tabulator, ein vertikaler Tabulator oder ein Zeilenumbruch. C++ ignoriert Leerraum.



Tipp

Sie können Leerraum an jeder beliebigen Stelle in Ihrem Programmtext einfügen, um die Lesbarkeit zu erhöhen, außer innerhalb von Worten.

Während C++ Leerraum ignoriert, unterscheidet es Groß- und Kleinschreibung. Die Variablen `fullspeed` und `FullSpeed` haben nichts miteinander zu tun. Während das Kommando `int` verstanden wird, hat C++ keine Ahnung, was `INT` bedeuten soll.

4.2.5 Deklarationen

Die Zeile `int nCelsius;` ist eine Deklarationsanweisung. Eine **Deklaration** ist eine Anweisung, die eine Variable definiert. Eine **Variable** ist ein Platzhalter für Werte eines bestimmten Typs. Eine Variable enthält einen Wert, wie eine Zahl oder ein Zeichen.

Der Begriff *Variable* kommt von algebraischen Gleichungen der Form:

```
x = 10
y = 3 * x
```

Im zweiten Ausdruck, ist `y` gleich 3 mal `x`, aber was ist `x`? Die Variable `x` fungiert als Platzhalter für einen Wert. In diesem Fall ist der Wert von `x` gleich 10, aber wir hätten den Wert von `x` ebenso gut auf 20, 30 oder -1 setzen können. Die zweite Formel macht immer Sinn, unabhängig vom Wert von `x`.

In der Algebra ist es erlaubt, mit einer Anweisung wie `x = 10` zu beginnen. In C++ muss der Programmierer eine Variable erst definieren, bevor er sie benutzen kann.

In C++ hat jede Variable einen Typ und einen Namen. Die Zeile `int nCelsius;` deklariert eine Variable `nCelsius`, die einen Integerwert aufnehmen kann. (Warum haben sie es nicht einfach

`integer` anstatt `int` genannt? Ich weiß es nicht. Das ist einfach eines der Dinge, mit denen Sie zu leben lernen müssen.)

Der Name einer Variable hat keine besondere Bedeutung für C++. Eine Variable muss mit einem Buchstaben beginnen ('A' bis 'Z' oder 'a' bis 'z'). Alle weiteren Zeichen müssen entweder ein Buchstabe, eine Ziffer ('0' bis '9') oder ein Unterstrich ('_') sein. Variablennamen können so lang sein, wie es für Sie Sinn macht.



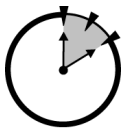
Es gibt natürlich eine Beschränkung, aber die ist viel größer als die Grenze des Lesers. Gehen Sie nicht über eine Länge hinaus, die sich der Leser nicht bequem behalten kann, sagen wir 20 Zeichen.



Nach Konvention beginnen Variablen mit einem kleinen Buchstaben. Jedes neue Wort in einer Variablen beginnt mit einem Großbuchstaben wie in der Variable `myVariable`. Ich erkläre die Bedeutung des `n` in `nCelsius` in Sitzung 5.



Versuchen Sie, Variablennamen kurz, aber aussagekräftig zu wählen. Vermeiden Sie Namen wie `x`, weil `x` keine Bedeutung hat. Eine Variable mit dem Namen `lengthOfLineSegment` ist viel aussagekräftiger.



10 Min.

4.2.6 Eingabe/Ausgabe

Die Zeilen, die mit `cin` und `cout` beginnen, werden als Eingabe-/Ausgabe-Anweisungen bezeichnet, oder kurz I/O-Anweisungen (von Input/Output). (Wie alle Ingenieure lieben Programmierer Abkürzungen und Kürzel.)

Die erste I/O-Anweisung gibt die Phrase »Temperatur in Grad Celsius:« auf `cout` (gesprochen »see-out«) aus. `cout` ist der Name des Standard-Ausgabe-Device von C++. In diesem Fall ist der Monitor das Ausgabe-Device.

Die nächste Zeile ist genau das Gegenteil. Die Zeile besagt, dass ein Wert aus dem Eingabe-Device von C++ bezogen, und in der Variable `nCelsius` gespeichert werden soll. Das Standard-Eingabe-Device von C++ ist die Tastatur. Das ist das C++-Analogon zu der algebraischen Formel $x=10$, die oben erwähnt wurde. Im Rest des Programms ist der Wert von `nCelsius` so, wie der Benutzer ihn hier eingegeben hat.

4.2.7 Ausdrücke

In den nächsten beiden Zeilen, die als Berechnungsausdrücke gekennzeichnet sind, deklariert das Programm eine Variable `nFactor`, und weist ihr den Ergebniswert einer Rechnung zu. Die Rechnung berechnet die Differenz von 212 und 32. In C++ wird eine solche Formel als Ausdruck bezeichnet.

Ein **Operator** ist ein Kommando, das einen Wert generiert. Der Operator in dieser Berechnung ist »-«.

Ein **Ausdruck** ist ein Kommando, das einen Wert hat. Der Ausdruck ist hier »212 – 32«.

4.2.8 Zuweisung

Die gesprochene Sprache kann mehrdeutig sein. Der Begriff *gleich* ist eine dieser Mehrdeutigkeiten.

Das Wort *gleich* kann bedeuten, dass zwei Dinge den gleichen Wert haben, wie etwa 100 Cents gleich einem Dollar sind. Gleich kann auch wie in der Mathematik eine Zuweisung bedeuten, wie etwa $y \text{ gleich } 3 \text{ mal } x$.

Um Mehrdeutigkeiten zu vermeiden, rufen C++-Programmierer den **Zuweisungsoperator** = auf. Der Zuweisungsoperator speichert den Wert auf der rechten Seite in der Variablen auf der linken Seite. Programmierer sagen, dass `nFactor` der Wert 212 – 32 zugewiesen wird.



0 Min.

4.2.9 Ausdrücke (Fortsetzung)

Der zweite Ausdruck im `Conversion.cpp` ist ein wenig komplexer als der erste Ausdruck. Dieser Ausdruck benutzt einige mathematische Symbole: * für die Multiplikation, / für Division und + für Addition. In diesem Fall wird die Berechnung auf Variablen, und nicht auf einfachen Konstanten ausgeführt.

Der Wert in der Variable `nFactor` (der unmittelbar vorher berechnet wurde) wird mit dem Wert in `nCelsius` multipliziert (der gleich der Tastatureingabe ist). Das Ergebnis wird durch 100 geteilt und um 32 erhöht. Das Ergebnis des gesamten Ausdrucks wird der Integervariablen `nFahrenheit` zugewiesen.

Die letzten beiden Kommandos geben den Text »Fahrenheit Wert ist:« auf dem Bildschirm aus, gefolgt vom Wert von `nFahrenheit`.

Zusammenfassung

Sie haben schließlich eine Erklärung des Programms `Conversion` gesehen, das Sie in den Sitzungen 2 und 3 eingegeben haben. Notwendigerweise waren diese Erklärungen auf einem hohen Abstraktionsniveau. Die Details kommen später.

- Alle Programme beginnen mit demselben Rahmen.
- C++ erlaubt Ihnen, Kommentare einzubinden, die Ihnen und anderen erklären, was die einzelnen Programmteile tun.
- C++-Ausdrücke sehen aus wie algebraische Ausdrücke, mit dem Unterschied, dass C++-Variablen deklariert sein müssen, bevor sie benutzt werden.
- = wird Zuweisung genannt.
- Eingabe- und Ausgabeanweisungen beziehen sich defaultmäßig in C++ auf die Tastatur bzw. den Bildschirm oder das MS-DOS-Fenster.

Selbsttest

1. Was tut die folgende C++-Anweisung (Siehe »Kommentare«)
`// Ich habe mich verlaufen`
2. Was tut die folgende C++-Anweisung (Siehe »Deklarationen«)
`int nQuizYourself; // hilf mir hier raus`
3. Was tut die folgende C++-Anweisung (Siehe »Eingabe/Ausgabe«)
`cout << »Hilf mir hier raus«`
4. Was tut die folgende C++-Anweisung (Siehe »Ausdrücke«)
`nHelpMeOutHere = 32;`



Freitagabend – Zusammenfassung

1. *Schreiben Sie ein Programm, das ein Auto herunterlässt, unter Verwendung dieser Objekte:*

```
Auto  
Reifen  
Radmutter  
Wagenheber  
Schraubenschlüssel
```

Lassen Sie uns weiterhin annehmen, dass unser Prozessor die folgenden Aktionen versteht:

```
greifen  
bewegen, nach oben bewegen, nach unten bewegen  
loslassen  
drehen
```

Hinweise:

- a. *Sie müssen annehmen, dass die Prozessorperson hoch und runter versteht und dass ein Wagenheber einen Griff hat.*
- b. *Nicht alle zur Verfügung stehenden Substantive und Verben werden verwendet.*
2. *Entfernen Sie das Minuszeichen zwischen 212 und 32 und erzeugen Sie das Programm neu. Zeichnen Sie die Fehlermeldung auf, und erklären Sie sie.*
3. *Beheben Sie das »Problem« so, wie es von Visual C++/GNU C++ vorgeschlagen wird, und erzeugen Sie Ihr Programm neu.*
4. *Führen Sie das Programm aus, und geben Sie einen bekannten Wert ein, wie etwa 100 Grad Celsius. Zeichnen Sie das Ergebnis auf.*
5. *Erklären Sie das resultierende Verhalten.*
6. *Erklären Sie, warum dies eine sehr unglückliche Situation wäre, wenn Visual C++/GNU C++ selbstkorrigierende Compiler wären.*

Hinweis: Glauben Sie es oder nicht, »32;« ist ein gültiges Kommando.

7. Entfernen Sie die schließenden Hochkommata in Zeile 28 von *Conversion.cpp* innerhalb von *rhide*, so dass die Zeile so aussieht:

```
cout << »Fahrenheit Wert ist:;
```

Erzeugen Sie das Programm, und achten Sie auf Fehlermeldungen.

8. Können Sie die Fehlermeldungen erklären, die Sie sehen?
9. Wenn GNU C++ versuchen würde, das Problem automatisch zu beheben, was würde es tun? Probieren Sie diese Lösung aus, und erzeugen Sie das Programm neu. Achten Sie auf das Ergebnis.
10. Führen Sie das »berichtigte« Programm aus. Geben Sie eine Temperatur von 100 Grad Celsius ein, und achten Sie auf das Ergebnis.
11. Denken Sie einen Augenblick darüber nach, was passiert wäre, wenn GNU C++ seine eigene Lösung angewendet hätte. Hilft ein solcher Zugang? Ist er schädlich?
12. Irgendwelche Kommentare?

Hinweise:

- a. GNU C++ denkt, dass eine Zeichenkette bei einem Hochkomma beginnt und bei dem nächsten Hochkomma endet.
- b. Der *rhide*-Editor hebt Worte hervor in Abhängigkeit davon, als was er die Worte interpretiert. Zeichenketten erscheinen hellblau.
- c. Wenn Sie verwirrt sind, gehen Sie zur Antwort von Frage 9 (siehe Anhang A). Wenn Sie diesen Hinweis verstanden haben, dann gehen sie zurück und beantworten alle diese Fragen.

13. Welche der folgenden Namen sind gültige Variablennamen?

- a. twoFeetOfRope**
- b. 2FeetOfRope**
- c. two Feet Of Rope**
- d. engthOf2Ropes**
- e. &moreRope**

14. Schreiben Sie ein Programm, das den Benutzer auffordert, drei Zahlen einzugeben, und dann die Summe dieser Zahlen ausgibt. Zusatz: Geben Sie das Mittel der eingegebenen Zahlen aus anstelle ihrer Summe.

Hinweise:

- a. Beginnen Sie mit dem Standardrahmen von C++.**
- b. Vergessen Sie nicht, Ihre Variablen zu deklarieren.**
- c. Vergessen Sie nicht, dass Division eine höhere Priorität hat als Addition. Es könnte sein, dass Sie Klammern verwenden müssen, insbesondere im Zusatzprogramm.**

☐ Freitag

☒ **Samstag**

☐ Sonntag

Samstagmorgen

Teil 2

Lektion 5

Variablentypen

Lektion 6

Mathematische Operationen

Lektion 7

Logische Operationen

Lektion 8

Kommandos zur Flusskontrolle

Lektion 9

Funktionen

Lektion 10

Debuggen

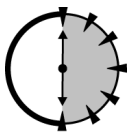


Variablentypen



Checkliste

- ☒ Variablen deklarieren
- ☒ Mit den Grenzen von Integervariablen umgehen
- ☒ float-Variablen verwenden
- ☒ Deklaration und Benutzung anderer Variablentypen



30 Min.

Ein Problem des Programms Conversion in Teil I ist, dass es nur mit Integerwerten arbeitet. Das ist kein Problem im alltäglichen Leben – es ist unwahrscheinlich, dass jemand eine Temperatur wie z.B. 10.5 eingeben wird. Ein schlimmeres Problem sind die Rundungsfehler. Die meisten Integerwerte in Celsius werden auf einen nicht ganzzahligen Wert in Fahrenheit abgebildet. Das Programm Conversion kümmert sich nicht darum. Es schneidet einfach die Nachkommastellen ab, ohne eine Warnung auszugeben.



Es muss nicht offensichtlich sein, aber das Programm wurde sorgfältig geschrieben, um die Auswirkungen des Rundens auf die Ausgabe so gering wie möglich zu halten.

Dieses Kapitel untersucht die Begrenzungen von Integervariablen. Auch andere Variablentypen werden untersucht, die eingeschlossen, die zur Reduktion von Rundungsfehlern eingeführt wurden. Wir werden uns ihre Vor- und Nachteile anschauen.

5.1 Dezimalzahlen

Integerzahlen sind die Zahlen, an die Sie am meisten gewöhnt sind: 1, 2, 3, usw. und die negativen Zahlen -1, -2, -3 usw. Im alltäglichen Leben sind Integerzahlen am nützlichsten; leider können sie keine gebrochenen Zahlen darstellen. Brüche aus Integerzahlen wie z.B. $\frac{2}{3}$, $\frac{1}{6}$ oder $3\frac{11}{26}$ sind zu umständlich, um damit zu arbeiten. Wenn zwei Brüche nicht den gleichen Nenner haben, ist es schwer, sie zu vergleichen. Z.B. bei der Arbeit am Auto hat es lange gedauert, bis ich wusste, welche Schraube größer ist – $\frac{3}{4}$ oder $\frac{2}{3}$ (es ist die letztere).



Mir wurde gesagt, aber ich kann es nicht beweisen, dass das Problem mit den Integerbrüchen zu der ansonsten unerklärlichen Bedeutung der 12 in unserem Alltag führt. 12 ist die kleinste Integerzahl, die durch 4, 3 und 2 teilbar ist. Ein Viertel von etwas ist in den meisten Fällen genug.

Die Einführung dezimaler Bruchteile hat sich als große Verbesserung herausgestellt. Es ist klar, dass 0.75 kleiner ist als 0.78 (das sind die gleichen Werte wie oben, ausgedrückt als Dezimalzahlen. Außerdem sind mathematische Berechnungen auf Gleitkommazahlen leichter, weil wir keinen kleinsten gemeinsamen Nenner finden oder anderen Unsinn machen müssen.

5.1.1 Begrenzungen von int in C++

Der Variablentyp `int` ist die C++-Version von Integer. Variablen vom Typ `int` leiden unter den gleichen Einschränkungen, unter denen auch ihre Zahläquivalente leiden.

Integer runden

Betrachten Sie das Problem, den Mittelwert von 3 Zahlen zu berechnen. (Das war eine Zusatzaufgabe in Sitzung 4.)

Gegeben seien drei `int`-Variablen `nValue1`, `nValue2` und `nValue3`. Eine Formel für die Berechnung des Mittelwertes ist

```
(nValue1 + nValue2 + nValue3) / 3
```

Diese Gleichung ist richtig und einsichtig. Lassen Sie uns die folgende ebenfalls korrekte und einsichtige Lösung betrachten:

```
nValue1/3 + nValue2/3 + nValue3/3
```

Um zu sehen, welchen Effekt das haben kann, betrachten Sie das folgende einfache Programm, das beide Methoden zur Berechnung des Mittelwertes benutzt:

```
// IntAverage - berechnet Mittelwert von drei
// Zahlen mit Typ int
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{
    int nValue1;
    int nValue2;
    int nValue3;

    cout << »Integerversion\n«;
    cout << »Geben Sie drei Zahlen ein.\n«;
    cout << »Drücken Sie Return nach jeder Zahl.\n«;
    cout << »#1:<»;
    cin >> nValue1;

    cout << »#2:<»;
    cin >> nValue2;

    cout << »#3:<»;
    cin >> nValue3;

    // die folgende Lösung hat keine so großen
    // Probleme mit Rundungsfehlern
    cout << »Addieren vor Teilen ergibt Mittelwert:<«;
    cout << (nValue1 + nValue2 + nValue3)/3;
    cout << »\n«;

    // diese Version hat große Probleme mit
    // Rundungsfehlern
    cout << »Teilen vor Addieren ergibt Mittelwert:<«;
    cout << nValue1/3 + nValue2/3 + nValue3/3;
    cout << »\n«;

    cout << »\n\n«;
    return 0;
}
```

Dieses Programm bekommt die drei Werte `nValue1`, `nValue2` und `nValue3` von `cin`, d.h. über die Tastatur. Es gibt dann den Mittelwert aus, der mit der Addition-vor-Division-Methode berechnet wurde, gefolgt von dem Mittelwert der mit der Division-vor-Addition-Methode berechnet wurde.

Nachdem ich das Programm erzeugt hatte, habe ich es ausgeführt, und die Werte 1, 2 und 3 eingegeben. Erwartet hatte ich, zweimal das Ergebnis 2 zu bekommen. Stattdessen bekam ich das Ergebnis, das Sie in Abbildung 5.1 sehen.



Abbildung 5.1: Der Division-vor-Addition-Algorithmus hat große Probleme mit Rundungsfehlern.

Um den Grund für dieses merkwürdige Verhalten zu verstehen, lassen Sie uns die Werte 1, 2 und 3 direkt in die Gleichung 2 einfügen.

```
1/3 +> 2/3 + 3/3
```

Da Integerzahlen keine Brüche darstellen können, ist das Ergebnis einer Integeroperation immer der abgerundete Bruch.

Dieses Abrunden von Integerzahlen wird auch als Abschneiden bezeichnet.

Unter Berücksichtigung des Integerabschneidens wird der obige Ausdruck zu

```
0 + 0 + 1
```

oder 1.

Der Addition-vor-Division-Algorithmus verhält sich entscheidend besser:

```
(1 + 2 + 3) / 3
```

ist gleich 2 oder 2.

Doch auch der Addition-vor-Division-Algorithmus ist nicht immer korrekt. Geben Sie die Werte 1, 1 und 3 ein. Beide Algorithmen geben 1 anstelle von $1 \frac{2}{3}$ zurück.



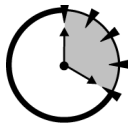
Sie müssen sehr vorsichtig sein, wenn Sie Divisionen mit Integerzahlen durchführen, weil Abschneiden schnell passiert ist.

Eingeschränkter Bereich

Ein zweites Problem mit den `int`-Variablen ist der beschränkte Bereich. Eine normale Integervariable kann einen maximalen Wert von 2.147.483.647 und einen minimalen Wert von -2.147.483.648 annehmen, also ungefähr plus/minus zwei Milliarden.



Einige ältere (in der Tat »sehr alte«) Compiler beschränken den Bereich von `int`-Variablen auf -32.768 bis 32.767.



20 Min.

5.1.2 Lösen des Abschneideproblems

Glücklicherweise versteht C++ Dezimalzahlen. Dezimalzahlen werden in C++ als *Gleitkommazahlen*, oder kurz *floats*, bezeichnet. Der Begriff Gleitkomma kommt daher, dass der Dezimalpunkt hin- und hergleiten kann, so wie es notwendig ist, um einen bestimmten Wert darzustellen.

Gleitkommavariablen werden in der gleichen Weise deklariert wie `int`-Variablen:

```
float fValue1;
```

Um zu sehen, wie Gleitkommazahlen das Rundungsproblem lösen, das Integerzahlen inhärent ist, habe ich alle `int`-Variablen im Programm `IntAverage` durch `float`-Variablen ersetzt. (Das resultierende Programm finden Sie auf der beiliegenden CD-ROM unter dem Namen `FloatAverage`.)

`FloatAverage` konvertiert die Zahlen 1, 2 und 3 in den richtigen Mittelwert 2, für beide Algorithmen, wie in Abbildung 5.2 zu sehen ist.



Wenn Sie die Absicht haben, Berechnungen durchzuführen, halten Sie sich an Gleitkommazahlen.

```
RHIDE Version 1.4 - Kein Projekt
Auto
Dies ist RHIDE Version 1.4. Alle Rechte bei Robert Nöhne, 1996.1997
(Sep 30 1997 23:06:57)
Integerversion
Gehen Sie drei Zahlen ein.
Drücken Sie Return nach jeder Zahl.
#1:1
#2:2
#3:3
Addieren vor Teilen ergibt Mittelwert:2
Teilen vor Addieren ergibt Mittelwert:2
```

Abbildung 5.2: Gleitkommavariablen berechnen den Mittelwert von 1, 2 und 3 korrekt.

5.1.3 Grenzen von Gleitkommazahlen

Während Gleitkommazahlen viele Berechnungsprobleme lösen können, wie z.B. Abschneidungen, haben Sie eine Reihe von Einschränkungen.

Zählen

An einer Stelle können Gleitkommavariablen nicht verwendet werden: wenn gezählt wird. Das betrifft auch C++-Konstrukte, bei denen ein Zähler verwendet wird. Das liegt daran, dass C++ nicht sicher sein kann, welche ganze Zahl mit einer Gleitkommazahl gemeint ist. Z.B. ist klar, dass 1.0 gleich 1 ist, aber was ist mit 0.9 und 1.1? Sollen diese auch als 1 angesehen werden?

C++ schließt diese Probleme aus, indem zum Zählen Variablen vom Typ `int` verwendet werden.

Berechnungsgeschwindigkeit

Historisch gesehen kann ein Computerprozessor ganzzahlige Arithmetik schneller ausführen als Arithmetik auf Gleitkommazahlen. Wenn also ein Prozessor in einer gegebenen Zeit 1000 ganze Zahlen addieren kann, so kann es sein, dass dieser Prozessor in der gleichen Zeit nur 200 Gleitkommaberechnungen ausführen kann. Das Problem der Berechnungsgeschwindigkeit wurde immer kleiner durch die Weiterentwicklung der Mikroprozessoren. Die meisten modernen Prozessoren enthalten spezielle Schaltkreise, mit denen Sie Gleitkommaberechnungen fast so schnell wie Ganzzahlenberechnungen ausführen können.

Genauigkeitsverluste

Auch Gleitkommavariablen lösen nicht alle Berechnungsprobleme. Gleitkommavariablen haben eine begrenzte Genauigkeit: ungefähr sechs Stellen.

Um zu sehen, warum dies ein Problem ist, betrachten Sie die Zahl $\frac{1}{3}$, die als 0.333 ... mit Fortsetzung dieser Reihe dargestellt wird. Das Prinzip einer endlosen Fortsetzung dieser Reihe macht in der Mathematik Sinn, aber nicht in einem Computer. Der Computer hat eine begrenzte Genauigkeit. So ist eine Gleitkommaversion von $\frac{1}{3}$ ungefähr 0.333333. Wenn diese 6 Nachkommastellen wieder mit 3 multipliziert werden, berechnet der Prozessor einen Wert von 0.999999 anstatt des mathematisch erwarteten Wertes 1. Der Genauigkeitsverlust, der auf die Beschränkungen der Gleitkommazahlen zurückzuführen ist, wird als *Rundungsfehler* bezeichnet. C++ kann viele Formen von Rundungsfehlern beheben. Z.B. kann C++ feststellen in der Ausgabe, dass der Benutzer 1 anstelle von 0.999999 gemeint hat. In anderen Fällen jedoch kann sogar C++ die Rundungsfehler nicht beheben.

»Nicht so begrenzter« Bereich

Der Datentyp `float` hat auch einen begrenzten Bereich, obwohl dieser Bereich viel größer ist als vom Datentyp `int`. Der maximale Wert ist ungefähr 10^{38} , das ist eine 1 mit 38 Nullen.



Nur die ersten sechs Ziffern haben eine Bedeutung, weil die restlichen 32 Ziffern unter Rundungsfehlern zu leiden haben. Eine Gleitkommavariablen kann einen Wert von 123.000.000 ohne Rundungsfehler speichern, nicht aber 123.456.789.

5.2 Andere Variablentypen

C++ stellte andere Variablentypen neben `int` und `float` bereit. Diese sind in Tabelle 5-1 zu sehen. Jeder Typ hat seine Vorteile und seine Grenzen.

Tabelle 5-1: Andere C++-Variablentypen

Name	Beispiel	Sinn
<code>char</code>	<code>'c'</code>	Eine einzige <code>char</code> -Variable kann ein einzelnes alphabetisches Zeichen oder eine Ziffer speichern. (Die einfachen Hochkommata zeigen an, dass es sich um ein einzelnes Zeichen handelt.)
<code>string</code>	<code>"Zeichenkette"</code>	Eine Kette von Zeichen; ein Satz. Wird benutzt, um ganze Phrasen zu speichern. (Die doppelten Hochkommata zeigen an, dass es sich um eine <i>Zeichenkette</i> handelt.)
<code>double</code>	<code>1.0</code>	Ein größerer Gleitkommatyp, der 15 signifikante Stellen und ein Maximum von 10^{308} hat.
<code>long</code>	<code>10L</code>	Ein großer Integertyp mit einem Bereich von -2 Milliarden bis 2 Milliarden.

So deklariert

```
// deklariere eine long-Variable und setze sie auf 1
long lVariable;
lVariable = 1;

// deklariere eine double-Variable und setze
// sie auf 1.0
double dVariable;
dVariable = 1.0;
```

die Variable `lVariable` als Variable vom Typ `long` und setzt ihren Wert auf 1, während `dVariable` vom Typ `double` ist, und auf den Wert 1.0 gesetzt wird.

48

Samstagmorgen

Es ist möglich, eine Variable in der gleichen Anweisung zu deklarieren und zu initialisieren:

```
int nVariable = 1; // deklariere eine Variable und
                  // initialisiere sie mit 1
```

**Hinweis**

Variablenamen haben für den C++-Compiler keine besondere Bedeutung.

Eine Variable von Typ `char` kann ein einzelnes Zeichen speichern, während eine Zeichenkette eine Kette von Zeichen enthält. So ist `'a'` das Zeichen `a`, während `"a"` eine Zeichenkette ist, die nur das Zeichen `a` enthält. (»Zeichenkette« ist eigentlich kein Variablentyp, aber in den meisten Fällen können Sie sie als solchen behandeln. Sie erfahren mehr Details über Zeichenketten in Sitzung 14).

**Warnung**

Das Zeichen `'a'` und die Zeichenkette `"a"` sind nicht das Gleiche. Wenn eine Anwendung eine Zeichenkette benötigt, können Sie nicht ein einzelnes Zeichen zur Verfügung stellen, selbst wenn die erwartete Zeichenkette nur ein einzelnes Zeichen enthält.

`long` und `double` sind erweiterte Formen von `int` und `float` – `long` steht für langes Integer, und `double` steht für doppeltes `float`.

5.2.1 Typen von Konstanten

Beachten Sie, wie jeder Datentyp ausgedrückt wird.

In einem Ausdruck wie `n = 1;` ist die Konstante `1` ein `int`. Wenn Sie `1` als `long` gemeint haben, müssten Sie `n = 1L;` schreiben. Vergleichen könnte man das mit einer `1`, die einem Ball auf der Ladefläche eines Pickups entspricht, während `1L` einem Ball auf einem LKW entspricht. Der Ball ist der gleiche, aber die Kapazität seines Behälters ist viel größer.

In gleicher Weise repräsentiert `1.0` den Wert `1` in einem Gleitkommacontainer. Beachten Sie jedoch, dass Gleitkommakonstanten defaultmäßig als `double` angenommen werden. Somit ist `1.0` eine Zahl vom Typ `double` und nicht vom Typ `float`.

5.2.2 Sonderzeichen

Allgemein können Sie jedes druckbare Zeichen in einer `char`-Variable oder in einer Zeichenkette speichern. Es gibt auch eine Menge von nicht druckbaren Zeichen, die so wichtig sind, dass das Gleiche auch für sie gilt. Tabelle 5-2 listet diese Zeichen auf.

Sie haben bereits das Zeichen für den Zeilenumbruch gesehen. Dieses Zeichen bricht eine Zeichenkette in einzelne Zeilen um.

Tabelle 5-2: Nicht druckbare, aber oft verwendete Zeichen

Zeichenkonstante	Bedeutung
'\n'	neue Zeile
'\t'	Tabulator
'\0'	Null
'\\'	Backslash

Bis jetzt haben wir einen Zeilenumbruch nur ans Ende einer Zeichenkette gestellt. Dieses Zeichen kann aber an jeder beliebigen Stelle in der Zeichenkette vorkommen. Betrachten Sie z.B. die folgende Zeichenkette:

```
cout << »Das ist Zeile 1\n Das ist Zeile 2«;
```

erzeugt die Ausgabe

```
Das ist Zeile 1
Das ist Zeile 2
```

In gleicher Weise bewegt das '\t'-Zeichen die Ausgabe an die nächste Tabulatorposition. Was das genau bedeutet, hängt vom Typ des Computers ab, den Sie benutzen.

Da das Backslash-Zeichen verwendet wird, um spezielle Zeichen darzustellen, muss es ein Zeichenpaar geben, um den Backslash selber darzustellen. Das Zeichen '\\' stellt den Backslash dar.

C++-Konflikte mit MS-DOS-Dateiname

MS-DOS benutzt das Backslash-Zeichen, um Verzeichnisnamen im Pfad zu einer Datei zu trennen. So stellt Root\FolderA\File eine Datei in *FolderA* dar, das ein Unterverzeichnis von *Root* ist.

Leider erzeugt der Gebrauch des Backslash von MS-DOS und C++ einen Konflikt. Der MS-DOS-Pfad Root\FolderA\File wird durch die C++-Zeichenkette "Root\\FolderA\\File" dargestellt. Der doppelte Backslash wird durch den speziellen Gebrauch des Backslash-Zeichens in C++ notwendig.

50

Samstagmorgen



10 Min.

5.3 Gemischte Ausdrücke

Ich hasse es fast, das hier darzustellen, aber C++ ermöglicht Ihnen, gemischte Variablentypen in einem Ausdruck zu verwenden. Das heißt, Sie können eine `int`-Zahl und eine `double`-Zahl addieren. Der folgende Ausdruck, in dem `nValue1` ein `int` ist, ist erlaubt:

```
// im folgenden Ausdruck wird der Wert von nValue1
// in ein double konvertiert, bevor die Zuweisung
// ausgeführt wird
int nValue = 1;
nValue1 = nValue1 + 1.0;
```

Ein Ausdruck, in dem die beiden Operanden nicht vom gleichen Typ sind, wird *Mischmodusausdruck* genannt. Mischmodusausdrücke erzeugen einen Wert von dem mächtigeren Typ der beiden Operanden. In diesem Fall wird `nValue1` in ein `double` konvertiert, bevor die Berechnung fortgesetzt wird.

In gleicher Weise kann ein Ausdruck eines Typs einer Variablen eines anderen Typs zugewiesen werden wie in der folgenden Anweisung:

```
// in der folgenden Anweisung wird der ganzzahlige
// Anteil von fVariable in nVariable gespeichert
fVariable = 1.0;
int nVariable;
nVariable = fVariable;
```



Genauigkeit oder Teile des Zahlenbereichs können verloren gehen, wenn die Variable auf der linken Seite der Anweisung »kleiner« ist. Im vorangegangenen Beispiel muss der Wert von `fVariable` abgeschnitten werden, bevor er in `nVariable` gespeichert werden kann.



0 Min.

Einen »größeren« Wert in eine »kleinere« Variable zu konvertieren, wird »*Promotion*« genannt, während die Konvertierung von Werten in der umgekehrten Richtung als »*Demotion*« bezeichnet wird. Wir sagen, dass der Wert der `int`-Variable `nVariable1` in ein `double` befördert (promoviert) wurde:

```
int nVariable = 1;
double dVariable = nVariable1;
```



Mischmodusausdrücke sind keine besonders gute Idee. Sie sollten Ihre eigenen Entscheidungen treffen, anstatt sie C++ zu überlassen. Es kann sein, dass C++ nicht richtig versteht, was Sie eigentlich wollen.

Namenskonventionen

Sie werden bemerkt haben, dass ich jeden Variablennamen mit einem bestimmten Zeichen beginne, das nichts mit dem Namen zu tun zu haben scheint. Diese speziellen Zeichen sind unten dargestellt. Wenn Sie diese Konvention verwenden, können Sie sofort erkennen, dass die Variable `dVariable` vom Typ `double` ist.

Zeichen	Typ
n	int
l	long
f	float
d	double
c	char (Zeichen)
sz	Zeichenkette

Diese führenden Zeichen helfen dem Programmierer, den Überblick über die Variablentypen zu behalten. Sie können sofort erkennen, dass es sich bei dem folgenden Ausdruck um einen Mischmodausdruck handelt, der eine `long`-Variable und eine `int`-Variable enthält.

```
nVariable = lVariable;
```

Bedenken Sie jedoch, dass die Verwendung spezieller Zeichen in Variablennamen für C++ keine besondere Bedeutung hat. Ich hätte genauso `q` zur Identifizierung von `int` verwenden können. Manche Programmierer verwenden überhaupt keine Namenskonvention.

Zusammenfassung

Wie Sie gesehen haben, sind Integervariablen effizient in Bezug auf Rechenzeit, und sie sind einfach zu handhaben. Sie haben jedoch Begrenzungen, wenn sie in Berechnungen eingesetzt werden. Gleitkommazahlen sind ideal für die Verwendung in mathematischen Gleichungen, da sie nicht unter signifikanten Rundungsfehlern oder einer signifikanten Beschränkung ihres Bereiches leiden. Auf der anderen Seite sind Gleitkommavariablen schwerer zu handhaben und nicht so universell einsetzbar wie Integervariablen.

- Integervariablen stellen Zähler dar, wie 1, 2, usw.
- Integervariablen haben einen Bereich von -2 Milliarden bis 2 Milliarden.
- Gleitkommavariablen stellen Dezimalbrüche dar.
- Gleitkommavariablen haben praktisch einen unbeschränkten Bereich.
- Der Variablentyp `char` wird zur Darstellung von ANSI-Zeichen verwendet.

Selbsttest

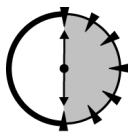
1. Was ist der Bereich von `int`-Variablen? (Siehe »Eingeschränkter Bereich«)
2. Warum leiden `float`-Variablen nicht unter signifikanten Rundungsfehlern? (Siehe »Lösen des Abschneideproblems«)
3. Was ist der Typ der Konstante 1? Was ist der Typ von 1.0? (Siehe »Typen von Konstanten«)

Mathematische Operationen



Checkliste

- ☒ Mathematische Operatoren von C++ gebrauchen
- ☒ Ausdrücke erkennen
- ☒ Klarheit durch spezielle mathematische Operatoren erhöhen



30 Min.

Die Programme Conversion und Average haben einfache mathematische Operationen wie Addition, Multiplikation und Division verwendet. Ich habe diese Operatoren verwendet, ohne sie vorher zu beschreiben, weil sie intuitiv klar sind. Diese Sitzung beschreibt die Menge der mathematischen Operatoren.

Die mathematischen Operatoren sind in Tabelle 6-1 aufgelistet. Die erste Spalte listet die Operatoren von oben nach unten geordnet, gemäß ihrer Priorität.

Tabelle 6-1: Mathematische Operatoren von C++

Operator	Bedeutung
+ (unär)	tut effektiv nichts
- (unär)	gibt das Negative des Arguments zurück
++ (unär)	Inkrement
-- (unär)	Dekrement
*	Multiplikation
/	Division
%	Modulo
+	Addition
-	Subtraktion
=, *, %, +=, -=	Zuweisungstypen

Jeder dieser Operatoren wird in den folgenden Abschnitten behandelt.

6.1 Arithmetische Operatoren

Die Operatoren Multiplikation, Division, Modulo, Addition und Subtraktion sind die Operatoren, die zur Ausführung gewöhnlicher Arithmetik verwendet werden. Jeder dieser Operatoren hat die übliche Bedeutung, die Sie aus der Schule kennen werden, vielleicht mit Ausnahme von Modulo.

Der Modulo-Operator ist dem sehr ähnlich, was mein Lehrer als »Rest nach Division« bezeichnet hat. Z.B. passt 4 drei mal in 15 hinein mit einem Rest von 3. Ausgedrückt in C++-Notation lautet dies »15 modulo 4 ist 3«.

Weil Programmierer immer bemüht sind, Nichtprogrammierer mit den einfachsten Dingen zu beeindrucken, definieren C++-Programmierer modulo wie folgt:

```
IntValue % IntDivisor
```

ist gleich

```
IntVaue - (IntValue / IntDivisor) * IndDivisor
```

Lassen Sie uns das an unserem früheren Beispiel ausprobieren:

```
15 % 4 ist gleich    15 - (15/4) * 4
                    15 - 3 * 4
                    15 - 12
                    3
```



Weil modulo auf Rundungsfehlern basiert, die Integerzahlen inhärent sind, ist modulo für Gleitkommazahlen nicht definiert.

6.2 Ausdrücke

Der häufigste Typ von C++-Anweisungen ist der Ausdruck. Ein *Ausdruck* ist eine Anweisung, die einen Wert hat. Alle Ausdrücke haben einen Wert.

Z.B. ist eine Anweisung, die einen mathematischen Operator enthält, ein Ausdruck, da alle diese Operatoren einen Wert zurückgeben. Ausdrücke können einfach oder auch kompliziert sein. In der Tat ist »1« ein Ausdruck. Es gibt fünf Ausdrücke in der folgenden Anweisung:

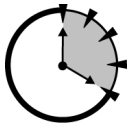
```
z = x * y + w;
1. x * y + w
2. x * y
3. x
4. y
5. w
```

Ein ungewöhnlicher Gesichtspunkt in C++ ist, dass ein Ausdruck eine vollständige Anweisung ist. Somit ist das folgende eine gültige C++-Anweisung.

```
1;
```

Lektion 6 – Mathematische Operationen 55

Alle Ausdrücke haben einen Typ. Wie wir bereits festgestellt haben, ist der Typ des Ausdrucks 1 gleich `int`. In einer Zuweisung ist der Typ des Ausdrucks auf der rechten Seite der Anweisung gleich dem Typ der Variablen auf der linken Seite des Ausdrucks – wenn nicht, führt C++ die notwendigen Konvertierungen durch.



20 Min.

6.3 Vorrang von Operatoren

Jeder der C++-Operatoren hat eine Priorität, mit der er innerhalb von zusammengesetzten Ausdrücken (d.h. Ausdrücken mit mehr als einem Operator) ausgewertet wird. Diese Eigenschaft wird als Vorrang bezeichnet.

Der Ausdruck

```
x/100 + 32
```

teilt `x` durch 100 bevor 32 addiert wird. In einem gegebenen Ausdruck führt C++ Multiplikationen und Divisionen vor Additionen und Subtraktionen aus. Wir sagen, dass Multiplikation und Division *Vorrang* vor Addition und Subtraktion haben.

Was ist, wenn der Programmierer `x` durch 100 plus 32 teilen möchte? Der Programmierer kann Ausdrücke durch Klammern verbinden:

```
x/(100 + 32)
```

Dies hat den gleichen Effekt, wie `x` durch 132 zu teilen. Der Ausdruck innerhalb der Klammern wird zuerst ausgewertet. Dies ermöglicht dem Programmierer die Vorrangsregeln einzelner Operatoren zu überschreiben.

Der ursprüngliche Ausdruck

```
x / 100 + 32
```

ist identisch mit dem Ausdruck

```
(x / 100) + 32
```

Der Vorrang der Operatoren aus Tabelle 6-1 ist in Tabelle 6-2 zu sehen.

Tabelle 6-2: Mathematische Operatoren von C++ und ihre Priorität

Priorität	Operator	Bedeutung
1	<code>+</code> (unär)	tut effektiv nichts
1	<code>-</code> (unär)	gibt das Negative des Argumentes zurück
2	<code>++</code> (unär)	Inkrement
2	<code>--</code> (unär)	Dekrement
3	<code>*</code>	Multiplikation
3	<code>/</code>	Division
3	<code>%</code>	Modulo
4	<code>+</code>	Addition
4	<code>-</code>	Subtraktion
5	<code>=</code> , <code>*=</code> , <code>%=</code> , <code>+=</code> , <code>-=</code>	Zuweisungstypen

56**Samstagmorgen**

Operatoren mit der gleichen Priorität werden von links nach rechts ausgewertet. Somit ist der Ausdruck

```
x / 10 / 2
```

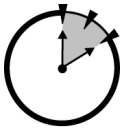
das Gleiche wie

```
(x / 10) / 2
```

Mehrstufige Klammerungen werden von innen nach außen ausgewertet. Im folgenden Ausdruck

```
(y / (2 + 3)) / x
```

wird die Variable y durch 5 geteilt, und das Ergebnis wird durch x geteilt.

**10 Min.**

6.4 Unäre Operatoren

Unäre Operatoren sind die Operatoren, die nur ein Argument haben.

Ein binärer Operator ist ein Operator, der zwei Argumente hat. Betrachten Sie z.B. $a + b$. In C++-Sprechweise sind die Argumente des Additionsoperators die Ausdrücke auf der linken und der rechten Seite des Operators.

Die unären Operatoren sind $+$, $-$, $++$ und $--$.

Der Minusoperator ändert das Vorzeichen seines Arguments. Positive Zahlen werden negativ und umgekehrt. Der Plusoperator ändert das Vorzeichen seines Arguments nicht. Letztendlich hat der Plusoperator keinen Effekt.

Die Operatoren $++$ und $--$ inkrementieren bzw. dekrementieren ihre Argumente um eins.

Wozu einen eigenen Inkrement-Operator?

Die Autoren von C++ haben erkannt, dass Programmierer »1« mehr als jede andere Konstante addieren. Als Bequemlichkeitsfaktor wurde eine spezielle »Addiere 1«-Instruktion in der Sprache eingeführt.

Zusätzlich haben die meisten Computerprozessoren eine Inkrementanweisung, die schneller ist als die Additionsanweisung. Als C++ entstanden ist, war das Einsparen von Anweisungen eine wichtige Sache, wenn man sich den damaligen Entwicklungsstand der Mikroprozessoren bewusst macht.

Unabhängig davon, aus welchem Grund die Inkrement- und Dekrementoperatoren eingeführt wurden, werden Sie in Sitzung 7 sehen, dass sie nützlicher sind, als Sie jetzt vielleicht denken.

**Hinweis**

Die Inkrement- und Dekrementoperatoren sind auf nicht-Gleitkomma-Variablen beschränkt.

Lektion 6 – Mathematische Operationen 57

Die Inkrement- und Dekrementoperatoren sind in der Hinsicht besonders, dass Sie in zwei Formen vorkommen: einer Präfix-Version und einer Postfix-Version.



Die Präfix-Version des Inkrements wird als ++x geschrieben, während das Postfix durch x++ ausgedrückt wird.

Betrachten Sie den Inkrement-Operator (der Dekrement-Operator ist genau analog). Nehmen Sie an, dass die Variable `n` den Wert 5 hat. Beide, `n++` und `++n`, inkrementieren den Wert von `n` zu 6. Der Unterschied ist, dass der Wert von `++n` in einem Ausdruck gleich 6 ist, während der Wert von `n++` in einem Ausdruck gleich 5 ist. Das folgende Beispiel demonstriert das:

```
// deklariere drei int-Variablen
int n1, n2, n3;

// der Wert von n1 und n2 ist gleich 6
n1 = 5;
n2 = ++n1;

// der Wert von n1 ist 6, aber der Wert n3 ist 5
n1 = 5;
n3 = n1++;
```

Somit bekommt `n2` den Wert von `n1`, *nachdem* `n1` über die Präfix-Version inkrementiert wurde, während `n3` den Wert von `n1` erhält, *bevor* `n1` über die Postfix-Version inkrementiert wird.

Teil 2 – Samstagmorgen
Lektion 6

6.5 Zuweisungsoperatoren

Die Zuweisungsoperatoren sind binäre Operatoren, die das Argument auf ihrer linken Seite verändern.

Der einfache Zuweisungsoperator '=' ist eine absolute Notwendigkeit in jeder Programmiersprache. Der Operator speichert den Wert des Arguments auf der rechten Seite im Argument auf der linken Seite. Die anderen Zuweisungsoperatoren scheinen irgendeiner Laune entsprungen zu sein.

Die Autoren von C++ haben festgestellt, dass Zuweisungen oft die folgende Form haben:

```
variable = variable # constant
```

wobei '#' ein binärer Operator ist. Um also einen Integeroperanden um 2 zu inkrementieren, könnte der Programmierer schreiben:

```
nVariable = nVariable + 2;
```

Dies besagt, dass 2 zum Wert von `nVariable` hinzugefügt und das Ergebnis in `nVariable` gespeichert werden soll.



Es ist üblich, dieselbe Variable auf der linken und auf der rechten Seite der Zuweisung stehen zu haben.

58

Samstagmorgen**0 Min.**

Weil die gleiche Variable auf der linken und der rechten Seite des Gleichheitszeichens steht, haben sie entschieden, dem Zuweisungsoperator einen weiteren Operator hinzuzufügen. Alle binären Operatoren haben eine Zuweisungsversion. Somit kann die obige Anweisung wie folgt geschrieben werden:

```
nVariable += 2;
```

Noch mal, dies besagt, dass 2 zum Wert von `nVariable` hinzugefügt werden soll.

**Tipp**

Anders als die Zuweisung selber, werden diese Zuweisungsoperatoren nicht allzu oft benutzt. In bestimmten Fällen kann Ihre Verwendung jedoch das Programm deutlich lesbarer machen.

Zusammenfassung

Die mathematischen Operatoren werden in C++-Programmen öfter verwendet als alle anderen Operatoren. Das ist wenig verwunderlich: C++-Programme konvertieren immer Temperaturen von Grad Celsius in Grad Fahrenheit und zurück und führen unzählige andere Operationen durch, die Addition, Subtraktion und Zählen erforderlich machen.

- Alle Ausdrücke haben einen Wert und einen Typ.
- Die Ordnung der Auswertung innerhalb eines Ausdrucks wird normalerweise bestimmt durch den Vorrang der Operatoren. Diese Reihenfolge kann mit Hilfe von Klammern überschrieben werden.
- Für viele der am häufigsten verwendeten Ausdrücke stellt C++ Kurzformen bereit. Der geläufigste ist der `i++`-Operator anstelle von `i = i + 1`.

Selbsttest

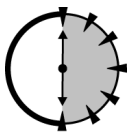
1. Was ist der Wert von `9 % 4`? (Siehe »Arithmetische Operatoren«)
2. Ist `9 % 4` ein Ausdruck? (Siehe »Ausdrücke«)
3. Wenn `n = 4`, was ist der Wert von `n + 10 / 2`? Warum ist er 9 und nicht 7? (Siehe »Vorrang von Operatoren«)
4. Wenn `n = 4`, was ist der Unterschied zwischen `++n` und `n++`? (Siehe »Unäre Operatoren«)
5. Wie würde man `n = n + 2` unter Verwendung des Operators `+=` schreiben? (Siehe »Zuweisungsoperatoren«)

Logische Operationen



Checkliste

- ☒ Einfache logische Operatoren und Variablen einsetzen
- ☒ Mit binären Zahlen arbeiten
- ☒ Bitweise Operationen ausführen
- ☒ Logischer Zuweisungsanweisungen erzeugen



30 Min.

Vielleicht mit Ausnahme der Inkrement- und Dekrementoperatoren sind die mathematischen Operatoren von C++ geläufige Operatoren und kommen im Alltag vor. Im Vergleich dazu sind die logischen Operatoren von C++ unbekannt.

Es ist nicht so, dass sich die Menschen nicht mit logischen Operationen beschäftigen. Wenn Sie sich an den mechanischen Reifenwechsler in Sitzung 1 erinnern, ist die Fähigkeit, logische Berechnungen auszudrücken, wie »wenn der Reifen platt ist UND ich einen Schraubenschlüssel habe...« ein Muss. Die Menschen berechnen ständig AND und OR, sie sind nur nicht daran gewöhnt, das hinzuschreiben.

Logische Operatoren zerfallen in zwei Klassen. Der erste Typ, den ich einfache logische Operatoren nenne, sind Operatoren, die im alltäglichen Leben vorkommen. Der zweite Typ, die bitweisen logischen Operatoren, gibt es nur in der Welt des Computers. Ich werde erst die einfachen Operatoren vorstellen, bevor wir uns den bitweisen Operatoren zuwenden.

7.1 Einfache logische Operatoren

Die einfachen logischen Operatoren repräsentieren dieselbe Art logische Operation, die Sie in Ihrem alltäglichen Leben treffen: wie z.B. ob etwas wahr oder falsch ist oder der Vergleich zweier Dinge. Die einfachen logischen Operatoren sind in Tabelle 7-1 zu sehen.

Tabelle 7-1: Einfache logische Operatoren

Operator	Bedeutung
==	Gleichheit; wahr, wenn das Argument auf der linken Seite den gleichen Wert wie das Argument auf der rechten Seite hat
!=	Ungleichheit; Gegenteil von Gleichheit
>, <	größer als, kleiner als; wahr, wenn das Argument auf der linken Seite größer/kleiner als das Argument auf der rechten Seite ist
>=, <=	größer oder gleich, kleiner oder gleich; wahr, wenn entweder > oder == wahr sind / < oder == wahr sind
&&	AND; wahr, wenn beide Argumente auf der rechten und der linken Seite wahr sind
	OR; wahr, wenn das linke Argument oder das rechte Argument wahr ist
!	NOT; wahr, wenn das Argument falsch ist

Die ersten sechs Einträge in Tabelle 7-1 sind die Vergleichsoperatoren. Der Gleichheitsoperator wird verwendet, um zwei Zahlen miteinander zu vergleichen. Z.B. ist das Folgende wahr (true), wenn der Wert der Variable `nVariable` gleich 0 ist:

```
nVariable == 0;
```



Verwechseln Sie nicht den Gleichheitsoperator `==` mit dem Zuweisungsoperator `=`. Das ist nicht nur ein oft gemachter Fehler, das ist auch ein Fehler, den der C++-Compiler nicht abfangen kann.

```
nVariable = 0; // Programmierer meinte nVariable == 0
```

Die Operatoren Größer-als (>) und Kleiner-als (<) sind ähnlich geläufig im alltäglichen Leben. Der folgende logische Vergleichsausdruck ist wahr:

```
int nVariable1 = 1;
int nVariable2 = 2;
nVariable1 < nVariable2;
```

Die Operatoren Größer-oder-gleich (>=) und Kleiner-oder-gleich (<=) sind ähnlich, nur dass sie die Gleichheit mit einschließen, was die anderen Operatoren nicht tun.

Die Operatoren && (AND) und || (OR) sind ähnlich geläufig. Diese Operatoren werden typischerweise mit den anderen logischen Operatoren kombiniert:

```
// wahr wenn nV2 größer als nV1, aber kleiner als nV3
(nV1 < nV2) && (nV2 < nV3)
```


Lektion 7 – Logische Operationen 61



Seien Sie vorsichtig mit der Verwendung des Vergleichsoperators auf Gleitkommazahlen. Betrachten Sie das folgende Beispiel:

```
float fVariable1 = 10.0;
float fVariable2 = (10 / 3) * 3;
```

fVariable1 == fVariable2; // sind die beiden gleich?

Der Vergleich in obigem Beispiel liefert nicht notwendigerweise wahr (true). 10/3 ist gleich 3.333..., aber C++ kann nicht eine unendliche Anzahl von Nachkommastellen darstellen. Als Gleitkommazahl gespeichert, wird 10/3 etwa zu 3.333333. Wenn Sie diese Zahl mit 3 multiplizieren, bekommen Sie 9.999999 als Ergebnis, was nicht genau gleich 10 ist.

Genauso wie $(10.0 / 3) * 3$ nicht genau 10.0 ist, kann das Ergebnis einer Gleitkommarechnung ein klein wenig daneben liegen. Solche kleinen Abweichungen werden Sie oder ich kaum beachten, aber der Computer. Gleichheit bedeutete exakte Gleichheit.

Ein sicherer Vergleich sieht so aus:

```
float fDelta = fVariable1 - fVariable2;
fDelta < 0.01 && fDelta > -0.01;
```

Der Vergleich liefert wahr, wenn die Variablen fVariable1 und fVariable2 innerhalb eines Deltawertes nebeneinander liegen. (Der Begriff »Delta« ist mit Absicht vage – er soll einen akzeptablen Fehler darstellen.)



Der numerische Prozessor in ihrem PC wird große Anstrengungen unternehmen, um solche Rundungsfehler zu vermeiden – anhängig von der Situation, kann der numerische Prozessor automatisch ein kleines Delta bereitstellen. Darauf sollten Sie sich aber nicht verlassen.

Teil 2 – Samstagmorgen
Lektion 7

7.1.1 Kurze Schaltkreise und C++

Die Operatoren && und || führen das aus, was als »Kurze-Schaltkreis-Evaluierung« bezeichnet wird. Betrachten Sie das Folgende:

```
condition1 && condition2;
```

Wenn condition1 nicht wahr ist, dann ist das Ergebnis nicht wahr, unabhängig davon, welchen Wert condition2 hat (d.h. condition2 kann wahr oder falsch sein ohne Einfluss auf das Ergebnis). In gleicher Weise ist

```
condition1 || condition2;
```

wahr, wenn condition1 wahr ist, unabhängig davon, welchen Wert condition2 hat.

Um Zeit zu sparen, wertet C++ condition1 zuerst aus. C++ wertet condition2 nicht aus, wenn condition1 bereits FALSE ist im Falle von &&, bzw. TRUE im Falle von ||.

7.1.2 Logische Variablentypen

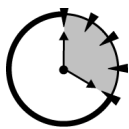
Wenn `>` und `&&` Operatoren sind, dann muss ein Vergleich wie `a > 10` ein Ausdruck sein. Natürlich muss das Ergebnis eines solchen Ausdrucks entweder `TRUE` oder `FALSE` sein.

Sie haben sicher bemerkt, dass in Sitzung 5 kein Boolescher Variablentyp erwähnt wurde. Es gibt also keinen Variablentyp, der nur die Werte `TRUE` und `FALSE` und sonst nichts annehmen kann. Was ist denn dann der Typ eines Ausdrucks von der Form `a > 10`?

C++ verwendet den Typ `int`, um Boolesche Werte zu speichern. Der Wert 0 steht für `FALSE`. Jeder andere Wert ungleich 0 bedeutet `TRUE`. Ein Ausdruck wie `a > 10` wird ausgewertet zu 0 (`FALSE`) oder 1 (`TRUE`).



Microsoft Visual Basic verwendet auch Integerwerte zur Speicherung von `TRUE` und `FALSE`, Vergleichsoperatoren geben in Visual Basic jedoch 0 (`FALSE`) oder -1 (`TRUE`) zurück.



20 Min.

7.2 Binäre Zahlen

Die so genannten bitweisen logischen Operatoren arbeiten auf ihren Argumenten auf Bitebene. Um zu verstehen, wie sie arbeiten, lassen Sie uns zuerst binäre Zahlen ansehen, d.h. Zahlen, so wie Computer sie darstellen.

Die Zahlen, mit denen wir vertraut sind, werden als Dezimalzahlen bezeichnet, da sie auf der Zahl 10 basieren. Eine Zahl wie 123 bedeutet 1×100 plus 2×10 plus 3×1 . Jede der Basiszahlen 100, 10 und 1 ist eine Potenz von 10.

$$123_{10} = 1 * 100 + 2 * 10 + 3 * 1$$

Die Verwendung von 10 als Basis für unser Zahlensystem kommt sehr wahrscheinlich daher, dass wir 10 Finger haben und diese ursprünglich als Zählhilfsmittel verwendet wurden. Wenn unser Zahlensystem von Hunden erfunden worden wäre, würde es sicher auf der Zahl 8 basieren (ein »Finger« jeder Tatze ist auf ihrer Rückseite und daher nicht sichtbar). Solch ein Oktalsystem würde ebenso gut arbeiten:

$$123_{10} = 173_8 = 1 + 64_{10} + 7 * 8_{10} + 3$$

Die kleinen Ziffern 10 und 8 weisen auf das Zahlensystem hin, 10 für dezimal (Basis 10) und 8 für oktal (Basis 8). Ein Zahlensystem kann jede Basis verwenden, nur nicht 1.

Computer haben im Wesentlichen zwei Finger. Aufgrund Ihres Aufbaus bevorzugen Sie es, mit der Basis 2 zu rechnen. Die Zahl 123_{10} wird ausgedrückt als

$$0 * 128 + 1 * 64 + 1 * 32 + 1 * 16 + 1 * 8 + 0 * 4 + 1 * 2 + 1 * 1$$

oder 01111011_2 .

Es ist Konvention, binäre Zahlen entweder mit 4, 8, 16, 32 oder 64 Bits darzustellen, selbst wenn die führenden Bits null sind. Das hängt auch wieder mit dem internen Aufbau von Computern zusammen.



Weil der Begriff »Ziffer« sich auf ein Vielfaches von 10 bezieht, wird eine binäre Ziffer als Bit bezeichnet. Bit kommt von binary digit. Acht Bits bilden ein Byte.

Lektion 7 – Logische Operationen 63

Mit einer so kleinen Basis ist es nötig, viele Bits zur Darstellung großer Zahlen zu verwenden. Es ist unbequem, einen Ausdruck wie 01111011_2 für die Darstellung eines einfachen Wertes wie 123_{10} zu verwenden. Programmierer bevorzugen es, Zahlen als Einheiten von jeweils 4 Bits darzustellen.

Eine einzelne 4-Bit-Ziffer ist im Wesentlichen die Basis 16, weil 4 Bits jeden Wert zwischen 0 und 15 darstellen können. Die Basis 16 ist unter dem Namen Hexadezimalsystem bekannt. *Hexadezimal* wird oft mit *hex* abgekürzt.

Hexadezimalzahlen verwenden die gleichen Ziffern für die Zahlen von 0 bis 9. Für die Ziffern zwischen 9 und 16 verwenden die Hexadezimalzahlen die ersten sechs Buchstaben des Alphabets. A für 10, B für 11, usw. Es ist also 123_{10} gleich $7B_{16}$.

$$7 * 16 + B \text{ (d.h. 11)} * 1 = 123$$

Weil Programmierer es bevorzugen, Zahlen in 4, 8, 32 oder 64 Bits auszudrücken, bevorzugen Sie in ähnlicher Weise eine Darstellung hexadezimaler Zahlen mit 1, 2, 4 oder 8 hexadezimalen Ziffern, selbst wenn die führenden Ziffern 0 sind.

Schließlich ist es unbequem, eine Hexadezimalzahl wie 7B mit Hilfe des Subscripts 16 auszudrücken, weil Terminals das nicht unterstützen. Selbst mit einem Textprogramm, wie ich es gerade benutze, ist es unbequem, den Zeichensatz auf Subscript umzuschalten und wieder zurück, nur um diese beiden Ziffern zu schreiben. Deshalb verwenden Programmierer die Konvention, dass eine Hexadezimalzahl mit "0x" beginnt (der Grund für eine solch merkwürdige Konvention geht auf die frühen Tage von C zurück). Dann wird 7B zu 0x7B. Mit dieser Konvention sind die Zahlen 0x123 und 123 voneinander unterscheidbar. (0x123 entspricht 291 dezimal.)

7.3 Bitweise logische Operationen

Alle Operatoren die bisher definiert wurden, können auf hexadezimale Zahlen genauso angewendet werden, wie sie auf Dezimalzahlen angewendet werden konnten. Der Grund, weshalb wir eine Multiplikation wie $0xC \times 0xE$ nicht im Kopf ausführen können, liegt vielmehr darin, wie wir die Multiplikation in der Schule gelernt haben, als an irgendwelchen Beschränkungen der Operatoren.

Zusätzlich zu den mathematischen Operatoren gibt es eine Menge von Operationen, die auf Einzelbit-Operatoren basieren. Diese Basisoperationen sind nur für 1-Bit-Zahlen definiert.

7.3.1 Die Einzelbit-Operatoren

Die bitweisen Operatoren führen logische Operationen auf einzelnen Bits aus. Wenn Sie 0 als falsch (false) und 1 als wahr (true) annehmen (das muss nicht so sein, aber es ist eine übliche Konvention), dann können Sie für den bitweisen AND-Operator Dinge sagen wie die folgenden:

```
1 (true) AND 1 (true) ist 1 (true)
1 (true) AND 0 (false) ist 0 (false)
```

Und in gleicher Weise für den OR-Operator.

```
1 (true) OR 0 (false) ist 1 (true)
0 (false) OR 0 (false) ist 0 (false)
```

Geschrieben in Tabellenform sieht das wie in den Tabellen 7-2 und 7-3 aus.

Tabelle 7-2: Wahrheitstafel für den Operator AND

AND	1	0
1	1	0
0	0	0

Tabelle 7-3: Wahrheitstafel für den Operator OR

OR	1	0
1	1	1
0	1	0

In Tabelle 7-2 wird das Ergebnis von 1 AND 0 in Zeile 1 gezeigt (1 in Zeilenkopf) und Spalte 2 (0 im Spaltenkopf).

Ein anderer logischer Operator, der so im alltäglichen Leben nicht vorkommt, ist der Operator »oder sonst«, der üblicherweise mit XOR abgekürzt wird. XOR liefert wahr, wenn eines der Argumente wahr ist, aber nicht beide Argumente gleichzeitig wahr sind. XOR ist in Tabelle 7-4 dargestellt.

Tabelle 7-4: Wahrheitstabelle für den Operator XOR

XOR	1	0
1	0	1
0	1	0

Ausgerüstet mit diesen Einzelbit-Operatoren können wir uns den logischen Operatoren von C++ zuwenden.

7.3.2 Die bitweisen Operatoren

Die bitweisen Operatoren von C++ führen Bitoperationen auf jedem einzelnen Bit ihres Argumentes aus. Die einzelnen Operatoren sind in der Tabelle 7.5 zu sehen.

Tabelle 7-5: Die bitweisen Operatoren von C++

Operator	Funktion
~	NOT: invertiere jedes Bit von 0 nach 1 und von 1 nach 0
&	AND: verknüpfe jedes Bit der linken Seite mit dem entsprechenden Bit auf der rechten Seite durch »und«
	OR
^	XOR

Lektion 7 – Logische Operationen 65

Der Operator NOT ist am einfachsten zu verstehen. NOT konvertiert 1 in 0 und 0 in 1. (D.h. 0 ist NOT 1 und 1 ist NOT 0.)

```
~01102 (0x6)
10012 (0x9)
```

Der Operator NOT ist der einzige unäre bitweise logische Operator. Die folgende Rechnung demonstriert den &-Operator.

```
01102
&
00112
00102
```

Von links nach rechts: 0 AND 0 ist 0 (erstes Bit), 1 AND 0 ist 0 (zweites Bit), 1 AND 1 ist 1 (drittes Bit) und 0 AND 1 ist 0 (am wenigsten signifikantes Bit).

Die gleichen Berechnungen können auf Zahlen ausgeführt werden, die als Hexadezimalzahl dargestellt sind, indem sie erst in Binärzahlen verwandelt werden und dann das Ergebnis in Hexadezimaldarstellung konvertiert wird.

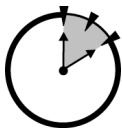
```
0x6      01102
&
0x3      00112
00102   -> 0x2
```

Teil 2 – Samstagmorgen
Lektion 7



Tipp

Solche Hin- und Herkonvertierungen sind viel einfacher mit Hexadezimalzahlen auszuführen als mit Dezimalzahlen. Glauben Sie es oder nicht, mit ein wenig Erfahrung können Sie bitweise Operationen im Kopf ausführen.



10 Min.

7.3.3 Ein einfacher Test

Wir benötigen ein einfaches Programm, um Ihre Fähigkeiten, bitweise Operationen auszuführen, zu testen; erst auf Papier und dann im Kopf. Listing 7-1 ist ein Programm, das zwei Hexadezimalzahlen von der Tastatur erwartet, und das Ergebnis der Operatoren AND, OR und XOR ausgibt.

Listing 1: 7-1: Test der bitweisen Operatoren

```
// BitTest - Eingabe von zwei Hexadezimalzahlen
//           über die Tastatur und dann Ausgabe
//           der Ergebnisse von Anwendung der
//           Operatoren &, | and ^
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* nArgs[])
{
    // setze Ausgabeformat auf hexadecimal
    cout.setf(ios::hex);
```

66

Samstagmorgen

```
// Eingabe des ersten Argumentes
int nArg1;
cout << "Eingabe nArg1 hexadezimal (4 Ziffern):";
cin >> nArg1;

int nArg2;
cout << "Eingabe nArg2 hexadezimal (4 Ziffern):";
cin >> nArg2;

cout << "nArg1 & nArg2 = 0x"
    << (nArg1 & nArg2) << "\n";
cout << "nArg1 | nArg2 = 0x"
    << (nArg1 | nArg2) << "\n";
cout << "nArg1 ^ nArg2 = 0x"
    << (nArg1 ^ nArg2) << "\n";

return 0;
}
```

Ausgabe:

```
Eingabe nArg1 hexadezimal (4 Ziffern): 0x1234
Eingabe nArg2 hexadezimal (4 Ziffern): 0x00ff
nArg1 & nArg2 = 0x34
nArg1 | nArg2 = 0x12ff
nArg1 ^ nArg2 = 0x12cb
```

Die erste Anweisung, die `cout.setf(ios::hex);` lautet, setzt das Ausgabeformat von standardmäßig dezimal auf hexadezimal (im Augenblick müssen Sie mir glauben, dass das so funktioniert).

Der Rest des Programms ist einfach. Das Programm liest `nArg1` und `nArg2` von der Tastatur und gibt dann alle Kombinationen von bitweisen Berechnungen aus.

Die Ausgabe des Programms bei Eingabe von `0x1234` und `0x00ff` sehen Sie oben am Ende des Listings.



Hexadezimalzahlen werden mit einem führenden 0x geschrieben.

7.3.4 Warum?

Der Sinn der meisten Operatoren ist klar. Niemand würde nach dem Sinn des Plus-Operators oder des Minus-Operators fragen. Der Sinn der Operatoren `<` und `>` ist klar. Für den Anfänger muss nicht klar sein, wann und warum man bitweise Operatoren verwendet.

Der Operator AND wird oft verwendet, um Information auszumaskieren. Nehmen Sie z.B. an, dass wir die am wenigsten signifikante Hexadezimalstelle aus einer Zahl mit vier Ziffern extrahieren möchten.

```
0x1234      0001 0010 0011 0100
&
0x000F      0000 0000 0000 1111
            0000 0000 0000 0100  -> 0x0004
```

Lektion 7 – Logische Operationen 67

Eine andere Anwendung ist das Setzen und Auslesen einzelner Bits.

Nehmen Sie an, dass wir in einer Datenbank Informationen über Personen in einem einzelnen Byte pro Person speichern. Das signifikanteste Bit könnte z.B. gesetzt werden, wenn die Person männlich ist, das nächste Bit, wenn es ein Programmierer ist, das nächste, wenn die Person attraktiv ist, und das am wenigsten signifikante Bit, wenn die Person einen Hund hat.

Bit	Bedeutung
0	1 -> männlich
1	1 -> Programmierer
2	1 -> attraktiv
3	1 -> hat einen Hund



0 Min.

Dieses Byte würde für jede Person kodiert und zusammen mit dem Namen, Versicherungsnummer und allen weiteren legalen Informationen gespeichert.

Ein hässlicher männlicher Programmierer, der einen Hund besitzt, würde als 1101_2 kodiert. Um alle Einträge in der Datenbank zu testen, um nach nicht attraktiven Programmierern zu suchen, die keinen Hund haben, unabhängig vom Geschlecht, würden wir den folgenden Ausdruck verwenden:

```
value & 0x0110) == 0x0100
***      ^ -> 0 = nicht attraktiv
          ^   1 = ist Programmierer
          * -> nicht von Interesse
          ^ -> von Interesse
```



Hinweis

In diesem Fall wird der Wert 0110 als Maske bezeichnet, weil er Bits ausmaskiert, die nicht von Interesse sind.

Zusammenfassung

Sie haben die mathematischen Operatoren aus Kapitel 6 bereits in der Schule gelernt. Sie haben dort sicherlich nicht die einfachen logischen Operatoren gelernt, sie kommen aber im alltäglichen Leben vor. Operatoren wie AND und OR sind ohne Erklärung verständlich. Da C++ keinen logischen Variablentyp hat, verwendet es 0 zur Darstellung von FALSE und alles andere zur Darstellung von TRUE.

Im Vergleich dazu sind die binären Operatoren etwas Neues. Diese Operatoren führen die gleichen Operationen AND und OR aus, aber auf jedem Bit separat.

Selbsttest

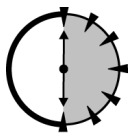
1. Was ist der Unterschied zwischen den Operatoren && und &? (Siehe »Einfache logische Operatoren«)
2. Was ist der Wert von (1 && 5)? (Siehe »Einfache logische Operatoren«)
3. Was ist der Wert von 1 & 5? (Siehe »Bitweise logische Operatoren«)
4. Drücken Sie 215 als Summe von Zehnerpotenzen aus. (Siehe »Binäre Zahlen«)
5. Drücken Sie 215 als Summe von Zweierpotenzen aus. (Siehe »Binäre Zahlen«)
6. Was ist 215 in Hexadezimaldarstellung? (Siehe »Binäre Zahlen«)

Kommandos zur Flusskontrolle



Checkliste

- ☒ Kontrolle über den Programmfluss
- ☒ Wiederholte Ausführung einer Gruppe
- ☒ Vermeidung von »Endlosschleifen«



30 Min.

Die Programme, die bisher im Buch vorgekommen sind, waren sehr einfach. Jedes Programm hat eine Menge von Eingabewerten entgegengenommen, das Ergebnis ausgegeben und die Ausführung beendet. Das ist ähnlich wie bei unserem computerisierten Mechaniker, wenn wir ihn anweisen, wie eine Schraube zu lösen ist, ohne ihm die Möglichkeit zu geben, zur nächsten Schraube oder zum nächsten Reifen zu gehen.

Was in unseren Programmen fehlt, ist eine Form von Flusskontrolle. Wir haben bisher keine Möglichkeit, kleinere Tests zu machen, und können keine Entscheidungen basierend auf diesen Tests treffen.

Dieses Kapitel widmet sich den verschiedenen C++-Kommandos zur Flusskontrolle.

8.1 Das Verzweigungskommando

Die einfachste Form der Flusskontrolle ist die Verzweigung. Diese Instruktion ermöglicht es dem Computer, zu entscheiden, welcher Pfad durch C++-Instruktionen gewählt werden soll auf der Basis logischer Bedingungen. In C++ wird das Verzweigungskommando durch die `if`-Anweisung implementiert:

```
if (m > n)
{
    // Instruktionen, die ausgeführt werden,
    // wenn m größer als n ist
}
else
{
    // ... Instruktionen, die ausgeführt werden
    // wenn nicht
}
```

Zuerst wird die Bedingung $m > n$ ausgewertet. Wenn das Ergebnis wahr ist, wird die Kontrolle an die Anweisungen übergeben, die der öffnenden Klammer `{` folgen. Wenn m nicht größer ist als n , wird die Kontrolle an die Anweisungen übergeben, die der öffnenden Klammer unmittelbar nach dem `else` folgen.

Der `else`-Zweig ist optional. Wenn er nicht da ist, verhält sich C++ so, als wäre er da, aber leer.



Tatsächlich sind die Klammern optional, wenn nur eine Anweisung in einem Zweig ausgeführt werden soll; es ist aber so einfach, Fehler zu machen, die der Compiler nicht abfangen kann, wenn nicht die Klammern als Marker verwendet werden können. Es ist viel sicherer, immer Klammern zu verwenden. Wenn Ihre Freunde Sie verführen wollen, keine Klammern zu verwenden, sagen Sie einfach »NEIN«.

Das folgende Programm demonstriert die `if`-Anweisung:

```
// BranchDemo - Eingabe von zwei Zahlen. Gehe
//              einen Pfad des Programms entlang,
//              wenn das erste Argument größer ist
//              als das zweite, oder sonst den
//              anderen Pfad
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{
    // Eingabe des ersten Argumentes ...
    int nArg1;
    cout << »Eingabe nArg1: «;
    cin >> nArg1;

    // ... und des zweiten
    int nArg2;
    cout << »Eingabe nArg2: »;
    cin >> nArg2;

    // entscheide nun, was zu tun ist:
    if (nArg1 > nArg2)
    {
        cout << »nArg1 größer als nArg2\n«;
    }
    else
    {
        cout << »nArg1 nicht größer nArg2\n«;
    }

    return 0;
}
```

Lektion 8 – Kommandos zur Flusskontrolle**71**

Hier liest das Programm Integerzahlen von der Tastatur und verzweigt entsprechend. Das Programm erzeugt die folgende typische Ausgabe:

```
Eingabe nArg1: 10
Eingabe nArg1: 8
nArg1 größer als nArg2
```

8.2 Schleifenkommandos

Verzweigungskommandos ermöglichen Ihnen, den Programmfluss den einen oder den anderen Pfad hinunter zu leiten. Das ist das C++-Äquivalent dazu, den computerisierten Mechaniker entscheiden zu lassen, ob er einen Schraubenschlüssel oder einen Schraubendreher verwendet, abhängig von der Problemstellung. Das bringt uns aber noch nicht zu dem Punkt, an dem der Mechaniker den Schraubenschlüssel mehr als einmal drehen kann, mehr als eine Radmutter entfernen kann oder mehr als einen Reifen des Autos bearbeiten kann. Dafür brauchen wir Schleifenanweisungen.

8.2.1 Die while-Schleife

Die einfachste Form der Schleifenanweisung ist eine `while`-Schleife, die wie folgt aussieht:

```
while(condition)
{
    // ... wird wiederholt ausgeführt, solange
    //    condition erfüllt ist
}
```

Die Bedingung `condition` wird geprüft. Wenn sie wahr ist, dann werden die Anweisungen innerhalb der Klammern ausgeführt. Sobald die schließende Klammer angetroffen wird, wird die Kontrolle an den Anfang zurückgegeben, und der Prozess beginnt von vorne. Der Effekt ist, dass der C++-Code zwischen den Klammern so lange ausgeführt wird, wie die Bedingung wahr ist.



Die Bedingung wird nur am Anfang der Schleife überprüft. Selbst wenn die Bedingung in der Schleife nicht mehr erfüllt ist, wird die Kontrolle die Schleife nicht verlassen, bis sie wieder an den Anfang der Schleife kommt.

Wenn die Bedingung zum ersten Mal wahr ist, was wird sie dann später falsch machen? Betrachten Sie das folgende Programm.

```
// WhileDemo - Eingabe einer Schleifenanzahl.
//           Ausgabe einer Zeichenkette in jedem
//           der nArg Durchläufe.
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{
    // Eingabe der Schleifenanzahl
    int nLoopCount;
    cout << »Eingabe Schleifenanzahl: <<
```

```

cin >> nLoopCount;

// so viele Schleifen wie eingegeben
while (nLoopCount > 0)
{
    nLoopCount = nLoopCount - 1;
    cout << »Nur « << nLoopCount
        << » weitere Schleifendurchläufe\n«;
}
return 0;
}

```

WhileDemo beginnt mit der Abfrage einer Schleifenanzahl vom Benutzer, die in der Variable `nLoopCount` gespeichert wird. Wenn dies getan ist, fährt das Programm mit dem Testen von `nLoopCount` fort. Wenn `nLoopCount` größer ist als null, dann wird `nLoopCount` um eins erniedrigt, und das Ergebnis wird auf dem Bildschirm ausgegeben. Das Programm geht dann an den Anfang der Schleife zurück, um zu testen, ob `nLoopCount` immer noch positiv ist.

Wenn das Programm WhileDemo ausgeführt wird, liefert es folgende Ausgabe:

```

Eingabe Schleifenanzahl: 5
Nur 4 weitere Schleifendurchläufe
Nur 3 weitere Schleifendurchläufe
Nur 2 weitere Schleifendurchläufe
Nur 1 weitere Schleifendurchläufe
Nur 0 weitere Schleifendurchläufe

```

Als ich eine Schleifenanzahl von 5 eingegeben habe, hat das Programm die Schleife fünfmal durchlaufen und hat jedes Mal den heruntergezählten Wert ausgegeben.

Wenn der Benutzer eine negative Schleifenanzahl eingibt, überspringt das Programm die Schleife. Weil die Bedingung nie war ist, wird die Schleife nie betreten. Wenn der Benutzer eine sehr große Zahl eingibt, läuft das Programm sehr lange in der Schleife, bis es fertig ist.

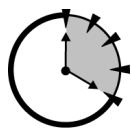


Eine andere, selten benutzte Version der while-Schleife, bekannt unter dem Namen do-while, ist identisch mit der while-Schleife, außer dass die Bedingung am Ende der Schleife getestet wird.

```

do
{
    // ... das Innere der Schleife
} while (condition);

```



20 Min.

Verwendung von Autodekrement

Das Programm dekrementiert den Schleifenzähler unter Verwendung von Zuweisungs- und Subtraktionsanweisungen. Eine kompaktere Anweisung würde Autodekrement verwenden.

Die folgende Schleife ist ein wenig einfacher als die obige.

```

while (nLoopCount > 0)
{
    nLoopCount--;
    cout << »Nur « << nLoopCount
        << » weitere Schleifendurchläufe\n«;
}

```

Die Logik in dieser Version ist die gleiche wie im Original – der einzige Unterschied ist die Art und Weise, in der `nLoopCount` dekrementiert wird.

Weil Autodekrement sowohl das Argument dekrementiert als auch dessen Wert zurückliefert, kann der Dekrementoperator mit jeder der anderen Anweisungen verknüpft werden. Z.B. ist die folgende Version die bisher kürzeste Schleifenkonstruktion:

```
while (nLoopCount-- > 0)
{
    cout << »Nur « << nLoopCount
        << » weitere Schleifendurchläufe\n«;
}
```



Tipp

Das ist die Version, die von den meisten C++-Programmierern verwendet wird.

Das ist die Stelle, wo der Unterschied zwischen Prädekrement und Postdekrement auftaucht.



Hinweis

Beide, `nLoopCount--` und `--nLoopCount` dekrementieren `nLoopCount`; der erste gibt den Wert von `nLoopCount` vor dem Dekrementieren zurück, der zweite danach.

Möchten Sie, dass die Schleife ausgeführt wird, wenn der Benutzer als Schleifenanzahl 1 eingibt? Wenn Sie die Prädekrement-Version verwenden, ist der Wert von `--nLoopCount` gleich 0 und der Rumpf der Schleife wird nie betreten. Mit der Postdekrement-Version ist der Wert von `nLoopCount--` gleich 1 und die Kontrolle wird an die Schleife übergeben.

Die gefürchtete Endlosschleife

Nehmen Sie an, dass der Programmierer einen Fehler gemacht hat, und vergessen hat, die Variable `nLoopCount` zu dekrementieren, wie im Beispiel unten zu sehen ist. Das Ergebnis ist ein Schleifen-zähler, der seinen Wert nie ändert. Die Bedingung ist entweder immer falsch oder immer wahr.

```
while (nLoopCount > 0)
{
    cout << »Nur « << nLoopCount
        << » weitere Schleifendurchläufe\n«;
}
```

Weil der Wert von `nLoopCount` sich niemals verändert, läuft das Programm in einer endlosen Schleife. Ein Ausführungspfad, der unendlich oft ausgeführt wird, wird als Endlosschleife bezeichnet. Eine Endlosschleife tritt dann auf, wenn die Bedingung, die zum Schleifenabbruch führen sollte, nie erfüllt werden kann – im Allgemeinen durch einen Programmierfehler.

Es gibt viele Wege, eine Endlosschleife zu produzieren, die meisten sind viel komplizierter als das hier dargestellte Beispiel.

8.2.2 Die for-Schleife

Eine zweite Form der Schleife ist die `for`-Schleife, die folgendes Format besitzt:

```
for (initialization; conditional; increment)
{
    // ... Body der Schleife
}
```

Die Ausführung der `for`-Schleife beginnt mit der Initialisierung. Die Initialisierung hat diesen Namen bekommen, weil dort üblicherweise Zählvariablen initialisiert werden. Die Initialisierung wird nur ein einziges Mal ausgeführt, wenn die `for`-Schleife zum ersten Mal durchlaufen wird.

Die Ausführung wird bei der Bedingung fortgesetzt. Ähnlich zu der `while`-Schleife wird die `for`-Schleife so lange ausgeführt, wie die Bedingung wahr ist.

Nachdem die Ausführung des Code im Body der `for`-Schleife abgeschlossen wurde, wird die Kontrolle an die Inkrementanweisung übergeben. Danach wird die Bedingung erneut geprüft und der Prozess wiederholt. Die Inkrementklausel enthält normalerweise eine Autoinkrement- oder Autodekrementanweisung zum Updaten der Zählvariablen.

Alle drei Klauseln sind optional. Wenn die Initialisierung oder die Inkrementklausel fehlen, ignoriert C++ sie. Wenn die Bedingung fehlt, führt C++ die Schleife unendlich oft aus (oder bis sonst etwas den Kontrollfluss abbricht).

Die `for`-Schleife kann am Beispiel besser verstanden werden. Das folgende Programm `ForDemo` ist nichts anderes als das Programm `WhileDemo`, nur dass eine `for`-Schleife verwendet wird.

```
// ForDemo - Eingabe einer Schleifenanzahl.
//           Ausgabe einer Zeichenkette in jedem
//           Durchlauf.
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{
    // Eingabe der Schleifenanzahl
    int nLoopCount;

    cout << »Eingabe Schleifenanzahl: «;
    cin >> nLoopCount;

    // so viele Schleifen wie eingegeben
    for (int i = nLoopCount; i > 0; i--)
    {
        cout << »Durchlauf für « << i << » \n«;
    }
    return 0;
}
```

Diese Version durchläuft die gleichen Schleifen wie zuvor. Der Unterschied ist jedoch, dass nicht der Wert von `nLoopCount` verändert, sondern eine Zählvariable verwendet wird.

Die Kontrolle beginnt mit der Deklaration einer Variablen `i`, die mit 1 initialisiert wird. Die `for`-Schleife überprüft dann die Variable `i`, um sicherzustellen, dass sie kleiner oder gleich dem Wert von `nLoopCount` ist. Wenn dies der Fall ist, führt das Programm die Ausgabeanweisung aus, inkrementiert `i` und fährt mit der Ausführung der Schleife fort.

Die `for`-Schleife ist auch bequem, wenn Sie von 0 bis zu einer Schleifenanzahl zählen wollen, statt von einer Schleifenanzahl auf 1 herunterzuzählen. Dies wird durch eine kleine Änderung in der Implementierung erreicht:

```
// ForDemo - Eingabe einer Schleifenanzahl.
//           Ausgabe einer Zeichenkette in jedem
//           Durchlauf.
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{
    // Eingabe der Schleifenanzahl
    int nLoopCount;
    cout << »Eingabe Schleifenanzahl: «;
    cin >> nLoopCount;

    // so viele Schleifen wie eingegeben
    for (int i = 1; i <= nLoopCount; i++)
    {
        cout << »Durchlauf für « << i << » \n«;
    }
    return 0;
}
```

Anstatt mit der Schleifenanzahl zu beginnen, startet diese Version bei 1 und zählt hoch, bis der vom Benutzer eingegebene Wert erreicht wird.



Hinweis

Die Verwendung der Variable `i` für `for`-Schleifen ist historisch bedingt (aus den frühen Tagen der Programmiersprache FORTRAN). Das ist der Grund, weshalb diese Schleifenvariablen sich nicht an die Standardkonventionen der Namensbildung halten.



Tipp

Wenn die Indexvariable innerhalb der Initialisierungsklausel deklariert wird, ist sie nur innerhalb der `for`-Schleife bekannt. C++-Programmierer sagen, dass der Gültigkeitsbereich (scope) der Variablen die `for`-Schleife ist. Im obigen Beispiel ist die Variable `i` für die `return`-Anweisung nicht zugreifbar, weil diese Anweisung nicht in der `for`-Schleife steht.

8.2.3 Spezielle Schleifenkontrolle

Es kann vorkommen, dass die Bedingung, die zum Abbruch der Schleife führen soll, nicht am Anfang noch am Ende der Schleife eintritt. Betrachten Sie das folgende Programm, das Zahlen summiert, die der Benutzer eingibt. Die Schleife bricht ab, wenn der Benutzer eine negative Zahl eingibt, d.h. die Schleife muss verlassen werden, bevor der negative Wert zu der Summe hinzugefügt wird.

Die Herausforderung dieses Problems besteht darin, dass das Programm die Schleife erst verlassen kann, wenn der Benutzer einen Wert eingegeben hat. Die Schleife muss aber verlassen werden, bevor der Wert zur Summe addiert wird.

76

Samstagmorgen

Für diese Fälle definiert C++ das Kommando `break`. Wenn `break` angetroffen wird, wird die aktuelle Schleife sofort verlassen. D.h. die Kontrolle wird dann an die Anweisung übergeben, die unmittelbar auf die schließende Klammer folgt.

Das Format der `break`-Anweisung ist wie folgt:

```
while(condition) // break geht bei for genauso
{
    if (condition2)
    {
        break; // Schleife verlassen
    }
} // Kontrolle kommt hier her, wenn
// Programm break antrifft
```

Ausgerüstet mit diesem neuen Kommando `break` sieht meine Lösung für das Additionsproblem wie das Programm `BreakDemo` aus:

```
// BreakDemo - Eingabe einer Reihe von Zahlen.
//           Bilde die Summe dieser Zahlen
//           bis der Benutzer eine negative
//           Zahl eingibt.
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{
    // initialisiere Summe
    int nAccumulator = 0;
    cout << »Dieses Programms addiert «
         << »die eingegebenen Zahlen.\n«;
    cout << »Beenden Sie die Schleife mit «
         << »einer negativen Zahl.\n«;

    // »Endlosschleife«
    for(;;)
    {
        // hole eine weitere Zahl
        int nValue = 0;
        cout << »Nächste Zahl: «;
        cin >> nValue;

        // wenn sie negativ ist ...
        if (nValue < 0)
        {
            // ... dann Abbruch
            break;
        }

        // ... sonst addiere sie zur Summe
        nAccumulator = nAccumulator + nValue;
    }
}
```


Lektion 8 – Kommandos zur Flusskontrolle

77

```
// jetzt haben wir die Schleife verlassen
// Ausgabe der Gesamtsumme
cout << »\nDie Gesamtsumme ist <<
    << nAccumulator
    << »\n«;

return 0;
}
```

Nachdem dem Benutzer die Spielregeln erklärt wurden (Eingabe einer negativen Zahl zum Abbruch usw.), durchläuft das Programm eine Endlosschleife.



Eine for-Schleife ohne Bedingung ist eine Endlosschleife.



Diese Schleife ist nicht wirklich endlos, weil sie eine break-Anweisung enthält. Trotzdem wird sie als Endlosschleife bezeichnet, da die Abbruchbedingung nicht im Kommando selber enthalten ist.

Wenn das Programm BreakDemo erst einmal in der Schleife ist, bekommt es eine Zahl über die Tastatur. Erst wenn das Programm eine Zahl gelesen hat, kann es bestimmen, ob diese Zahl zum Abbruch der Schleife führt. Wenn die eingegebene Zahl negativ ist, wird die Kontrolle an `break` übergeben, was zum Abbruch der Schleife führt. Wenn die Zahl nicht negativ ist, wird die `break`-Anweisung übersprungen und die Kontrolle wird an den Ausdruck übergeben, der diese Zahl zur Summe addiert.

Wenn das Programm erst einmal die Schleife verlassen hat, gibt es die Gesamtsumme aus, und beendet die Ausführung. Hier ist die Ausgabe einer Beispielsitzung:

```
Dieses Programms addiert die eingegebenen Zahlen.
Beenden Sie die Schleife mit einer negativen Zahl.
Nächste Zahl: 1
Nächste Zahl: 2
Nächste Zahl: 3
Nächste Zahl: 4
Nächste Zahl: -1

Die Gesamtsumme ist 10
```



Wenn Sie eine Operation wiederholt auf einer Variablen ausführen, stellen Sie sicher, dass die Variable korrekt initialisiert wurde, bevor die Schleife betreten wird. In diesem Fall setzt das Programm `nAccumulator` auf Null, bevor die Schleife betreten wird, in der Werte `nValue` zu `nAccumulator` addiert werden.

**10 Min.**

8.3 Geschachtelte Kontrollkommandos

Die drei bisherigen Programme in diesem Kapitel entsprechen den Anweisungen an den Mechaniker, wie er eine Radmutter lösen soll: den Schraubenschlüssel so lange drehen, bis die Radmutter herunterfällt. Wie funktioniert es, den Mechaniker zu instruieren, so lange Radmuttern zu entfernen, bis der Reifen herunterfällt? Dafür müssen wir geschachtelte Schleifen implementieren.

Eine Schleifenanweisung innerhalb einer anderen Schleifen wird als **geschachtelte Schleife** bezeichnet. Lassen Sie uns exemplarisch das Programm BreakDemo zu einem Programm umbauen, das eine beliebige Anzahl von Folgen summiert. In diesem Programm NestedDemo summiert die innere Schleife Werte auf, die von der Tastatur kommen, bis eine negative Zahl eingegeben wird. Die äußere Schleife läuft so lange, bis die Summe einer Sequenz gleich 0 ist.

```
// NestedDemo - Eingabe einer Reihe von Zahlen.
//           Bilde die Summe dieser Zahlen,
//           bis der Benutzer eine negative
//           Zahl eingibt. Setze den Prozess
//           fort, bis die Summe gleich 0 ist.
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{
    // die äußere Schleife
    cout << »Dieses Programm summiert mehrere \n«
    << »Reihen von Zahlen. Beenden Sie jede «
    << »Reihe mit einer negativen Zahl.\n«
    << »Beenden Sie die Eingabe der Reihen\n«
    << »so, dass die Summe gleich 0 ist; \n«;
    << »geben Sie z.B. 1, -1 ein. \n«;

    // Summieren von Zahlenreihen
    int nAccumulator;
    do
    {
        // nächste Zahlenreihe
        nAccumulator = 0;
        cout << »\Nächste Zahlenreihe\n«;

        // Endlosschleife
        for(;;)
        {
            // hole nächste Zahl
            int nValue = 0;
            cout << »Nächste Zahl: «;
            cin >> nValue;

            // wenn sie negativ ist ...
            if (nValue < 0)
            {
                // ... dann Abbruch
                break;
            }
        }
    }
}
```

```

        // ... sonst addiere sie zur Summe
        nAccumulator = nAccumulator + nValue;
    }

    // Ausgabe der Gesamtsumme
    cout << »\nDie Gesamtsumme ist «
        << nAccumulator
        << »\n«;

    // ...und beginne mit der nächsten Zahlen-
    // folge, wenn die Summe nicht null war.
    } while (nAccumulator != 0);
    cout << »Programm wird beendet.\n«;
    return 0;
}

```

8.4 Können wir switchen?

Eine letzte Kontrollanweisung ist nützlich, wenn wir eine endliche Anzahl von Fällen haben. Die `switch`-Anweisung ist wie eine zusammengesetzte `if`-Anweisung, in dem Sinne, dass sie eine Anzahl verschiedener Möglichkeiten enthält anstatt eines einzelnen Tests:

```

switch(expression)
{
    case c1:
        // gehe hierhin, wenn expression == c1
        break;
    case c2:
        // gehe hierhin, wenn expression == c2
        break;
    default:
        // ansonsten gehe hierhin
}

```



0 Min.

Der Wert von `expression` muss ein Integer sein (`int`, `long` oder `char`). Die `case`-Werte `c1`, `c2`, `c3` müssen konstant sein. Wenn die `switch`-Anweisung angetroffen wird, wird der Ausdruck ausgewertet und verglichen mit den `case`-Konstanten. Die Kontrolle wird an die `case`-Konstante übergeben, die passt. Wenn keine passende `case`-Konstante gefunden wird, geht die Kontrolle an `default` über.



Hinweis

Die `break`-Anweisungen sind nötig, um das `switch`-Kommando zu verlassen. Ohne die `break`-Anweisungen würde der Fluss von einem Fall zum nächsten fließen.

Zusammenfassung

Die einfache `if`-Anweisung ermöglicht es dem Programmierer, den Programmfluss abhängig vom Wert eines Ausdrucks den einen oder den anderen Pfad entlang fließen zu lassen. Die Schleifenanweisungen fügen die Fähigkeit hinzu, Codeblöcke wiederholt auszuführen, so lange bis eine Bedingung falsch wird. Schließlich stellt die `break`-Anweisung einen Extralevel Kontrolle bereit, die einen Schleifenabbruch an jeder beliebigen Stelle gestattet.

- Die `if`-Anweisung wertet einen Ausdruck aus. Wenn der Ausdruck nicht 0 (d.h. er ist `true`) ist, wird die Kontrolle an den Block übergeben, der dem `if` unmittelbar folgt. Wenn nicht, wird die Kontrolle an den `else`-Zweig übergeben. Wenn es keinen `else`-Zweig gibt, geht die Kontrolle auf die Anweisung über, die der `if`-Anweisung folgt.
- Die Schleifenkommandos `while`, `do while` und `for` führen einen Codeblock so lange aus, bis eine Bedingung nicht mehr wahr ist.
- Die `break`-Anweisung ermöglicht den Abbruch von Schleifen an jeder beliebigen Stelle.

In Sitzung 9 werden wir uns mit Wegen beschäftigen, wie C++-Programme durch die Verwendung von Funktionen vereinfacht werden können.

Selbsttest

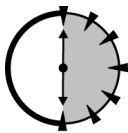
1. Ist es ein Fehler, den `else`-Zweig eines `if`-Kommandos wegzulassen? Was passiert? (Siehe »Das Verzweigungskommando«)
2. Welches sind die drei Typen von Schleifen-Kommandos? (Siehe »Schleifenkommandos«)
3. Was ist eine Endlosschleife? (Siehe »Die gefürchtete Endlosschleife«)
4. Welches sind die drei »Klauseln«, die eine `for`-Schleife ausmachen? (Siehe »Die `for`-Schleife«)

Funktionen



Checkliste

- ☒ void-Funktionen schreiben
- ☒ Funktionen mit mehreren Argumenten schreiben
- ☒ Funktionen überladen
- ☒ Funktionstemplates erzeugen
- ☒ Speicherklassen von Variablen bestimmen



30 Min.

Einige der Beispielprogramme in Sitzung 8 sind schon ein wenig fortgeschrittener, aber wir müssen weiter C++ lernen. Programme mit mehreren Schachtelungsebenen können schwer zu überblicken sein. Fügen Sie die vielfältigen und komplizierten Verzweigungen hinzu, wie sie von realen Anwendungen erwartet werden, und die Programme sind überhaupt nicht mehr nachvollziehbar.

Glücklicherweise stellt C++ eine Möglichkeit bereit, selbstständige Blöcke von Code zu separieren, die als Funktionen bezeichnet werden. In dieser Sitzung werden wir sehen, wie C++ Funktionen deklariert, erzeugt und benutzt.

9.1 Code einer Sammelfunktion

Wie so vieles, werden auch Funktionen am besten an einem Beispiel verstanden. Dieser Abschnitt beginnt mit einem Beispielprogramm FunctionDemo, das das Programm NestedDemo aus Sitzung 8 vereinfacht, indem es für einen bestimmten Teil der Logik eine Funktion definiert. Dieser Abschnitt erklärt dann, wie diese Funktion definiert wird, und wie sie aufgerufen wird, indem der Beispielcode als Muster verwendet wird.

9.1.1 Sammelcode

Das Programm NestedDemo in Sitzung 8 enthält eine innere Schleife, die eine Folge von Zahlen summiert, und eine äußere Schleife, die das wiederholt ausführt, bis sich der Benutzer dazu entschließt, das Programm zu beenden. Von der Logik her könnten wir die innere Schleife separieren, d.i. der Teil des Programms, der eine Folge von Zahlen aufsummiert, von der äußeren Schleife, die den Prozess wiederholt.

Der folgende Beispielcode zeigt das vereinfachte Programm NestedDemo, in dem die Funktion `sumSequence()` eingeführt wurde.



Den Namen von Funktionen folgen normalerweise unmittelbar ein Paar Klammern.

```
// FunctionDemo - demonstriert den Gebrauch von
//                Funktionen, indem die innere
//                Schleife von NestedDemo in eine
//                eigene Funktion gepackt wird

#include <stdio.h>
#include <iostream.h>

// sumSequence - addiere eine Reihe von Zahlen, die
//                der Benutzer über die Tastatur
//                eingibt, bis zur ersten negativen
//                Zahl - gib dann die Summe zurück
int sumSequence(void)
{
    // Endlosschleife
    int nAccumulator = 0;
    for(;;)
    {
        // hole weitere Zahl
        int nValue = 0;
        cout << »Nächste Zahl: »;
        cin >> nValue;

        // wenn sie negativ ist ...
        if (nValue < 0)
        {
            // ...dann Schleife verlassen
            break;
        }

        // ... ansonsten Zahl zur Summe addieren
        nAccumulator = nAccumulator + nValue;
    }

    // Rückgabe der Summe
    return nAccumulator;
}

int main(int nArg, char* pszArgs[])
{
    // Anfang Main
```

```

cout << »Dieses Programm summiert mehrere \n«
      << »Reihen von Zahlen. Beenden Sie jede «
      << »Reihe mit einer negativen Zahl.\n«
      << »Beenden Sie die Eingabe der Reihen\n«
      << »so, dass die Summe gleich 0 ist; \n«;
      << »geben Sie z.B. 1, -1 ein. \n«;
// summiere Folgen von Zahlen ...
int nAccumulatedValue;
do
{
    // summiere eine Zahlenreihe, die über
    // die Tastatur eingegeben wird
    cout << »\nNächste Zahlenreihe\n«;
    nAccumulatedValue = sumSequence();

    // Ausgabe der Gesamtsumme
    cout << »\nDie Gesamtsumme ist »
          << nAccumulatedValue
          << »\n«;

    // ... bis die Summe gleich 0 ist.
} while (nAccumulatedValue != 0);
cout << »Programm wird beendet.\n«;
return 0;
}                                     // Ende Main

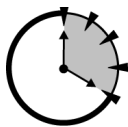
```

Aufrufen der Funktion sumSequence()

Lassen Sie uns erst einmal das Hauptprogramm ansehen, das sich zwischen den beiden Klammern befindet, die mit den beiden Kommentaren *Anfang Main* und *Ende Main* markiert sind. Dieses Codesegment sieht ähnlich aus wie Code, den wir schon einmal geschrieben haben. Die Zeile

```
nAccumulatedValue = sumSequence();
```

ruft die Funktion `sumSequence()` auf und speichert ihren Rückgabewert in der Variablen `nAccumulatedValue`. Dieser Wert wird in den folgenden drei Programmzeilen nacheinander ausgegeben. Das Programm setzt die Schleife so lange fort, bis die Summe, die von der inneren Funktion zurückgegeben wird, 0 ist, was anzeigt, dass der Benutzer das Programm beenden möchte.



20 Min.

Definition der Funktion sumSequence()

Der Codeblock, der in Zeile 13 beginnt und bis Zeile 38 geht, bildet die Funktion `sumSequence()`.

Wenn das Hauptprogramm die Funktion `sumSequence()` in Zeile 55 aufruft, geht die Kontrolle von diesem Aufruf an den Anfang dieser Funktion in Zeile 14 über. Die Programmausführung wird dort fortgesetzt.

Die Zeilen 16 bis 34 sind identisch mit denen der inneren Schleife im Programm `NestedDemo`. Nachdem das Programm diese Schleife verlassen hat, geht die Kontrolle auf die `return`-Anweisung in Zeile 37 über. Diese bewirkt einen Rücksprung zum Aufruf der Funktion in Zeile 55, zusammen mit dem Wert, der in `nAccumulatedValue` gespeichert wird. In Zeile 55 speichert das Programm den zurückgegebenen `int`-Wert in der lokalen Variable `nAccumulatedValue`, und setzt die Ausführung fort.



In diesem Fall ist der Aufruf von `sumSequence( )` ein Ausdruck, weil er einen Wert hat. Ein solcher Aufruf kann überall da verwendet werden, wo ein Ausdruck erwartet wird.

9.2 Funktion

Das Programm `FunctionDemo` demonstriert die Definition und den Gebrauch einer einfachen Funktion.

Eine **Funktion** ist ein logisch separater C++-Codeblock. Die Funktion hat die folgende allgemeine Form:

```
<return type> name(<arguments to the function>)
{
    // ...
    return <expression>;
}
```

Die Argumente einer Funktion sind Werte, die an die Funktion als Eingaben übergeben werden können. Der Rückgabewert ist ein Wert, der von der Funktion zurückgegeben wird. Z.B. im Aufruf `square(10)` ist 10 das Argument der Funktion `square( )`, der Rückgabewert ist 100.

Sowohl Argumente als auch Rückgabewerte sind optional. Wenn eines von beiden fehlt, wird stattdessen das Schlüsselwort `void` verwendet. D.h., wenn eine Funktion eine `void`-Argumentliste hat, erwartet die Funktion keine Argumente, wenn sie aufgerufen wird. Wenn der Rückgabewert `void` ist, gibt die Funktion keinen Wert an den Aufrufenden zurück.

Im Beispielprogramm `FunctionDemo` ist der Name der Funktion `sumSequence( )`, der Typ des Rückgabewertes ist `int`, und die Funktion hat keine Argumente.



Der Default-Argumenttyp einer Funktion ist `void`. Somit kann eine Funktion `int fn(void)` auch als `int fn( )` deklariert werden.

9.2.1 Warum Funktionen?

Der Vorteil von Funktionen vor anderen C++-Kontrollkommandos ist, dass sie einen Teil des Codes für einen bestimmten Zweck vom Rest des Programms separieren. Durch diese Trennung kann sich der Programmierer auf eine Funktion konzentrieren, wenn er den Code schreibt.



Eine gute Funktion kann in einem Satz beschrieben werden, der nur wenige Unds und Oders enthält. Z.B. »die Funktion `sumSequence( )` summiert eine Folge von Zahlen, die vom Benutzer eingegeben werden.« Diese Definition ist kurz und klar.

Die Funktionskonstruktion machte es mir möglich, im Wesentlichen zwei verschiedene Teile des Programms FunctionDemo zu schreiben. Ich habe mich darauf konzentriert, die Summe einer Folge von Zahlen zu erzeugen, als ich die Funktion `sumSequence()` geschrieben habe. Ich habe mir an dieser Stelle keine Gedanken um irgendeinen anderen Code gemacht, der diese Funktion aufrufen könnte.

Genauso konnte ich mich beim Schreiben der Funktion `main()` auf die Behandlung der von `sumSequence()` zurückgegebenen Werte konzentrieren. Dabei musste ich nur wissen, was die Funktion zurückgibt, aber nicht, wie sie intern arbeitet.

9.2.2 Einfache Funktionen

Die einfache Funktion `sumSequence()` gibt einen Integerwert zurück, den sie berechnet hat. Funktionen können jede Art eines regulären Variablentyps zurückgeben. Z.B. kann eine Funktion `double` oder `char` zurückgeben.

Wenn eine Funktion keinen Wert zurückgibt, dann wird der Rückgabewert dieser Funktion mit `void` angegeben.



Eine Funktion kann durch ihren Rückgabewert bezeichnet werden. Eine Funktion, die ein `int` zurückgibt, wird oft als Integer-Funktion bezeichnet. Eine Funktion, die keinen Wert zurückgibt, wird als void-Funktion bezeichnet.

Z.B. führt die folgende void-Funktion eine Operation durch, gibt aber keinen Wert zurück.

```
void echoSquare()
{
    int nValue;
    cout << »Wert:<<
    cin >> nValue;
    cout << »\nQuadrat: » << nValue * nValue << »\n<<
    return;
}
```

Die Kontrolle beginnt bei der öffnenden Klammer und wird fortgesetzt bis zur `return`-Anweisung. Die `return`-Anweisung in einer void-Funktion darf keinen Rückgabewert enthalten.



Die `return`-Anweisung in einer void-Funktion ist optional. Wenn sie nicht da ist, wird die Kontrolle an die aufrufende Funktion zurückgegeben, wenn die schließende Klammer angetroffen wird.

9.2.3 Funktionen mit Argumenten

Einfache Funktionen haben nur einen begrenzten Nutzen, da die Kommunikation dieser Funktionen durch ihren Rückgabewert wie eine Einbahnstraße ist. Kommunikation in beiden Richtungen ist aber besser; diese wird durch Argumente erreicht. Ein *Funktionsargument* ist eine Variable, deren Wert an die Funktion bei ihrem Aufruf übergeben wird.

Beispielfunktion mit Argumenten

Das folgende Beispiel definiert und benutzt eine Funktion `square( )`, die das Quadrat einer double-Gleitkommazahl zurückgibt, die ihr übergeben wurde:

```
// SquareDemo - demonstriert den Gebrauch von
// Funktionen mit Argumenten

#include <stdio.h>
#include <iostream.h>
// square - gibt das Quadrat ihres Argumentes
// dVar zurück.
double square(double dVar)
{
    return dVar * dVar;
}

int sumSequence(void)
{
    // Endlosschleife
    int nAccumulator = 0;
    for(;;)
    {
        // hole weitere Zahl
        double dValue = 0;
        cout << »Nächste Zahl: »;
        cin >> dValue;

        // wenn sie negativ ist ...
        if (dValue < 0)
        {
            // ... dann Schleife verlassen
            break;
        }

        // ... ansonsten berechne Quadrat
        int nValue = (int)square(dValue);
        nAccumulator = nAccumulator + nValue;
    }

    // Rückgabe der Summe
    return nAccumulator;
}

int main(int nArg, char* pszArgs[])
{
    // Anfang Main
    cout << »Dieses Programm summiert die Quadrate\n«
        << »von Zahlenreihen. Beenden Sie jede\n«
        << »Reihe mit einer negativen Zahl.\n«
        << »Beenden Sie die Eingabe der Reihen\n«
        << »so, dass die Summe gleich 0 ist; \n«;

    // Summiere Folgen von Zahlen ...
    int nAccumulatedValue;
    do
    {
```

```

// Summiere Quadrate einer Zahlenfolge,
// die über die Tastatur eingegeben wird
cout << »\Nächste Zahlenreihe\n«;
nAccumulatedValue = sumSequence();

// Ausgabe der Gesamtsumme
cout << »\nDie Gesamtsumme ist «
    << nAccumulatedValue
    << »\n«;

// ... bis die Summe gleich 0 ist.
} while (nAccumulatedValue != 0);
cout << »Programm wird beendet.\n«;
return 0;
}                                     // Ende Main

```

Das ist das gleiche Programm wie FunctionDemo, außer dass SquareDemo die Quadrate der eingegebenen Werte summiert.

Der Wert von dValue wird an die Funktion square() übergeben in der Zeile

```
int nValue = (int)square(dValue);
```

innerhalb der Funktion sumSequence(). Die Funktion square() multipliziert den ihr in Zeile 12 übergebenen Wert mit sich selber und gibt das Ergebnis zurück. Das Ergebnis wird in der Variable dSquare gespeichert, die in Zeile 33 zu der Summe addiert wird.

Funktionen mit mehreren Argumenten

Funktionen können mehrere Argumente haben, die durch Kommata getrennt werden. Die folgende Funktion gibt das Produkt ihrer beiden Argumente zurück:

```

int product(int nArg1, int nArg2)
{
    return nArg1 * nArg2;
}

```

Wertkonvertierung (Casten)

Zeile 32/33 des Programms SquareDemo enthält einen Operator, den wir vorher noch nicht gesehen haben.

```
nAccumulator = nAccumulator + (int)dValue;
```

Das (int) vor dValue zeigt an, dass der Programmierer möchte, dass der Wert von dValue von seinem aktuellen Typ, in diesem Fall double, nach int konvertiert wird.

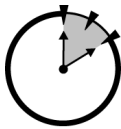
Ein solcher Cast ist eine explizite Konvertierung von einem Typ in einen anderen. Jeder numerische Typ kann in jeden anderen numerischen Typ konvertiert werden. Ohne diesen Cast hätte C++ diese Typkonvertierung selber vorgenommen, jedoch ohne eine Warnung ausgegeben.

Die Funktion main()

Es sollte klar sein, dass `main()` nichts anderes ist als eine Funktion, wenn auch eine Funktion mit merkwürdigen Argumenten.

Wenn ein Programm erzeugt wird, fügt C++ Code hinzu, der ausgeführt wird, bevor Ihr Programm überhaupt startet. Dieser Code initialisiert die Umgebung, in der Ihr Programm operiert. Z.B. öffnet dieser Code die beiden I/O-Objekte `cin` und `cout`.

Wenn die Umgebung erst einmal eingerichtet ist, ruft dieser C++-Code die Funktion `main()` auf, womit die Ausführung Ihres Codes beginnt. Wenn Ihr Programm beendet ist, wird `main()` wieder verlassen. Dies ermöglicht dem C++-Code, einige Dinge aufzuräumen, bevor die Kontrolle wieder an das Betriebssystem zurückgegeben wird, das dann das Programm löscht.

**10 Min.****9.2.4 Mehrere Funktionen mit gleichem Namen**

Zwei Funktionen in einem Programm dürfen nicht den gleichen Namen haben, sonst hat C++ keine Möglichkeit, sie zu unterscheiden, wenn sie aufgerufen werden sollen. In C++ jedoch gehören zum Namen einer Funktion die Anzahl und die Typen der Argumente. Die folgenden Funktionen sind daher voneinander verschieden:

```
void someFunction(void)
{
    // ... führe eine Funktion aus
}
void someFunction(int n)
{
    // ... führe eine andere Funktion aus
}
void someFunction(double d)
{
    // ... führe eine noch andere Funktion aus
}
void someFunction(int n1, int n2)
{
    // ... tue etwas ganz anderes
}
```

`someFunction()` ist eine Abkürzung für alle vier Funktionen, in gleicher Weise, wie Stephen eine Kurzform meines Namens ist. Ob ich nun die Kurzform zur Beschreibung der Funktion wähle, so kennt C++ doch auch die Funktionen `someFunction(void)`, `someFunction(int)`, `someFunction(double)` und `someFunction(int, int)`.



void als Argumenttyp ist optional; `someFunction(void)` und `someFunction()` sind die gleiche Funktion.

Eine typische Anwendung könnte wie folgt aussehen:

```
int nVariable1, nVariable2;    // äquivalent zu
                              // int Variable1;
                              // int Variable2;

double dVariable;

// Funktionen unterscheiden sich durch ihre
// Argumenttypen (aufgerufene Funktion im Kommentar)
someFunction();               // someFunction(void)
someFunction(nVariable1);     // someFunction(int)
someFunction(dVariable);     // someFunction(double)
someFunction(nVariable1, nVariable2);
                              // someFunction(int, int)

// das funktioniert auch mit Konstanten
someFunction(1);              // someFunction(int)
someFunction(1.0);           // someFunction(double)
someFunction(1, 2);          // someFunction(int, int)
```

In jedem der Fälle stimmt der Typ der Argumente mit dem vollen Namen der drei Funktionen überein.



Der Rückgabebetyp ist nicht Teil des Funktionsnamens. D.h., die beiden folgenden Funktionen haben den gleichen Namen und können daher nicht im gleichen Programm vorkommen:

```
int someFunction(int n);    // vollständiger Name der
double someFunction(int n); // beiden Funktionen
                          // ist someFunction(int)
```

9.3 Funktionsprototypen

In den bisherigen Beispielprogrammen wurden die Funktionen `sumSequence( )` und `square( )` beide in Codesegmenten definiert, die vor den Aufrufen der Funktionen standen. Das muss nicht immer so sein: Eine Funktion kann an beliebiger Stelle in einem Modul definiert werden. Ein *Modul* ist ein anderer Name für eine C++-Quelldatei.

Trotzdem muss jemand `main( )` den vollständigen Namen der Funktion mitteilen, bevor sie aufgerufen werden kann. Betrachten Sie den folgenden Codeschnipsel:

```
int main(int argc, char* pArgs[])
{
    someFunc(1, 2);
}
int someFunc(double dArg1, int nArg2)
{
    // ... tue etwas
}
```

Der Aufruf von `someFunc( )` von `main( )` aus kennt nicht den vollen Namen der Funktion. Von den Argumenten her könnte man vermuten, dass der Name `someFunc(int, int)` und der Rückgabebetyp `void` ist. Wie sie sehen können, ist das falsch.

Was benötigt wird, ist ein Weg, um `main()` den vollständigen Namen der Funktion `someFunc()` mitzuteilen, bevor er benutzt wird. Was gebraucht wird, ist eine Funktionsdeklaration. Eine *Funktionsdeklaration*, oder ein *Prototyp*, sieht so aus wie die Funktion, nur ohne Body. In Gebrauch sieht ein Prototyp wie folgt aus:

```
int someFunc(double, int); // Prototyp
int main(int argc, char* pArgs[])
{
    someFunc(1, 2);
}
int someFunc(double dArg1, int nArg2)
{
    // ... tue etwas
}
```

Der Aufruf in `main()` weiß nun, dass die 1 erst nach `double` gecastet werden muss, bevor die Funktion aufgerufen wird. Weiterhin weiß `main()`, dass die Funktion `someFunc()` ein `int` zurückgibt; dieser Rückgabewert wird hier jedoch ignoriert.



Tipp

C++ erlaubt es dem Programmierer, Rückgabewerte zu ignorieren.

9.4 Verschiedene Speichertypen

Es gibt drei verschiedene Orte, an denen Funktionsvariablen gespeichert werden können. Variablen, die innerhalb einer Funktion deklariert werden, sind lokal. Im folgenden Beispiel ist die Variable `nLocal` für die Funktion `fn()` lokal:

```
int nGlobal;
void fn()
{
    int nLocal;
    static int nStatic;
}
```



0 Min.

Die Variable `nLocal` existiert nicht, bis die Funktion `fn()` aufgerufen wird. Außerdem hat nur die Funktion `fn()` Zugriff auf `nLocal` – andere Funktionen können nicht in die Funktion »eindringen« und auf die Variable zugreifen.

Im Vergleich dazu existiert die Variable `nGlobal` so lange, wie das Programm läuft. Alle Funktionen haben jederzeit Zugriff auf `nGlobal`.

Die statische Variable `nStatic` ist eine Mischung aus einer lokalen und einer globalen Variable. Die Variable `nStatic` wird erzeugt, wenn die Deklaration erstmalig bei der Ausführung angetroffen wird (ungefähr dann, wenn die Funktion `fn()` aufgerufen wird). Zusätzlich ist `nStatic` nur innerhalb von `fn()` zugreifbar. Anders als `nLocal` existiert `nStatic` auch dann noch, wenn das Programm `fn()` bereits verlassen hat. Wenn `fn()` der Variablen `nStatic` einen Wert zuweist, bleibt dieser Wert erhalten, d.h. er steht auch im nächsten Aufruf der Funktion wieder zur Verfügung.



Es gibt einen vierten Variablentyp, `auto`, der aber heute die gleiche Bedeutung hat wie `local`.

Zusammenfassung

Sie sollten jetzt eine Vorstellung davon haben, wie komplex Programme werden können, und wie kleine Funktionen die Programmlogik vereinfachen können. Wohlgeformte Funktionen sind klein, haben im Idealfall weniger als 50 Zeilen, und haben weniger als 7 `if`- oder Schleifen-Kommandos. Solche Funktionen sind besser zu verstehen, und damit einfacher zu schreiben und zu debuggen. Und ist das nicht das Ziel?

- C++-Funktionen sind das Mittel, um den Code in handliche Teile zu zerlegen.
- Funktionen können beliebig viele Argumente haben, über die Werte beim Aufruf der Funktion übergeben werden.
- Funktionen können einen einzelnen Wert an den Aufrufenden zurückgeben.
- Funktionsnamen können überladen werden, wenn sie durch die Anzahl und die Typen der Argumente unterscheidbar bleiben.

Es ist sehr schön, Ihr Programm mit mehreren Ausdrücken in unterschiedlichen Variablen und auf mehrere Funktionen verteilt auszudrücken, aber das bringt alles nichts, wenn Sie das Programm nicht zum Laufen bekommen. In Sitzung 10 werden Sie die elementaren Techniken kennenlernen, um Fehler in Ihren Programmen zu finden.

Selbsttest

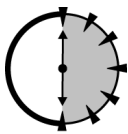
1. Wie rufen Sie eine Funktion auf? (Siehe »Aufrufen der Funktion `sumSequence( )`«)
2. Was bedeutet der Rückgabewert `void`? (Siehe »Funktion«)
3. Warum sollte man Funktionen schreiben? (Siehe »Warum Funktionen«)
4. Was ist der Unterschied zwischen einer lokalen und einer globalen Variable? (Siehe »Verschiedene Speichertypen«)



Debuggen

Checkliste

- ☒ Fehlertypen unterscheiden
- ☒ Die »Crash-Meldungen« der C++-Umgebung verstehen
- ☒ Die Ausgabetechnik des Debuggens beherrschen
- ☒ Debuggen mit Visual C++ und GNU C++



30 Min.

Sie werden sicher festgestellt haben, dass die Programme, die Sie als Teil der Übungen in früheren Kapiteln geschrieben haben, nicht beim ersten Mal funktioniert haben. In der Tat habe ich selten, wenn überhaupt schon einmal, ein nicht-triviales C++-Programm geschrieben, das nicht irgendeinen Fehler enthielt, als ich es auszuführen versuchte.

Ein Programm, das auf Anhieb funktioniert, wenn Sie es ausprobieren, wird auch Goldstern-Programm (*gold-star program*) genannt.

10.1 Fehlertypen

Es gibt zwei Typen von Fehlern. Die Fehler, die der Compiler abfangen kann, sind sehr einfach zu beheben. Der Compiler wird Sie im Allgemeinen zu dem Fehler hinführen. Manchmal ist vielleicht die Beschreibung des Fehlers nicht korrekt – es ist leicht, einen Compiler zu verwirren – aber wenn Sie Ihr C++-Paket richtig kennen, ist es nicht schwierig, die Fehler zu beheben.

Eine zweite Sorte von Fehlern umfasst die Fehler, die der Compiler nicht finden kann. Diese Fehler werden erst dann sichtbar, wenn Sie das Programm ausführen. Fehler, die erst beim Ausführen des Programms sichtbar werden, werden als *Laufzeitfehler* bezeichnet. Fehler, die der Compiler finden kann, werden als *Compilezeitfehler* bezeichnet.

Laufzeitfehler sind viel schwieriger zu finden, weil Sie keinen Anhaltspunkt dafür haben, was falsch gelaufen ist, außer Fehlerausgaben, die Ihr Programm vielleicht generiert hat.

Es gibt zwei verschiedene Techniken, um Bugs zu finden. Sie können Ausgabeanweisungen an wichtigen Stellen im Programm einbauen und das Programm neu generieren. Sie können eine Vorstellung davon bekommen, was falsch gelaufen ist, wenn diese Ausgabeanweisungen ausgeführt

werden. Ein mächtigerer Zugang ist die Verwendung eines separaten Programms, das als Debugger bezeichnet wird. Ein Debugger gibt Ihnen die Möglichkeit, Ihr Programm bei der Ausführung zu kontrollieren. In dieser Sitzung behandeln wir den Zugang über Ausgabeanweisungen. In Sitzung 16 lernen Sie den Umgang mit den Debuggern von Visual C++ und GNU C++.

10.2 Die Technik der Ausgabeanweisungen

Der Ansatz, zum Debuggen Ausgabeanweisungen in den Code einzufügen, wird als Technik der Ausgabeanweisungen bezeichnet.



Diese Technik wird oft als Schreibansatz bezeichnet. Diese Bezeichnung geht auf die frühen Tage der Programme zurück, die in FORTRAN geschrieben wurden. Ausgaben werden in FORTRAN durch WRITE-Anweisungen ausgeführt.

Um zu sehen, wie das funktionieren könnte, lassen Sie uns den Fehler in folgendem fehlerhaften Programm beheben:

```
// ErrorProgram - dieses Programm berechnet den
//                  Mittelwert von Zahlen - es enthält
//                  jedoch einen Bug
#include <stdio.h>
#include <iostream.h>

int main(int argc, char* pszArgs[])
{
    cout << »Dieses Programm muss abstürzen!\n«;

    // summiere Zahlen, bis der Benutzer eine
    // negative Zahl eingibt, dann gib den
    // Mittelwert aus
    int nSum;
    for (int nNums = 0; ;)
    {
        // eine weitere Zahl
        int nValue;
        cout << »\nNächste Zahl:«;
        cin >> nValue;

        // wenn sie negativ ist ...
        if (nValue < 0)
        {
            // ... dann gib Mittelwert aus
            cout << »\nMittelwert ist: »
                << nSum/nNums
                << »\n«;
            break;
        }
    }
}
```

```

        // nicht negativ, addierte zur Summe
        nSum += nValue;
    }
    return 0;
}

```

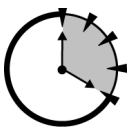
Nachdem ich das Programm eingegeben habe, erzeuge ich das Programm, um die ausführbare Datei ErrorProgram.exe zu generieren. Ungeduldig lenke ich den Windows-Explorer auf den Ordner, der das Programm enthält, und führe voll Vertrauen einen Doppelklick auf ErrorProgram aus, um das Programm laufen zu lassen. Ich gebe die Werte 1, 2 und 3 ein, gefolgt von -1 um die Eingabe abzuschließen. Aber anstatt den erwarteten Wert 2 auszugeben, beendet sich das Programm mit der nicht sehr freundlichen Fehlermeldung in Abbildung 10.1 und ohne jegliche Ausgabe.



Abbildung 10.1: Die erste Version von ErrorProgram bricht plötzlich ab, ohne eine Ausgabe zu erzeugen.



Führen Sie ErrorProgram noch nicht in der C++-Umgebung aus.



20 Min.

10.3 Abfangen von Bug Nr. 1

Die Fehlermeldung in Abbildung 10.1 zeigt den allgegenwärtigen Fehlertext »Dieses Programm wurde aufgrund eines ungültigen Vorgangs geschlossen ...«. Selbst von den Informationen über die Registerinhalte haben Sie nichts, womit Sie das Debuggen beginnen könnten.



Tatsächlich ist doch ein klein wenig Information enthalten: Oben in der Fehlermeldung steht »ERRORPROGRAM verursachte einen Teilungsfehler«. Das würde normalerweise nicht viel helfen, hier jedoch schon, weil nur eine einzige Division im Programm vorkommt. Aber lassen Sie uns diese Meldung aus diesem Grund ignorieren.

In der Hoffnung, mehr Informationen über ErrorProgram innerhalb der C++-Umgebung zu bekommen, kehre ich dahin zurück. Was dann passiert, ist bei Visual C++ und GNU C++ etwas verschieden.

10.3.1 Visual C++

Ich gebe dieselben Werte 1, 2, 3 und -1 ein, genauso wie eben, als das Programm einen Crash erlebte.



Eines der ersten Dinge, die Sie tun sollten, wenn Sie ein Problem finden wollen ist, eine Menge von Operationen zu finden, die Ihr Programm fehlerhaft verlaufen lassen. Indem Sie das Problem reproduzierbar machen, können Sie den Fehler nicht nur fürs Debuggen immer wieder neu erzeugen, sondern Sie wissen auch, wann das Problem nicht mehr auftritt.

Visual C++ erzeugt eine Ausgabe wie in Abbildung 10.2 zu sehen ist. Das Fenster zeigt an, dass das Programm fehlerhaft beendet wurde, weil durch null geteilt wurde.

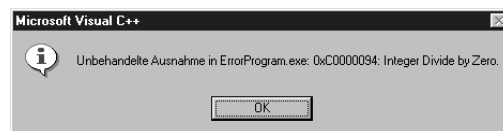


Abbildung 10.2: Die Fehlermeldung von Visual C++ ist nur wenig besser als die Windowsfehlermeldungen.

Diese Fehlermeldung ist nur ein bisschen besser als die Fehlermeldung in Abbildung 10.1. Wenn ich jedoch OK klicke, zeigt Visual C++ das Programm ErrorProgram mit einem gelben Pfeil, der auf die Division zeigt, wie in Abbildung 10.3 zu sehen ist. Das ist die C++-Zeile, in der der Fehler aufgetreten ist.

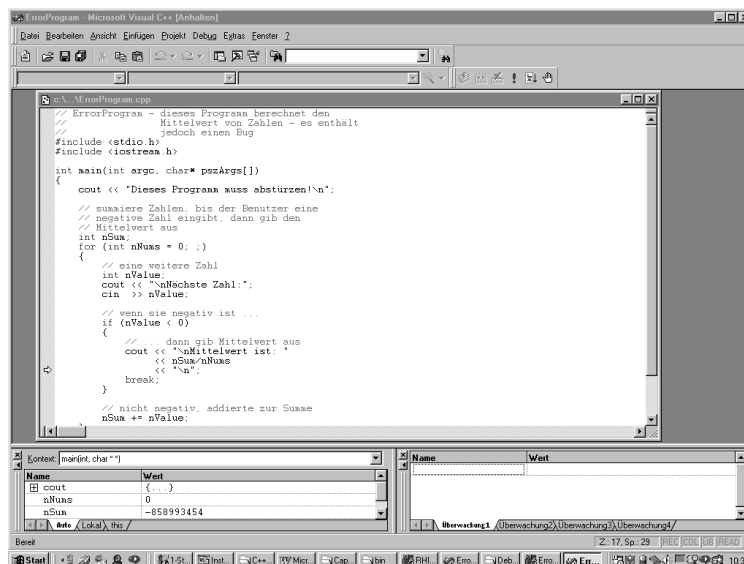


Abbildung 10.3: Visual C++ zeigt klar auf die Division durch nVals hin.

Jetzt weiß ich, dass, als die Division durchgeführt wurde, `nNums` gleich 0 gewesen sein muss. (Ansonsten kann es dort keinen Fehler durch Teilen durch null geben.) Ich kann sehen, wo `nNums` mit 0 initialisiert wird, aber wo wird es inkrementiert? Es wird nicht inkrementiert, und das ist der Fehler. Natürlich soll `nNums` in der Inkrement-Klausel der `for`-Schleife inkrementiert werden.

Um den Fehler zu beheben, ersetze ich die `for`-Schleife wie folgt:

```
for(int nNums = 0; ; nNums++);
```

10.3.2 GNU C++

Die Fehlersuche verläuft in der GNU C++-Umgebung ähnlich. Ich lasse das Programm innerhalb von `rhide` laufen. Als Antwort auf meine Eingabe 1, 2, 3 und -1, terminiert `ErrorProgram` mit der Fehlermeldung, die in Abbildung 10.4 zu sehen ist. Anstelle des normalen Exit-Codes von `0x00` (0 als Dezimalzahl) sehe ich den Fehlercode von `0xff` (255 dezimal). Das sagt mir nicht mehr, als dass ein Fehler beim Ausführen des Programms aufgetreten ist (so viel wusste ich auch vorher schon).



Abbildung 10.4: Der Fehlercode von `ErrorProgram` in `rhide` ist `0xff`.

Nach dem Klicken auf OK öffnet `rhide` ein kleines Fenster am unteren Rand des Displays. Dieses Fenster, das in Abbildung 10.5 zu sehen ist, teilt mir mit, dass der Fehler im Programm `ErrorProgram.cpp(28)` in der Funktion `main( )` aufgetreten ist. Diese etwas kryptische Fehlermeldung zeigt an, dass der Fehler in Zeile 28 von `ErrorProgram.cpp` aufgetreten ist und dass sich diese Zeile innerhalb der Funktion `main( )` befindet (Letzteres hätte ich auch durch einen Blick auf das Listing herausfinden können, aber es ist trotzdem eine schöne Information).

Von hier aus kann ich das Problem auf die gleiche Art und Weise lösen, wie im Fall von Visual C++.



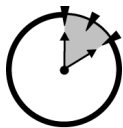
Abbildung 10.5: Die Fehlermeldung von rhide ist genauso informativ wie die von Visual C++, nur noch kryptischer.

Wie kann C++ eine Fehlermeldung an den Sourcecode binden?

Die Information, die ich erhalten haben, als ich das Programm direkt von Windows bzw. von einem MS-DOS-Fenster aus aufgerufen habe, waren nicht sehr aussagekräftig. Beide, Visual C++ und rhide, waren in der Lage, mich an die Stelle im Programm zu führen, in der der Fehler aufgetreten ist. Wie machen sie das?

C++ hat zwei Modi, ein Programm zu erzeugen. Defaultmäßig wird eine C++ im so genannten Debug-Modus erzeugt. Im Debug-Modus fügt C++ Informationen über Zeilennummern hinzu, die Zeilen im C++-Code auf Maschinenanweisungen abbilden. Diese Abbildung kann beispielsweise besagen, dass Zeile 200 des Maschinencodes von der Zeile 16 im C++-Quellcode erzeugt wurde. Als der Fehler Division-durch-null aufgetreten ist, wusste das Betriebssystem, dass der Fehler bei Offset 0x200 des ausführbaren Maschinencodes aufgetreten ist. C++ kann dies mittels der Debug-Informationen bis zur Zeile 16 des Quellcodes zurückverfolgen.

Wie Sie sich vorstellen können, benötigt die Debug-Information viel Speicher. Das macht die ausführbare Datei größer und langsamer. Zur Lösung dieses Problems stellen beide, GNU C++ und Visual C++, einen Erzeugungsmodus bereit, der keine Debug-Informationen enthält. Dieser Modus wird Release-Modus genannt.



10 Min.

10.4 Abfangen von Bug Nr. 2

Stolz auf meinen Erfolg, führe ich das Programm wieder mit der bekannten Eingabe 1, 2, 3 und -1 aus, die vorher das Programm zum Absturz gebracht hat. Diesmal stürzt das Programm zwar nicht ab, es läuft aber auch nicht. Die Ausgabe sieht merkwürdig aus:

```
Dieses Programm muss abstürzen!
Nächste Zahl: 1
Nächste Zahl: 2
Nächste Zahl: 3
Nächste Zahl: -1
Mittelwert ist: -286331151
Press any key to continue
```

Offensichtlich wurde entweder `nSum` oder `nNums` (oder beide) nicht richtig deklariert. Um fortzufahren, benötige ich den Inhalt dieser beiden Variablen. In der Tat würde es helfen, wenn ich auch den Inhalt von `nValue` kennen würde, weil `nValue` verwendet wird, um die Summe `nSum` zu berechnen.

Um die Werte von `nSum`, `nNums` und `nValue` zu erfahren, modifiziere ich die `for`-Schleife wie folgt:

```
for (int nNums = 0; ;)
{
    int nValue;
    cout << »\nNächste Zahl:<<
    cin >> nValue;
    if (nValue < 0)
    {
        cout << »\nMittelwert ist: »
            << nSum/nNums << »\n<<
        break;
    }
    // Ausgabe kritischer Informationen
    cout << »nSum = » << nSum << »\n<<
    cout << »nNums = » << nNums << »\n<<
    cout << »nValue = »<< nValue << »\n<<
    cout << »\n<<

    nSum += nValue;
}
```

Beachten Sie die hinzugefügten Ausgabeanweisungen. Diese drei Zeilen geben die Werte von `nSum`, `nNums` und `nValue` in jeder Iteration der Schleife aus.

Die Ergebnisse der Programmausführung mit der mittlerweile Standard gewordenen Eingabe 1, 2, 3 und -1 sind unten dargestellt. Schon im ersten Schleifendurchlauf scheint der Wert von `nSum` nicht korrekt zu sein. In der Tat addiert das Programm bereits beim ersten Schleifendurchlauf einen Wert zu `nSum`. An diesem Punkt würde ich erwarten, dass der Wert von `nSum` gleich 0 ist. Das scheint das Problem zu sein.

```
Dieses Programm muss abstürzen!
Nächste Zahl: 1
nSum = -858993460
nNums = 0
nValue = 1

Nächste Zahl: 2
nSum = -858993459
nNums = 1
nValue = 2

Nächste Zahl: 3
nSum = -858993457
nNums = 2
nValue = 3

Nächste Zahl:
```

Lektion 10 – Debuggen**99**

Eine genaue Untersuchung des Programms zeigt, dass `nSum` deklariert, aber nicht initialisiert wurde. Die Lösung ist, die Deklaration von `nSum` wie folgt zu verändern:

```
int nSum = 0;
```



So lange, bis eine Variable initialisiert wurde, ist der Wert der Variablen unbestimmt.

Sobald ich selbst davon überzeugt bin, dass das Ergebnis korrekt ist, räume ich das Programm wie folgt auf:

```
// ErrorProgram - dieses Programm berechnet den
//                Mittelwert von Zahlen - der Bug
//                wurde entfernt
#include <stdio.h>
#include <iostream.h>

int main(int argc, char* pszArgs[])
{
    cout << »Dieses Programm funktioniert!\n«;

    // summiere Zahlen, bis der Benutzer eine
    // negative Zahl eingibt, dann gib den
    // Mittelwert aus
    int nSum = 0;
    for (int nNums = 0; ;nNums++)
    {
        // eine weitere Zahl
        int nValue;
        cout << »\nNächste Zahl:«;
        cin >> nValue;

        // wenn sie negativ ist ...
        if (nValue < 0)
        {
            // ... dann gib Mittelwert aus
            cout << »\nMittelwert ist: »
                << nSum/nNums << »\n«;
            break;
        }

        // nicht negativ, addierte zur Summe
        nSum += nValue;
    }
    return 0;
}
```

Teil 2 – Samstagmorgen
Lektion 10

100 Samstagmorgen

Ich erzeuge das Programm neu, und teste die Folge 1, 2, 3 und -1 erneut. Diesmal sehe ich den erwarteten Mittelwert von 2:

```
Dieses Programm funktioniert!
Nächste Zahl: 1
Nächste Zahl: 2
Nächste Zahl: 3
Nächste Zahl: -1
Mittelwert ist: 2
```



0 Min.

Nachdem ich das Programm mit einer Reihe von Eingaben getestet habe, bin ich davon überzeugt, dass das Programm nun richtig ist. Ich entferne die zusätzlichen Ausgabeanweisungen und erzeuge das Programm neu, um das Debuggen des Programms abzuschließen.

Zusammenfassung

Es gibt zwei Arten von Fehlern: Compilezeitfehler, die von C++-Compiler erzeugt werden, wenn er auf eine nicht logische Code-Struktur trifft, und Laufzeitfehler, die erzeugt werden, wenn das Programm eine nicht logische Sequenz legaler Instruktionen ausführt.

Compilezeitfehler sind relativ leicht zu beheben, weil die C++-Compiler Sie direkt an die Fehlerstelle führen können. Die Umgebungen von Visual C++ und GNU C++ versuchen, Sie so gut wie möglich dabei zu unterstützen. In dem hier vorgestellten Beispielprogramm waren sie sehr erfolgreich, genau auf das Problem zu zeigen.

Wenn die Meldungen zu Laufzeitfehlern, die von der C++-Umgebung erzeugt werden, nicht ausreichen, bleibt es dem Programmierer überlassen, den Code zu debuggen. In dieser Sitzung haben wir die so genannte Technik der Ausgabeanweisungen verwendet.

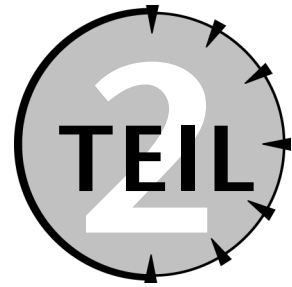
- C++-Compiler sind sehr penibel bei dem, was sie akzeptieren, um Programmierfehler nach Möglichkeit während der Programmerzeugung zu finden, in der Fehler leichter zu beheben sind.
- Das Betriebssystem versucht ebenfalls, Laufzeitfehler abzufangen. Wenn diese Fehler zu einem Programmabsturz führen, gibt das Betriebssystem Fehlerinformationen zurück, die Visual C++ und GNU C++ zu interpretieren versuchen.
- Ausgabeanweisungen, die an kritischen Stellen im Programm eingefügt werden, können den Programmierer zur Quelle des Laufzeitfehlers führen.

Obwohl sehr einfach, ist die Technik mit Ausgabeanweisungen sehr effektiv bei kleinen Programmen. Sitzung 16 zeigt Ihnen noch effektivere Debug-Techniken.

Selbsttest

1. Was sind die beiden grundlegenden Fehlertypen? (Siehe »Fehlertypen«)
2. Wie können Ausgabeanweisungen helfen, Fehler zu finden? (Siehe »Die Technik der Ausgabeanweisungen«)
3. Was ist der Debug-Modus? (Siehe Kasten »Wie kann C++ eine Fehlermeldung an den Source-code binden?«)

Samstagmorgen – Zusammenfassung



1. Ich führe die folgende Funktion aus, um kleine Tonnen in Kilogramm umzurechnen. Wenn ich den Wert 2 eingebe, erhalte ich ein falsches Ergebnis:

```
int ton2kg(int nTons)
{
    int nLongTons = nTons / 1.1;
    return 1000 * nLongTons;
}
```

a. Was ist falsch an dem Programm?

b. Welches Ergebnis erhalte ich?

Zusatz: Welche Warnung erhalte ich?

2. Stellen Sie sicher, dass Sie die Antwort zu 1 kennen (sie können spicken, wenn das nötig ist). Nun, was kann ich tun, um das Problem in 1 zu beheben?
3. Ich habe den folgenden kleinen Bruder der Funktion `ton2kg( )` geschrieben:

```
int ton2g(int nTon)
{
    return nTon * 1000000;
}
```

Weil ich mit großen Schiffen arbeite, übergebe ich der Funktion einen Wert von 5000 Tonnen. Der Wert, den ich zurückbekomme, ist offensichtlich falsch.

a. Was ist passiert?

b. Was kann ich zur Lösung des Problems tun?

4. Welche der folgenden Aussagen sind wahr?

```
int n1 = 1, n2 = -3, n3 = 10, n4 = 4
```

a. `n1 < n2`

b. `(n1 + n4) == 5`

c. `(n3 > n4) && (n4 > n1)`

d. $(n3 / 4.0) == 2.5$

e. $(n1 > n3) \ \&\& \ (-n4 > n2)$

Was ist der Wert von $n4$ nach diesen Berechnungen?

5. **Gegeben sei** $n1 = 0101\ 1101_2$

a. **Was ist das hexadezimale Äquivalent von $n1$?**

b. **Was ist der dezimale Wert von $n1$?**

c. **Was ist der Wert von $n1 * 2$ in Binärformat?**

Zusatz: Was ist der Unterschied im Bitmuster von $n1$ und $2 * n1$?

d. **Was ist der Wert von $n1 | 2$?**

e. **Was ist der Wert von $(n1 \& 2) == 0$?**

6. **(Das ist eine harte Nuss) Was ist der endgültige Wert von $n1$?**

```
int n1 = 10;
if (n1 > 11)
{
    if (n1 > 12)
    {
        n1 = 0;
    }
else
{
    n1 = 1;
}
}
```

7. **Was ist der Unterschied zwischen `while( )` und `do...while( )` im folgenden Beispielcode:**

```
int n1 = 10;
while(n1 < 5)
{
    n1++;
}

do
{
    n1++;
} while(n1 < 5);
```

8. Schreiben Sie eine Funktion `double cube(double d)` **als Ergänzung der ersten Funktion.**

9. Was passiert im folgenden Ausdruck in Anwesenheit beider Funktionen?

```
int n = cube(3.0);
```

10. Fügen Sie schließlich die Funktion `double cube(int n)` **zu den ersten beiden Funktionen hinzu. Was passiert?**

Hinweis: Schreiben Sie die Prototypdeklarationen der drei Funktionen `cube( )` **auf.**

Samstag- nachmittag

Teil 3

Lektion 11

Das Array

Lektion 12

Einführung in Klassen

Lektion 13

Einstieg C++-Zeiger

Lektion 14

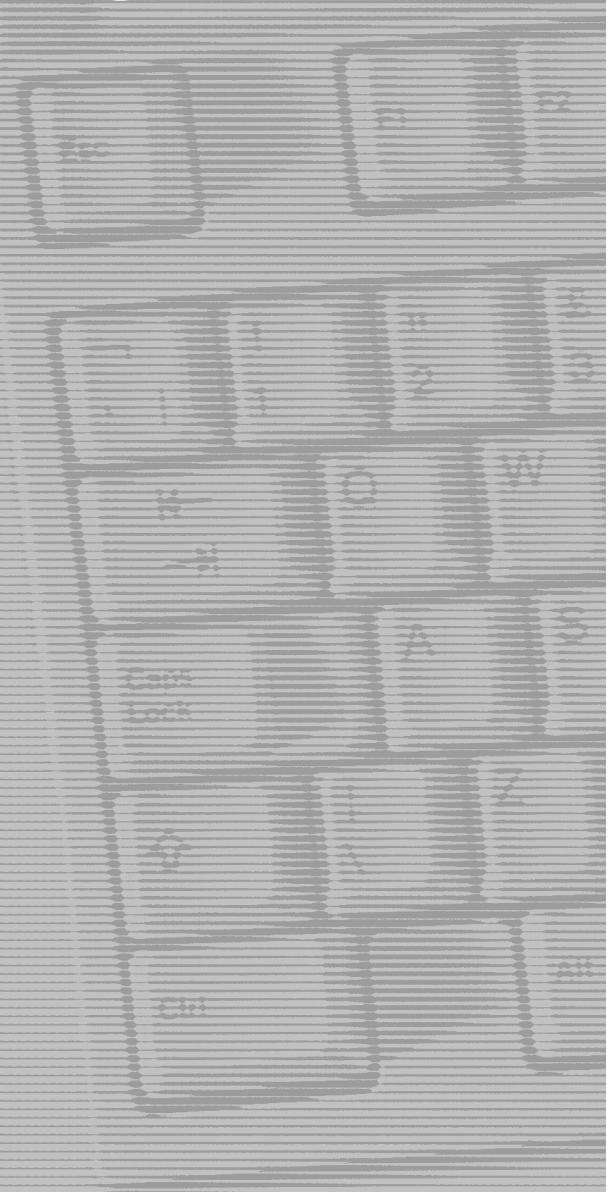
Mehr zu Zeigern

Lektion 15

Zeiger auf Objekte

Lektion 16

Debuggen II

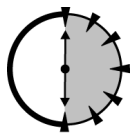


Das Array



Checkliste

- ☒ Den Datentyp Array einführen
- ☒ Arrays verwenden
- ☒ Arrays initialisieren
- ☒ Das gebräuchlichste Array verwenden – die Zeichenkette



30 Min.

Die Programme, die wir bisher geschrieben haben, waren immer nur mit einer Zahl beschäftigt. Das Summationsprogramm liest immer eine Zahl von der Tastatur, addiert sie zur bisherigen Summe, die in einer einzigen Variablen gespeichert ist, und liest die nächste Zahl. Wenn wir zu unserer ersten Analogie zurückkehren, dem menschlichen Programm, richten sich diese Programme auf jeweils nur eine Radmutter. Es gibt aber Fälle, in denen wir alle Radmuttern speichern wollen, bevor wir damit anfangen, mit ihnen zu arbeiten.

Diese Sitzung untersucht, wie Werte gespeichert werden können, gerade so, wie ein Mechaniker mehrere Radmuttern gleichzeitig halten oder aufbewahren kann.

11.1 Was ist ein Array?

Lassen Sie uns damit beginnen zu untersuchen, warum und wozu *Arrays* gut sind. Ein Array ist eine Folge von Objekten, normalerweise Zahlen, wobei jedes Objekt über einen Offset angesprochen wird.

Betrachten Sie das folgende Problem. Ich brauche ein Programm, das eine Folge von Zahlen von der Tastatur liest. Ich verwende die mittlerweile Standard gewordenen Regel, dass eine negative Eingabe die Folge abschließt. Nachdem die Zahlen eingegeben sind, und erst dann, soll das Programm die Werte in der Standardausgabe ausgeben.

106 Samstagnachmittag

Ich könnte versuchen, Werte in aufeinanderfolgenden Variablen zu speichern:

```
cin >> nV1;
if (nV1 >= 0)
{
    cin >> nV2;
    if (nV2 >= 0)
    {
        ...
    }
}
```

Wie Sie sehen, kann dieser Ansatz nicht mehr als ein paar Zahlen handhaben.

Ein Array löst das Problem viel schöner:

```
int nV;
int nValues[128];
for (int i = 0; ; i++)
{
    cin >> nV;
    if (nV < 0)
    {
        break;
    }
    nValues[i] = nV;
}
```

Die zweite Zeile des Schnipsels deklariert ein Array `nValues`. Deklarationen von Arrays beginnen mit dem Typ der Array-Elemente, in diesem Fall `int`, gefolgt von dem Namen des Array. Das letzte Element einer Array-Deklaration ist eine öffnende und eine schließende Klammer, die die maximale Anzahl Elemente enthält, die im Array gespeichert werden können. Im Sourcecode-Schnipsel ist `nValues` als Folge von 128 Integerzahlen deklariert.

Der Schnipsel liest eine Zahl von der Tastatur und speichert sie in einem Element von `nValues`. Auf ein einzelnes Element des Array wird zugegriffen über den Namen des Array, gefolgt von Klammern, die den Index enthalten. Die erste Zahl im Array ist `nValues[0]`, die zweite Zahl ist `nValues[1]` usw.



Im Gebrauch repräsentiert `nValues[i]` das *i*-te Element. Die Indexvariable *i* muss eine Zählvariable sein; d.h. *i* muss entweder `int` oder `long` sein. Wenn `nValues` ein Array von `int` ist, dann ist `nValues[i]` vom Typ `int`.

Zu weiter Zugriff

Mathematiker zählen in ihren Array von 1 ab. Das erste Element eines mathematischen Arrays ist `x(1)`. Die meisten Programmiersprachen beginnen auch bei 1. C++ beginnt das Zählen bei 0. Das erste Element eines C++-Arrays ist `nValues[0]`.



Es gibt einen guten Grund dafür, dass C++ bei 0 anfängt zu zählen, aber Sie müssen sich bis Sitzung 12 gedulden, in der Sie diesen Grund erfahren werden.

Weil die Indizierung von C++ bei 0 beginnt, ist das letzte Element eines Array mit 128 `int`-Elementen `nArray[127]`.

Unglücklicherweise führt C++ keinen Test durch, ob ein verwendeter Index im Indexbereich des Array liegt. C++ wird Ihnen bereitwillig Zugriff auf `nArray[200]` geben. Sogar `nArray[-15]` ist in C++ erlaubt.

Als Illustration stellen Sie sich bitte vor, die Abstände auf den Landstraßen werden mit Hilfe von äquidistant stehenden Strommasten gemessen. (Im Westen von Texas ist das von der Wirklichkeit nicht weit entfernt.) Lassen Sie uns diese Einheit Mastenlänge nennen. Die Straße zu mir nach Hause beginnt an der Abzweigung von der Hauptstraße und führt auf geradem Wege zu meinem Haus. Die Länge dieser Strecke beträgt exakt 9 Mastenlängen.

Wenn wir die Nummerierung bei den Masten an der Landstraße beginnen, dann ist der Mast, der meinem Haus am nächsten steht, der Mast mit der Nummer 10. Dies sehen Sie in Abbildung 1.1.

Ich kann jede Position entlang der Straße ansprechen, indem ich Masten zähle. Wenn wir Abstände auf der Hauptstraße messen, berechnen wir einen Abstand von 0. Der nächste diskrete Punkt ist eine Mastenlänge entfernt usw., bis wir zu meinem Haus kommen, 9 Mastenlängen entfernt.

Ich kann einen Abstand von 20 Mastenlängen von der Hauptstraße entfernt messen. Natürlich liegt dieser Punkt nicht auf der Straße. (Erinnern Sie sich, dass die Straße an meinem Haus endet.) Tatsächlich weiß ich nicht einmal, was Sie dort vorfinden würden. Sie könnten auf der nächsten Hauptstraße sein, auf freiem Feld, oder im Wohnzimmer meines Nachbarn. Diesen Ort zu untersuchen, ist schlimm genug, aber dort etwas abzulegen, ist noch viel schlimmer. Etwas auf freiem Feld abzulegen, ist eine Sache, aber ins Wohnzimmer meines Nachbarn einzubrechen, könnte Sie in Schwierigkeiten bringen.

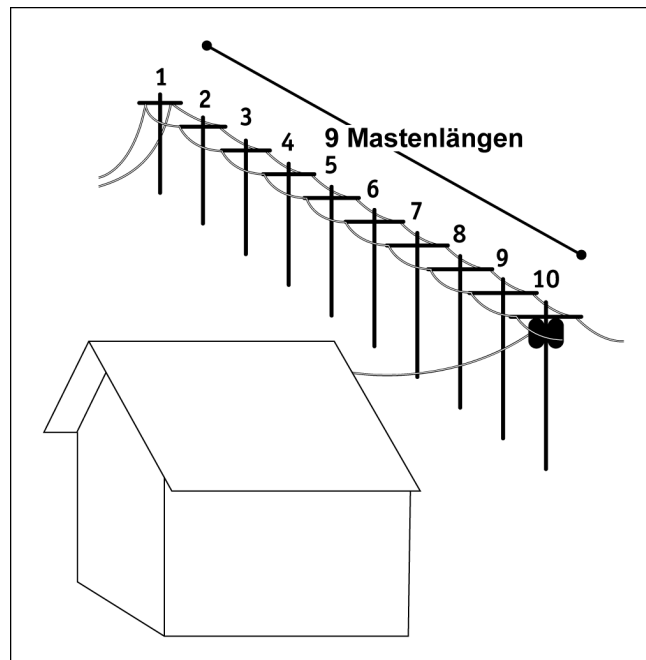


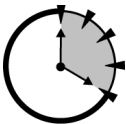
Abbildung 11.1: Man braucht 10 Masten um eine Länge von 9 Mastenlängen abzustecken.

108 Samstagnachmittag

Analog ergibt das Lesen von `array[20]` eines Array mit 10 Elementen einen mehr oder weniger zufälligen Wert. In das Element `array[20]` zu schreiben, hat ein unvorhersehbares Ergebnis. Es kann gut gehen, es kann zu einem fehlerhaften Verhalten führen oder sogar zum Absturz des Programms.



Das Element, auf das in einem Array `nArray` mit 128 Elementen am häufigsten illegal zugegriffen wird, ist `nArray[128]`. Obwohl es nur ein Element außerhalb des Arrays liegt, ist der Zugriff darauf ebenso gefährlich wie das Anfassen jeder anderen nicht korrekten Adresse.

**11.1.2 Ein Array in der Praxis**

Das folgende Programm löst das anfänglich gestellte Problem. Das Programm liest eine Folge von Integerwerten von der Tastatur ein, bis der Benutzer eine negative Zahl eingibt. Das Programm zeigt dann alle eingegebenen Zahlen und ihre Summe.

20 Min.

```
// ArrayDemo - demonstriert die Verwendung von Arrays
//             durch Einlesen von Zahlen, die dann
//             in der gleichen Reihenfolge ausgegeben
//             werden
#include <stdio.h>
#include <iostream.h>

// Prototypdeklarationen
int sumArray(int nArray[], int nSize);
void displayArray(int nArray[], int nSize);

int main(int nArg, char* pszArgs[])
{
    // Initialisierung der Summe
    int nAccumulator = 0;
    cout << »Dieses Programm summiert Zahlen,«
         << »die der Benutzer eingibt\n«;
    cout << »Beenden Sie die Schleife durch «
         << »Eingabe einer negativen Zahl\n«;

    // speichere Zahlen in einem Array
    int nInputValues[128];
    int nNumValues = 0;
    do
    {
        // hole nächste Zahl
        int nValue;
        cout << »Nächste Zahl: »;
        cin >> nValue;

        // wenn sie negativ ist...
        if (nValue < 0)    // Kommentar A
        {
            // ... dann Abbruch
```



```
        break;
    }

    // ... ansonsten speichere die Zahl
    // im Array nInputValues
    nInputValues[nNumValues++] = nValue;
} while(nNumValues < 128); // Kommentar B

// Ausgabe der Werte und der Summe
displayArray(nInputValues, nNumValues);
cout << »Die Summe ist »
    << sumArray(nInputValues, nNumValues)
    << »\n«;
return 0;
}

// displayArray - zeige die Elemente eines
//               Array der Länge nSize
void displayArray(int nArray[], int nSize)
{
    cout << »Der Wert des Array ist:\n«;
    for (int i = 0; i < nSize; i++)
    {
        cout.width(3);
        cout << i << »: » << nArray[i] << »\n«;
    }
    cout << »\n«;
}

// sumArray - gibt die Summe der Elemente eines
//            int-Array zurück
int sumArray(int nArray[], int nSize)
{
    int nSum = 0;
    for (int i = 0; i < nSize; i++)
    {
        nSum += nArray[i];
    }
    return nSum;
}
```

Das Programm ArrayDemo beginnt mit der Deklaration eines Prototyps der Funktionen `sumArray( )` und `displayArray( )`. Der Hauptteil des Programms enthält die typischen Eingabeschleifen. Dieses Mal jedoch werden die Werte im Array `nInputValues` gespeichert, wobei die Variable `nNumValues` die Anzahl der bereits im Array gespeicherten Werte enthält. Das Programm liest keine weiteren Werte ein, wenn der Benutzer eine negative Zahl eingibt (Kommentar A), oder wenn die Anzahl der Elemente im Array erschöpft ist (das ist der Test bei Kommentar B).

110 Samstagnachmittag

Das Array `nInputValues` ist als 128 Integerzahlen lang deklariert. Sie können denken, dass das genug ist für jedermann, aber verlassen Sie sich nicht darauf. Mehr Werte in ein Array zu schreiben, als es speichern kann, führt zu einem fehlerhaften Verhalten des Programms und oft zum Programmabsturz. Unabhängig davon, wie groß Sie das Array machen, bauen Sie immer einen Check ein, der sicherstellt, dass Sie nicht über die Grenzen des Array hinauslaufen.

Die Funktion `main()` endet mit der Ausgabe des Array-Inhaltes und der Summe der eingegebenen Zahlen. Die Funktion `displayArray()` enthält die typische `for`-Schleife, die zum Traversieren eines Array verwendet wird. Beachten Sie, dass der Index mit 0 und nicht mit 1 initialisiert wird. Beachten Sie außerdem, dass die `for`-Schleife abbricht, bevor `i` gleich `nSize` ist.

In gleicher Weise iteriert die Funktion `sumArray()` in einer Schleife durch das Array und addiert alle Werte zu der in `nSum` enthaltenen Summe. Nur um Nichtprogrammierer nicht weiter raten zu lassen, der Begriff »iterieren« meint das Traversieren durch eine Menge von Objekten wie das Array. Wir sagen »die Funktion `sumArray()` iteriert durch das Array.«

11.1.3 Initialisierung eines Array

Ein Array kann initialisiert werden, wenn es deklariert wird.



Eine nicht initialisierte Variable enthält einen zufälligen Wert.

Der folgende Codeschnipsel zeigt, wie das gemacht wird:

```
float fArray[5] = {0.0, 1.0, 2.0, 3.0, 4.0};
```

Dies initialisiert `fArray[0]` mit 0, `fArray[1]` mit 1, `fArray[2]` mit 2, usw. Die Anzahl der Initialisierungskonstanten kann auch gleichzeitig die Größe des Array definieren. Z.B. hätten wir durch Zählen der Konstanten feststellen können, dass `fArray` fünf Elemente hat. C++ kann auch zählen. Die folgende Deklaration ist identisch mit der obigen:

```
float fArray[] = {0.0, 1.0, 2.0, 3.0, 4.0};
```

Es ist nicht nötig, den gleichen Wert immer wieder zu wiederholen, um ein großes Array zu initialisieren. Das folgende initialisiert die 25 Einträge in `fArray` mit 1.0:

```
float fArray[25] = {1.0};
```

11.1.4 Warum Arrays benutzen?

Oberflächlich gesehen tut das Programm `ArrayDemo` nicht mehr als unser früheres Programm, das kein Array verwendete. Klar, diese Version kann ihre Eingabe wiederholend ausgeben, bevor die Summe ausgegeben wird, ist aber sehr umständlich.

Die Möglichkeit, Eingabewerte wieder anzuzeigen, weist geradezu auf einen entscheidenden Vorteil von Arrays hin. Arrays gestatten einem Programm, eine Reihe von Zahlen mehrfach zu bear-

beiten. Das Hauptprogramm war in der Lage, das Array der Eingabewerte an die Funktion `displayArray( )` zur Darstellung zu übergeben und das gleiche Array an die Funktion `sumArray( )` zur Summenbildung zu übergeben.

11.1.5 Arrays von Arrays

Arrays sind Meister darin, Folgen von Zahlen zu speichern. Einige Anwendungen erfordern Folgen von Folgen. Ein klassisches Beispiel einer Matrixkonfiguration ist die Tabelle. Ausgelegt wie ein Schachbrett erhält jedes Element der Tabelle einen x-Offset und einen y-Offset.

C++ implementiert die Matrix wie folgt:

```
int nMatrix[2][3];
```

Diese Matrix hat zwei Elemente in der einen Dimension und 3 Elemente in der anderen Dimension, also 6 Elemente. Wie Sie erwarten werden, ist eine Ecke der Matrix `nMatrix[0][0]`, während die andere Ecke `nMatrix[1][2]` ist.



Ob Sie `nMatrix` 10 Elemente lang machen in der einen oder der anderen Dimension, ist eine Frage des Geschmacks.

Eine Matrix kann in der gleichen Weise initialisiert werden, wie ein Array:

```
int nMatrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

Dies initialisiert das Array `nMatrix[0]`, das drei Elemente besitzt, mit den Werten 1, 2 und 3, und die drei Elemente des Array `nMatrix[1]` mit den Werten 4, 5 und 6.

11.2 Arrays von Zeichen

Die Elemente eines Arrays können von jedem beliebigen C++-Variablentyp sein. Arrays mit Elementen vom Typ `float`, `double` und `long` sind möglich. Arrays von `char`, d.h. Arrays von Zeichen, haben eine besondere Bedeutung.

Ein Array von Zeichen, das meinen Vornamen enthält, würde so aussehen:

```
char sMyName[] = {'S', 't', 'e', 'p', 'h', 'e', 'n'};
```

Das folgende kleine Programm gibt meinen Namen auf einem MS-DOS-Fenster aus, der Standardausgabe.

```
// CharDisplay - Ausgabe eines Zeichenarray über die
// Standardausgabe, das MS-DOS-Fenster
#include <stdio.h>
#include <iostream.h>

// Prototypdeklarationen
void displayCharArray(char sArray[], int nSize);

int main(int nArg, char* pszArgs[])
{
```

112 Samstagnachmittag

```

char cMyName[] = {'S', 't', 'e', 'p', 'h', 'e', 'n'};
displayCharArray(cMyName, 7);
cout << »\n«;
return 0;
}

// displayCharArray - gibt ein Zeichenarray
//                      Zeichen für Zeichen aus
void displayCharArray(char sArray[], int nSize)
{
    for(int i = 0; i < nSize; i++)
    {
        cout << sArray[i];
    }
}

```

Das Programm läuft gut, es ist aber unbequem, die Länge eines Array zusammen mit dem Array selber zu übergeben. Wie haben dieses Problem bei der Eingabe von Integerzahlen dadurch vermieden, dass wir die Regel aufgestellt haben, dass eine negative Zahl das Ende der Eingabe bedeuten soll. Wenn wir hier dasselbe machen könnten, müssten wir nicht die Länge des Array mit übergeben – wir würden dann wissen, dass das Array zu Ende ist, wenn dieses besondere Zeichen angetroffen wird.

Lassen Sie uns den Code 0 verwenden, um das Ende eines Zeichenarray zu markieren.



Das Zeichen, dessen Wert 0 ist, ist nicht das gleiche wie 0. Der Wert von 0 ist 0x30. Das Zeichen, dessen Wert 0 ist, wird oft als \0 geschrieben, um den Unterschied klar zu machen. In gleicher Weise ist \y das Zeichen, das den Wert y hat. Das Zeichen \0 wird auch Nullzeichen genannt.

Unter Verwendung dieser Regel sieht unser kleines Programm so aus:

```

// DisplayString - Ausgabe eines Zeichenarrays
//                      über die Standardausgabe, das
//                      MS-DOS-Fenster
#include <stdio.h>
#include <iostream.h>

// Prototypdeklarationen
void displayString(char sArray[]);

int main(int nArg, char* pszArgs[])
{
    char cMyName[] =
        {'S', 't', 'e', 'p', 'h', 'e', 'n', '\0'};
    displayString(cMyName);
    cout << »\n«;
    return 0;
}

// displayString - gibt eine Zeichenkette
//                      Zeichen für Zeichen aus
void displayString(char sArray[])

```

```

{
    for(int i = 0; sArray[i] != 0; i++)
    {
        cout << sArray[i];
    }
}

```

Die Deklaration von `cMyName` deklariert das Zeichen-Array mit dem Extrazeichen `'\0'` am Ende. Das Programm `DisplayString` iteriert durch das Zeichen-Array, bis das Nullzeichen angetroffen wird.

Die Funktion `displayString()` ist einfacher zu benutzen als ihr Vorgänger `displayCharArray()`. Es ist nicht mehr nötig, die Länge des Zeichen-Array mit zu übergeben. Weiterhin arbeitet `displayString()` auch dann, wenn die Länge der Zeichenkette zur Compilezeit nicht bekannt ist. Z.B. wäre das der Fall, wenn der Benutzer eine Zeichenkette über die Tastatur eingeben würde.

Ich habe den Begriff Zeichenkette verwendet, als wäre es ein fundamentaler Typ, wie `int` oder `float`. Als ich den Begriff eingeführt habe, habe ich auch erwähnt, dass die Zeichenkette eine Variation eines bereits existierenden Typs ist. Wie Sie jetzt sehen können, ist eine Zeichenkette ein Null-terminiertes Zeichenarray.

C++ unterstützt eine optionale, etwas bequemere Art der Initialisierung von Zeichenarrays durch eine in Hochkommata eingeschlossene Zeichenkette. Die Zeile

```
char szMyName[] = »Stephen«;
```

ist exakt äquivalent mit der Zeile

```
char cMyName[] = {'S', 't', 'e', 'p', 'h', 'e', 'n', '\0'};
```

im vorigen Beispiel.



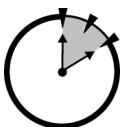
Hinweis

Die hier verwendete Namenskonvention ist nur eine Konvention, C++ kümmert sich nicht darum. Das Präfix `sz` steht für eine nullterminierte Zeichenkette.



Hinweis

Die Zeichenkette `»Stephen«` ist acht Zeichen lang, nicht sieben – das Nullzeichen nach dem `n` wird mitgezählt.



10 Min.

11.3 Manipulation von Zeichenketten

Der C++-Programmierer muss oft Zeichenketten manipulieren.

Obwohl C++ eine Reihe von Manipulationsfunktionen für Zeichenketten bereitstellt, lassen Sie uns unsere eigenen Funktionen schreiben, um ein Gefühl dafür zu bekommen, wie diese Funktionen arbeiten.

114 Samstagnachmittag**11.3.1 Unsere eigene Verbindungsfunktion**

Lassen Sie uns mit einem einfachen, wenn auch etwas längeren C++-Programm beginnen, das zwei Zeichenketten miteinander verbindet.

```
// Concatenate - verbindet zwei Zeichenketten mit
//           einem » - « in der Mitte
#include <stdio.h>
#include <iostream.h>
// die folgende Include-Datei wird für die
// für str-Funktionen benötigt
// #include <string.h>

// Prototypdeklarationen
void concatString(char szTarget[], char szSource[]);

int main(int nArg, char* pszArgs[])
{
    // lies erste Zeichenkette ...
    char szString1[256];
    cout << »Zeichenkette #1:<<
    cin.getline(szString1, 128);

    // ... nun die zweite Zeichenkette ...
    char szString2[128];
    cout << »Zeichenkette #2:<<
    cin.getline(szString2, 128);

    // ... füge » - « an die erste an ...
    concatString(szString1, » - «);
    // strcat(szString1, » - «);

    // ... füge jetzt die zweite Zeichenkette an ...
    concatString(szString1, szString2);
    // strcat(szString1, szString2);

    // ... und zeige das Ergebnis
    cout << »\n« << szString1 << »\n«;

    return 0;
}

// concatString - hängt die Zeichenkette szSource
//           ans Ende der Zeichenkette szTarget
void concatString(char szTarget[], char szSource[])
{
    // finde das Ende der ersten Zeichenkette
    int nTargetIndex = 0;
    while(szTarget[nTargetIndex])
    {
        nTargetIndex++;
    }

    // füge die zweite ans Ende der ersten an
    int nSourceIndex = 0;
```

```

while(szSource[nSourceIndex])
{
    szTarget[nTargetIndex] =
        szSource[nSourceIndex];
    nTargetIndex++;
    nSourceIndex++;
}

// füge terminierende Nullzeichen an
szTarget[nTargetIndex] = '\0';
}

```

Die Funktion `main( )` liest zwei Zeichenketten mittels der Funktion `getline( )`.



Die Alternative `cin >> szString` liest bis zum ersten Leerraum. Hier wollen wir bis zum Ende der Zeile lesen.

Die Funktion `main( )` verbindet die beiden Zeichenketten mittels der Funktion `concatString( )` und gibt dann das Ergebnis aus.

Die Funktion `concatString( )` setzt das zweite Argument, `szSource`, an das Ende des ersten Argumentes, `szTarget`.

Die erste Schleife innerhalb von `concatString( )` iteriert durch die Zeichenkette `szTarget` so lange, bis `nTargetIndex` auf der Null am Ende der Zeichenkette steht.



Die Schleife `while(value != 0)` ist das Gleiche wie `while(value)`, weil `value` als falsch interpretiert wird, wenn es gleich 0 ist, und als wahr wenn es ungleich 0 ist.

Die zweite Schleife iteriert durch die Zeichenkette `szSource` und kopiert die Elemente dieser Zeichenkette in `szTarget`, beginnend mit dem ersten Zeichen von `szSource` und dem Nullzeichen in `szTarget`. Die Schleife endet, wenn `nSourceIndex` auf dem Nullzeichen von `szSource` steht.

Die Funktion `concatString( )` setzt ein abschließendes Nullzeichen ans Ende der Ergebniszeichenkette, bevor sie zurückkehrt.



Vergessen Sie nicht, die Zeichenketten, die Sie in Ihren Programmen selber erzeugen, durch ein Nullzeichen abzuschließen. In der Regel werden Sie dadurch feststellen, dass Sie das vergessen haben, dass die Zeichenkette »Müll« am Ende enthält, oder dadurch, dass das Programm abstürzt, wenn Sie versuchen, die Zeichenkette zu manipulieren.

116 Samstagnachmittag

Das Ergebnis der Programmausführung sieht wie folgt aus:

```
Zeichenkette #1:Die erste Zeichenkette
Enter string #2:DIE ZWEITE ZEICHENKETTE
Die erste Zeichenkette - DIE ZWEITE ZEICHENKETTE
Press any key to continue
```



Es ist sehr verführerisch, C++-Anweisungen wie die folgende zu schreiben:

```
char dash[] = » - »;
concatString(dash, szMyName);
```

Das funktioniert so nicht, weil dash nur vier Zeichen zur Verfügung hat. Die Funktion wird ohne Zweifel über das Ende des Array dash hinausgehen.

11.3.2 Funktionen für C++-Zeichenketten

C++ stellt wesentliche Funktionalitäten für Zeichenketten in den Stream-Funktionen >> und << bereit. Sie werden einige dieser Funktionalitäten in Sitzung 28 sehen. Auf einem Basislevel stellt C++ eine Menge einfacher Funktionen bereit, die in Tabelle 11-1 zu sehen sind.

Tabelle 11-1: C++-Bibliotheksfunktionen für die Manipulation von Zeichenketten

Name	Operation
<code>int strlen(string)</code>	Gibt die Anzahl der Zeichen einer Zeichenkette zurück
<code>void strcat(target, source)</code>	Fügt die source-Zeichenkette ans Ende der target-Zeichenkette an
<code>void strcpy(target, source)</code>	Kopiert eine Zeichenkette in einen Puffer
<code>int strstr(source1, source2)</code>	Findet das erste Vorkommen von source2 in source1
<code>int strcmp(source1, source2)</code>	Vergleicht zwei Zeichenketten
<code>int stricmp(source1, source2)</code>	Vergleicht zwei Zeichenketten, ohne Groß- und Kleinschreibung zu beachten

Im Programm Concatenate hätten wir den Aufruf von `concatString( )` durch einen Aufruf von `strcat( )` ersetzen können, was uns ein wenig Aufwand erspart hätte.

```
strcat(szString, » - »);
```



Sie müssen die Anweisung `#include <string.h>` am Anfang Ihres Programms einfügen, das die Funktionen `str...` verwendet.

11.3.3 Wide Character

Der C++-Standardtyp `char` ist ein 8-Bit-Feld, das in der Lage ist, Werte von 0 bis 255 darzustellen. Es gibt 10 Ziffern, 26 kleine Buchstaben und 26 große Buchstaben. Selbst wenn die verschiedenen Umlaute und sonstigen Sonderzeichen hinzugefügt werden, ist immer noch genügend Platz, um auch noch das römische und das kyrillische Alphabet unterzubringen.

Probleme mit dem `char`-Typ treten erst dann auf, wenn Sie damit beginnen, asiatische Zeichen, insbesondere japanische und chinesische, darzustellen. Es gibt buchstäblich Tausende dieser Symbole – viel mehr, als sich mit 8 Bit darstellen lassen.

C++ enthält Support für einen neueren Zeichentyp `char` oder `wide character`. Obwohl dies kein elementarer Datentyp wie `char` ist, behandeln ihn mehrere C++-Funktionen, als wäre er das. Z.B. vergleicht `wstrchr()` zwei Mengen von `wchar`. Wenn Sie internationale Anwendungen schreiben und auch die asiatischen Sprachen unterstützen wollen, müssen Sie diese `wide-character`-Funktionen verwenden.

11.4 Obsolete Ausgabefunktionen

C++ stellt auch eine Menge von I/O-Funktionen auf einem niedrigen Level bereit. Die nützlichste ist die Ausgabefunktion `printf()`.



Das sind die originalen I/O-Funktionen von C. Streameingabe und -ausgabe kamen erst mit C++.

In seiner einfachsten Form gibt `printf()` eine Zeichenkette auf `cout` aus.

```
printf(»Dies ist eine Ausgabe auf cout«);
```

Die Funktion `printf()` führt Ausgaben unter Verwendung eingebetteter Kommandos zur Formatkontrolle durch, die jeweils mit einem `%`-Zeichen beginnen. Z.B. gibt das Folgende den Wert einer `int`-Zahl und einer `double`-Zahl aus:

```
int nInt = 1;
double dDouble = 3.5;
printf(»Der int-Wert ist %i; der float-Wert ist %f«,
      nInt, dDouble);
```

Der Integerwert wird an der Stelle von `%i` eingefügt, der `double`-Wert erscheint an der Stelle von `%f`.

```
Der int-Wert ist 1; der float-Wert ist 3.5
```



0 Min.

Obwohl die Funktion `printf()` kompliziert in ihrer Benutzung ist, stellt Sie doch eine Kontrolle über die Ausgabe dar, die man mit Streamfunktionen nur schwer erreichen kann.

118 Samstagnachmittag**Zusammenfassung**

Das Array ist nichts anderes als eine Folge von Variablen. Jede dieser Variablen, die alle den gleichen Typ haben, wird über einen Arrayindex angesprochen – wie z.B. Hausnummern die Häuser in einer Straße bezeichnen. Die Kombination von Arrays und Schleifen, wie `for` und `while`, ermöglichen es Programmen, eine Menge von Elementen einfach zu bearbeiten. Das bekannteste C++-Array ist das Null-terminierte Zeichenarray, das auch als Zeichenkette bezeichnet wird.

- Arrays ermöglichen es Programmen, schnell und effizient durch eine Anzahl von Elementen durchzugehen, unter Verwendung eines C++-Schleifenkommandos. Der Inkrementteil einer `for`-Schleife z.B. ist dafür gedacht, einen Index heraufzuzählen, während der Bedingungsteil dafür gedacht ist, auf das Ende des Array zu achten.
- Zugriff auf Elemente außerhalb der Grenzen eines Array ist gleichermaßen häufig anzutreffen wie gefährlich. Es ist verführerisch, auf das Element 128 eines Array mit 128 Elementen zuzugreifen. Weil die Zählung jedoch bei 0 startet, hat das letzte Element den Index 127 und nicht 128.
- Das Abschließen eines Zeichenarrays durch ein spezielles Zeichen erlaubt es Funktionen, zu wissen, wann das Array zu Ende ist, ohne ein spezielles Feld für die Zeichenlänge mitzuführen. Hierfür verwendet C++ das Zeichen `'\0'`, das den Bitwert 0 hat, und kein normales Zeichen ist. Programmierer verwenden den Begriff *Zeichenkette* oder *ASCII-Zeichenkette* für eine Null-terminiertes Zeichenarray.
- Der im Abendland entwickelte 8-Bit-Datentyp `char` kann die vielen tausend Spezialzeichen, die in einigen asiatischen Sprachen vorkommen, nicht darstellen. Um diese Zeichen zu verwalten, stellt C++ den speziellen Datentyp `wide character` bereit, der als `wchar` bezeichnet wird. C++ enthält spezielle Funktionen, die mit `wchar` arbeiten können; diese sind in der Standardbibliothek von C++ enthalten.

Selbsttest

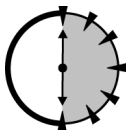
1. Was ist die Definition eines Array? (Siehe »Was ist ein Array«)
2. Was ist der Offset des ersten und des letzten Elementes im Array `myArray[128]`? (Siehe »Zu weiter Zugriff«)
3. Was ist eine Zeichenkette? Was ist der Typ einer Zeichenkette? Was beendet eine Zeichenkette? (Siehe »Arrays von Zeichen«)

Einführung in Klassen



Checkliste

- ☒ Klassen verwenden, um Variablen verschieden Typs in einem Objekt zu gruppieren
- ☒ Programme schreiben, die Klassen verwenden



30 Min.

Arrays sind großartig, wenn es darum geht, eine Folge von Objekten zu verarbeiten, wie z.B. `int`-Zahlen oder `double`-Zahlen. Arrays arbeiten nicht wirklich gut, wenn Daten verschiedenen Typs gruppiert werden sollen, wie z.B. die Sozialversicherungsnummer und der Name einer Person. C++ stellt hierfür eine Struktur bereit, die als Klasse bezeichnet wird.

12.1 Gruppieren von Daten

Viele der Programme in den vorangegangenen Sitzungen lesen eine Reihe von Zahlen, manche in Arrays, bevor diese verarbeitet werden. Ein einfaches Array ist für allein stehende Zahlen großartig. Es ist jedoch so, dass oft (wenn nicht sogar fast immer) Daten in Form von gruppierten Informationen auftreten. Z.B. könnte ein Programm den Benutzer nach seinem Vornamen, Namen und der Sozialversicherungsnummer fragen – nur in dieser Verbindung machen diese Werte Sinn. Aus einem Grund, der in Kürze klar werden wird, nenne ich eine solche Gruppierung ein *Objekt*.

Eine bestimmte Art, ein Objekt zu beschreiben, nenne ich *parallele Arrays*. In diesem Zugang definiert der Programmierer ein Array von Zeichenketten für den Vornamen, ein zweites für den Nachnamen und ein drittes für die Sozialversicherungsnummer. Diese drei Werte werden über den Index koordiniert.

120 Samstagnachmittag**12.1.1 Ein Beispiel**

Das folgende Programm benutzt parallele Arrays, um eine Reihe von Vornamen, Nachnamen und Sozialversicherungsnummern einzugeben und anzuzeigen. `szFirstName[i]`, `szLastName[i]` und `nSocialSecurity[i]` sollen gemeinsam ein Objekt bilden.

```
// ParallelData - speichert zusammengehörende Daten
//                in parallelen Arrays
#include <stdio.h>
#include <iostream.h>
#include <string.h>

// »parallele Arrays« speichern zusammengehörende
// Daten (sind global, damit alle Funktionen Zugriff
// darauf haben)
char szFirstName[25][128];
char szLastName [25][128];
int  nSocialSecurity[25];

// getData - lies einen Namen und eine
//           Sozialversicherungsnummer
//           gib 0 zurück, wenn nichts mehr
//           gelesen werden soll
int getData(int index)
{
    cout << »\nVorname:<<
    cin  >> szFirstName[index];

    // wenn der Vorname 'ende' oder 'ENDE' ist ...
    if ((strcmp(szFirstName[index], »ende«) == 0)
        ||
        (strcmp(szFirstName[index], »ENDE«) == 0))
    {
        // ... gib Kennung für Ende zurück
        return 0;
    }

    // Lade den Rest des Objektes
    cout << »Nachname:<<
    cin  >> szLastName[index];

    cout << »Sozialversicherungsnummer:<<
    cin  >> nSocialSecurity[index];

    return 1;
}

// displayData - gibt den »index«-ten Datensatz aus
void displayData(int index)
{
    cout << szFirstName[index]
         << » »
         << szLastName[index]
         << »/<<
         << nSocialSecurity[index]
```

Lektion 12 – Einführung in Klassen 121

```

        << >>\n<<;
    }

    int main(int nArg, char* pszArgs[])
    {
        // lade Vornamen, Nachnamen und
        // Sozialversicherungsnummer
        cout << >>Lese Vornamen, Nachnamen und\n<<
            << >>Sozialversicherungsnummer\n<<;
            << >>Geben Sie 'ende' als Vorname ein, \n<<;
            << >>um das Programm zu beenden\n<<;
        int index = 0;
        while (getData(index))
        {
            index++;
        }

        cout << >>\nEinträge:\n<<;
        for (int i = 0; i < index; i++)
        {
            displayData(i);
        }
        return 0;
    }

```

Die drei koordinierten Arrays sind wie folgt deklariert.

```

char szFirstName[25][128];
char szLastName[25][128];
int sSocialSecurity[25];

```

Die drei Arrays bieten Platz für jeweils 25 Einträge. Vorname und Nachname sind auf 128 Zeichen begrenzt.



Es werden keine Überprüfungen gemacht, die sicherstellen, dass das 128-Zeichen-Limit nicht überschritten wird. In realen Applikationen ist es nicht akzeptabel, auf solche Überprüfungen zu verzichten.

Die Funktion `main()` liest erst die Objekte in der Schleife ein, die mit `while(getData(index))` in der Funktion `main()` beginnt. Der Aufruf von `getData()` liest den nächsten Eintrag. Die Schleife wird verlassen, wenn `getData()` eine 0 zurückgibt, die anzeigt, dass der Eintrag vollständig ist.

Das Programm ruft dann `displayData()` auf, um die eingegebenen Objekte auszugeben.

Die Funktion `getData()` liest Daten von `cin` in die drei Arrays. Die Funktion gibt 0 zurück, wenn der Benutzer einen Vornamen `ende` oder `ENDE` eingibt. Wenn der Vorname davon verschieden ist, werden die verbleibenden Daten gelesen und es wird eine 1 zurückgegeben, um anzuzeigen, dass noch weitere Objekte gelesen werden sollen.

122 Samstagnachmittag

Die folgende Ausgabe stammt von einem Beispiellauf von ParallelData.

```
Lese Vornamen, Nachnamen und
Sozialversicherungsnummer
Geben Sie 'ende' als Vorname ein,
um das Programm zu beenden

Vorname:Stephen
Nachname:Davis
Sozialversicherungsnummer:1234

Vorname:Scooter
Nachname:Dog
Sozialversicherungsnummer:3456

Vorname:Valentine
Nachname:Puppy
Sozialversicherungsnummer:5678

Vorname:ende

Einträge:
Stephen Davis/1234
Scooter Dog/3456
Valentine Puppy/5678
```

12.1.2 Das Problem

Der Zugang der parallelen Arrays ist eine Lösung für das Problem, Daten zu gruppieren. In vielen älteren Programmiersprachen gab es keine Alternative dazu. Für große Datenmengen wird das Synchronisieren möglicherweise vieler Arrays zu einem Problem.

Das einfache Programm ParallelData muss sich nur um drei Arrays kümmern. Denken Sie an die Datenmenge, die eine Kreditkarte pro Datensatz beanspruchen würde. Es würden sicherlich Dutzende von Arrays benötigt.

Ein zweites Problem ist, dass es für einen pflegenden Programmierer nicht offensichtlich ist, dass die Arrays zusammengehören. Wenn der Programmierer nur auf einigen Arrays ein Update durchführt, werden die Daten korrupt.

12.2 Die Klasse

Was benötigt wird, ist eine Struktur, die alle Daten speichern kann, die benötigt werden, um ein einzelnes Objekt zu beschreiben. Ein einzelnes Objekt würde dann den Vornamen, den Nachnamen und die Sozialversicherungsnummer enthalten. C++ verwendet eine solche Struktur, die als Klasse bezeichnet wird.

Lektion 12 – Einführung in Klassen 123

12.2.1 Das Format einer Klasse

Eine Klasse zur Speicherung eines Vornamens, Namens und einer Sozialversicherungsnummer könnte so aussehen:

```
// die Datensatzklasse
class NameDataSet
{
    public:
        char szFirstName[128];
        char szLastName [128];
        int  nSocialSecurity;
};
// eine einzelne Instanz
NameDataSet nds;
```

Eine Klassendefinition beginnt mit dem Schlüsselwort `class`, gefolgt von dem Namen der Klasse und einem öffnenden/schließenden Klammernpaar.



Das alternative Schlüsselwort `struct` kann auch verwendet werden. Die beiden Schlüsselwörter `class` und `struct` sind identisch, mit der Ausnahme, dass bei `struct` eine `public`-Deklaration angenommen wird.

Die erste Zeile innerhalb der Klammern ist das Schlüsselwort `public`.



Spätere Sitzungen werden die anderen Schlüsselwörter von C++ außer `public` vorstellen.

Nach dem Schlüsselwort `public` kommen die Einträge, die für die Beschreibung eines Objektes benötigt werden. Die Klasse `NameDataSet` enthält den Vor- und Nachnamen zusammen mit der Sozialversicherungsnummer. Erinnern Sie sich daran, dass eine Klassendeklaration alle Daten umfasst, die ein Objekt beschreiben.

Die letzte Zeile deklariert die Variable `nds` als einen einzelnen Eintrag der Klasse `NameDataSet`. Programmierer sagen, dass `nds` eine Instanz der Klasse `NameDataSet` ist. Sie instanziierten die Klasse `NameDataSet`, um das Objekt `nds` zu erzeugen. Schließlich sagen die Programmierer, dass `szFirstName` und die anderen Elemente oder Eigenschaften der Klasse sind.

Die folgende Syntax wird für den Zugriff auf die Elemente eines Objektes verwendet:

```
NameDataSet nds;
nds.nSocialSecurity = 10;
cin >> nds.szFirstName;
```

Hierbei ist `nds` eine Instanz der Klasse `NameDataSet` (d.h. ein bestimmtes `NameDataSet`-Objekt). Die Integerzahl `nds.nSocialSecurity` ist eine Eigenschaft des Objektes `nds`. Der Typ von `nds.nSocialSecurity` ist `int`, während der Typ von `nds.szFirstName` gleich `char[]` ist.

124 Samstagnachmittag

Ein Klassenobjekt kann bei seiner Erzeugung wie folgt initialisiert werden:

```
NameDataSet nds = {»Vorname«, »Nachname«, 1234};
```

Der Programmierer kann auch Arrays von Objekten deklarieren und initialisieren:

```
NameDataSet ndsArray[2] = {{»VN1«, »NN1«, 1234}
                             {»VN2«, »NN2«, 5678}};
```

Nur der Zuweisungsoperator ist für Klassenobjekte defaultmäßig definiert. Der Zuweisungsoperator weist dem Zielobjekt eine binäre Kopie des Quellobjektes zu. Beide Objekte müssen denselben Typ besitzen.

12.2.2 Beispielprogramm

Die Klassen-basierte Version des Programms ParallelData sieht wie folgt aus:

```
// ClassData - speichert zusammengehörende Daten
//              in einem Array von Objekten
#include <stdio.h>
#include <iostream.h>
#include <string.h>

// NameDataSet - speichert Vorname, Name und
//              Sozialversicherungsnummer
class NameDataSet
{
public:
    char szFirstName[128];
    char szLastName [128];
    int  nSocialSecurity;
};

// getData - lies einen Namen und eine
//           Sozialversicherungsnummer
//           gib 0 zurück, wenn nichts mehr
//           gelesen werden soll
int getData(NameDataSet& nds)
{
    cout << »\nVorname:<«;
    cin  >> szFirstName[index];

    // wenn der Vorname 'ende' oder 'ENDE' ist ...
    if ((strcmp(nds.szFirstName, »ende«) == 0)
        ||
        (strcmp(nds.szFirstName, »ENDE«) == 0))
    {
        return 0;
    }

    cout << »Nachname:<«;
    cin  >> nds.szLastName;

    cout << »Sozialversicherungsnummer:<«;
    cin  >> nds.nSocialSecurity;
```


Lektion 12 – Einführung in Klassen 125

```

    return 1;
}

// displayData - gib »index«-ten Datensatz aus
void displayData(NameDataSet& nds)
{
    cout << nds.szFirstName
        << » »
        << nds.szLastName
        << »/«
        << nds.nSocialSecurity
        << »\n«;
}

int main(int nArg, char* pszArgs[])
{
    // alloziere 25 Datensätze
    NameDataSet nds[25];

    // lade Vornamen, Nachnamen und
    // Sozialversicherungsnummer
    cout << »Lies Vornamen, Nachnamen und \n«
        << »Sozialversicherungsnummer\n«;
    << »Geben Sie 'ende' als Vorname ein, \n«;
    << »um das Programm zu beenden\n«;
    int index = 0;
    while (getData(nds[index]))
    {
        index++;
    }

    cout << »\nEinträge:\n«;
    for (int i = 0; i < index; i++)
    {
        displayData(nds[i]);
    }
    return 0;
}

```

In diesem Fall alloziert die Funktion `main( )` 25 Objekte aus der Klasse `NameDataSet`. Wie zuvor betritt `main( )` eine Schleife, in der Einträge von der Tastatur gelesen werden unter der Verwendung der Methode `getData( )`. Statt einen einfachen Index zu übergeben (oder einen Index und ein Array), übergibt `main( )` das Objekt, das `getData(NameDataSet)` mit Werten füllen soll.

In gleicher Weise verwendet `main( )` die Funktion `displayData(NameDataSet)`, um alle `NameDataSet`-Objekte anzuzeigen.

Die Funktion `getData( )` liest die Objektinformation in das übergebene Objekt vom Typ `NameDataSet`, das `nds` heißt.



Die Bedeutung des Und-Zeichens (&), das dem Argument der Funktion `getData( )` hinzugefügt wurde, wird in Sitzung 13 vollständig erklärt. An dieser Stelle reicht es aus zu erwähnen, dass dieses Zeichen dafür sorgt, dass Änderungen, die in `getData( )` ausgeführt werden, auch in `main( )` sichtbar sind.

126

Samstagnachmittag



0 Min.

12.2.3 Vorteile

Die grundlegende Struktur des Programms `ClassData` ist die gleiche wie die von `ParallelData`. `ClassData` muss jedoch nicht mehrere Arrays verwalten. Bei einem Objekt, das so einfach ist wie `NameDataSet`, ist es nicht offensichtlich, dass dies ein entscheidender Vorteil ist. Überlegen Sie sich einmal, wie beide Programme aussehen würden, wenn `NameDataSet` alle Einträge enthalten würde, die für eine Kreditkarte benötigt würden. Je größer die Objekte, desto größer der Vorteil.



Hinweis

Je weiter wir die Klasse `NameDataSet` entwickeln, desto größer wird der Vorteil.

Zusammenfassung

Arrays können nur Folgen von Objekten gleichen Typs speichern; z.B. ein Array für `int` oder `double`. Die Klasse ermöglicht es dem Programmierer, Daten mit verschiedenen Typen in einem Objekt zu gruppieren. Z.B. könnte eine Klasse `Student` eine Zeichenkette für den Namen des Studenten, eine Integervariable für seine Immatrikulationsnummer und eine Gleitkommazahl für seinen Notendurchschnitt enthalten. Die Kombination von Array und Klassenobjekten kombiniert die Vorteile eines jeden in einer einzelnen Datenstruktur.

- Die Elemente eines Klassenobjektes müssen nicht vom selben Typ sein; wenn Sie verschieden sind, müssen sie über ihren Namen und nicht über einen Index angesprochen werden.
- Das Schlüsselwort `struct` kann an Stelle des Schlüsselwortes `class` verwendet werden; `struct` ist ein Überbleibsel aus den Tagen von C.

Selbsttest

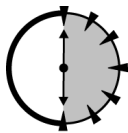
1. Was ist ein Objekt? (Siehe »Gruppieren von Daten«)
2. Was ist der ältere Begriff für das C++-Schlüsselwort `class`? (Siehe »Das Format einer Klasse«)
3. Was bedeuten die kursiv geschriebenen Worte? (Siehe »Das Format einer Klasse«)
 - a. *Instanz* einer Klasse
 - b. *Instanzieren* einer Klasse
 - c. *Element* einer Klasse

Einstieg C++-Zeiger



Checkliste

- ☒ Variablen im Speicher adressieren
- ☒ Den Datentyp Zeiger einführen
- ☒ Die inhärente Gefahr von Zeigern erkennen
- ☒ Zeiger an Funktionen übergeben
- ☒ Objekte frei allozieren, bekannt als Heap



30 Min.

Teil II führte die C++-Operatoren ein. Operationen wie Addition, Multiplikation, bitweises AND und logisches OR wurden auf elementaren Datentypen wie `int` und `float` ausgeführt. Es gibt einen anderen Variablentyp, den wir noch betrachten müssen – Zeiger.

Für jemanden, der mit anderen Programmiersprachen vertraut ist, sieht C++ bis jetzt aus wie jede andere Programmiersprache. Viele Programmiersprachen enthalten nicht die dargestellten logischen Operatoren und C++ bringt seine eigene Semantik mit, aber es gibt keine Konzepte in C++, die nicht in anderen Sprachen auch vorhanden wären. Bei der Einführung von Zeigern in die Sprache verabschiedet sich C++ von anderen, konventionelleren Sprachen.



Zeiger wurden tatsächlich im Vorgänger von C++, der Sprache C eingeführt. Alles in diesem Kapitel geht in C genauso.

Zeiger sind nicht nur ein Segen. Während Zeiger C++ einmalige Fähigkeiten verleihen, können sie syntaktisch kompliziert sein und sind eine häufige Fehlerquelle. Dieses Kapitel führt den Variablentyp Zeiger ein. Es beginnt mit einigen konzeptionellen Definitionen, geht durch die Syntax von Zeigern, und stellt dann einige Probleme dar, die Programmierer mit Zeigern haben können.

128 Samstagnachmittag**13.1 Was ist deine Adresse?**

Wie in der Aussage »jeder muss irgendwo sein«, ist jede Variable irgendwo im Speicher des Computers vorhanden. Der Speicher ist aufgeteilt in einzelne Bytes, wobei jedes Byte seine eigene Adresse hat, 0, 1, 2, 3 usw.



Es gilt die Konvention, Speicheradressen hexadezimal zu schreiben.

Verschiedene Variablentypen benötigen unterschiedlich viele Bytes im Speicher. Tabelle 13-1 zeigt den Speicher, der von den Variablentypen in Visual C++ 6 und GNU C++ auf einem Pentium-Prozessor benötigt wird.

Tabelle 13-1: Speicherbedarf verschiedener Variablentypen

Variablentyp	Speicherbedarf [Bytes]
int	4
long	4
float	4
double	8

Betrachten Sie das folgende Testprogramm Layout, das die Anordnung der Variablen im Speicher illustriert. (Ignorieren Sie den Operator & – es ist ausreichend zu sagen, dass &n die Adresse der Variablen n zurückgibt.)

```
// Layout - dieses Programm versucht einen Eindruck
//          davon zu vermitteln, wie Variablen im
//          Speicher angeordnet sind
#include <stdio.h>
#include <iostream.h>

int main(int nArgc, char* pszArgs[])
{
    int    m1;
    int    n;
    long   l;
    float  f;
    double d;
    int    m2;

    // setze Ausgabemodus auf hexadezimal
    cout.setf(ios::hex);
```

```
// gib die Adresse jeder Variable aus
// in der obigen Reihenfolge, um einen
// Eindruck von ihrer Größe zu bekommen
cout << >>-- = 0x<< << (long)&m1 << >>\n<<;
cout << >>&n = 0x<< << (long)&n << >>\n<<;
cout << >>&l = 0x<< << (long)&l << >>\n<<;
cout << >>&f = 0x<< << (long)&f << >>\n<<;
cout << >>&d = 0x<< << (long)&d << >>\n<<;
cout << >>-- = 0x<< << (long)&m2 << >>\n<<;

return 0;
}
```

Die Ausgabe dieses Programms finden Sie in Listing 13-1. Sie können sehen, dass die Variable `n` an der Adresse `0x65fdf0` gespeichert wurde.



Machen Sie sich keine Sorgen, wenn die Werte, die von Ihrem Programm ausgegeben werden, davon verschieden sind. Nur der Abstand zwischen den Adressen ist entscheidend.

Aus dem Vergleich der Adressen können wir ableiten, dass die Größe von `n` gleich 4 Bytes ist (`0x65fdf4 – 0x65fdf0`), die Größe der `long`-Variablen `l` ebenfalls gleich 4 ist (`0x65fdf0 – 0x65fdec`) usw.

Das ist aber nur dann so richtig, wenn wir annehmen können, dass die Variablen unmittelbar hintereinander im Speicher angeordnet werden, was bei GNU C++ der Fall ist. Für Visual C++ ist dafür eine besondere Projekteinstellung nötig.

Listing 13-1: Ausgaben des Programms Layout

```
-- = 0x65fdf4
&n = 0x65fdf0
&l = 0x65fdec
&f = 0x65fde8
&d = 0x65fde0
-- = 0x65fddc
```

Es gibt nichts in der Definition von C++, das vorschreiben würde, dass die Variablen wie in Listing 13-1 angeordnet sein müssen. Was uns betrifft ist es ein großer Zufall, dass GNU C++ und Visual C++ die gleiche Anordnung der Variablen im Speicher gewählt haben.

13.2 Einführung in Zeigervariablen

Lassen Sie uns mit einer neuen Definition und einer Reihe neuer Operatoren beginnen. Eine Zeigervariable ist eine Variable, die eine Adresse enthält, im Allgemeinen die Adresse einer anderen Variable. Zu dieser Definition gehören die neuen Operatoren in Tabelle 13-2.

130 Samstagnachmittag**Tabelle 13-2: Operatoren für Zeiger**

Operator	Bedeutung
& (unär)	die Adresse von
* (unär)	(in einem Ausdruck) das, worauf gezeigt wird (in einer Deklaration) Zeiger auf

Die Verwendung dieser Features wird am besten in einem Beispiel deutlich:

```
void fn()
{
    int nInt;
    int* pnInt;

    pnInt = &nInt;    // pnInt zeigt nun auf nInt
    *pnInt = 10;      // speichert 10 in int-Platz,
                     // auf den pnInt zeigt
}
```

Die Funktion `fn( )` beginnt mit der Deklaration von `nInt`. Die nächste Anweisung deklariert eine Variable `pnInt`, die vom Typ »Zeiger auf `int`« ist.



*Zeigervariablen werden wie normale Variablen deklariert, mit Ausnahme des Zeichens *. Dieses Zeichen kann irgendwo zwischen dem Namen des Basistyps, in diesem Falle `int`, und dem Variablennamen stehen. Es wird jedoch zunehmend üblich, den Stern an das Ende des Variablentyps zu setzen.*



Wie die Namen von `int`-Variablen mit `n` beginnen, ist der erste Buchstabe von Zeigervariablen `p`. Somit ist `pnX` ein Zeiger auf eine `int`-Variable `X`. Auch das ist nur eine Konvention, um den Überblick zu behalten – C++ kümmert sich nicht um die Namen, die Sie verwenden.

In einem Ausdruck bedeutet der unäre Operator `&` »Adresse von«. Somit würden wir die erste Zuweisung lesen als »speichere die Adresse von `nInt` in `pnInt`«.

Um es noch konkreter zu machen, nehmen wir an, dass der Speicherbereich von `fn( )` bei Adresse `0x100` beginnt. Lassen Sie uns weiterhin annehmen, dass `nInt` an Adresse `0x102` und `pnInt` an Adresse `0x106` steht. Die Anordnung hier ist einfacher als die Ausgabe des Programms Layout, aber die Konzepte sind identisch.

Die erste Zuweisung wird in Abbildung 13.1 dargestellt. Hier können Sie sehen, dass der Wert von `&nInt` (`0x102`) in `pnInt` gespeichert wird.

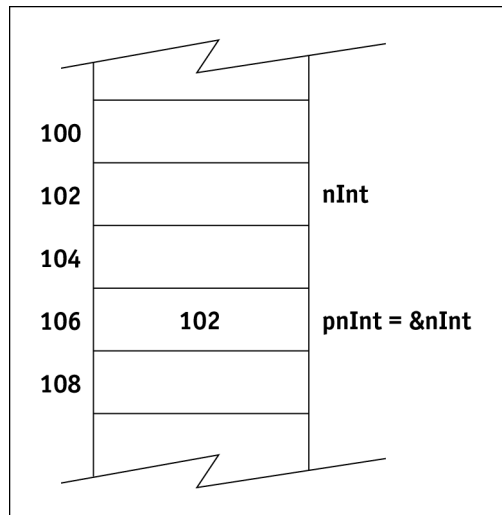


Abbildung 13.1: Speichern der Adresse von nInt in pnInt.

Die zweite Zuweisung in dem kleinen Programmschnipsel besagt, »speichere 10 in der Adresse, auf die pnInt zeigt«. Abbildung 13.2 demonstriert dies. Der Wert 10 wird an der Adresse gespeichert, die pnInt enthält; diese ist 0x102 (die Adresse von nInt).

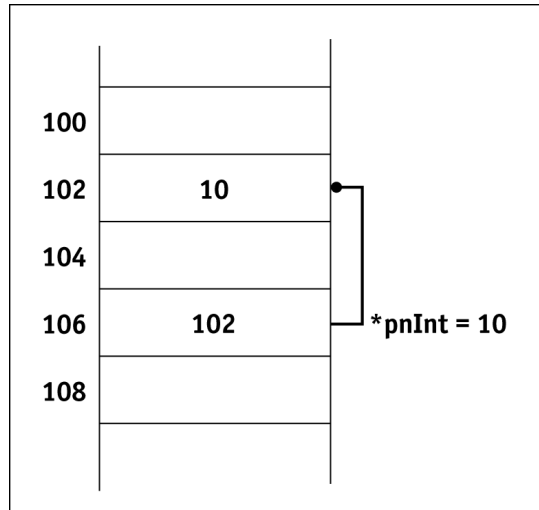
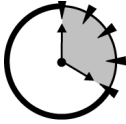


Abbildung 13.2: Speichern von 10 in der Adresse, auf die pnInt zeigt.

132 Samstagnachmittag**20 Min.****13.3 Typen von Zeigern**

Erinnern Sie sich daran, dass jeder Ausdruck einen Typ und einen Wert hat. Der Typ des Ausdrucks `&nInt` ist ein Zeiger auf Integer, der als `int*` geschrieben wird. Ein Vergleich dieses Ausdruckstyps mit der Deklaration von `pnInt` zeigt, dass die Typen exakt zueinander passen.

```
pnInt = &nInt; // beide Seiten sind vom Typ int*
```

In gleicher Weise ist der Typ von `*pnInt` gleich `int`, weil `pnInt` vom Typ `int*` ist.

```
*pnInt = 10; // beide Seiten sind vom Typ int
```

Sprachlich ausgedrückt ist der Typ von etwas, auf das `pnInt` verweist, gleich `int`.

**Hinweis**

Abgesehen davon, dass eine Zeigervariable einen unterschiedlichen Typ hat, wie `int*` und `double*`, ist der Zeiger an sich ein elementarer Typ. Unabhängig, auf was er zeigt, benötigt ein Zeiger auf einem Rechner mit Pentiumprozessor 4 Bytes.

Es ist extrem wichtig, dass Typen zusammenpassen. Bedenken Sie, was passieren könnte, wenn das Folgende erlaubt wäre:

```
int n1;
int* pnInt;
pnInt = &n1;
*pnInt = 100.0;
```

Die zweite Zuweisung versucht, einen 8-Bit-Wert 100.0 von Typ `double` in den 4 Bytes zu speichern, die von der Variable `n1` belegt werden. Das Ergebnis ist, dass Variablen in der Nähe ausgelöscht werden. Das wird grafisch im folgenden Programm `LayoutError` dargestellt, das Sie in Listing 13-2 finden. Die Ausgabe dieses Programms finden Sie am Ende des Listings.

```
// LayoutError - demonstriert das Ergebnis, wenn
// Zeiger falsch verwendet werden
#include <stdio.h>
#include <iostream.h>

int main(int nArgc, char* pszArgs[])
{
    int    upper = 0;
    int    n      = 0;
    int    lower = 0;

    // gib die Werte der Variablen vorher aus ...
    cout << »darüber = » << upper << »\n«;
    cout << »n      = » << n      << »\n«;
    cout << »darunter = » << lower << »\n«;

    // nun speichere einen double-Wert im Speicher,
    // der für ein int alloziert wurde
    cout << »\ndouble wird zugewiesen\n«;
    double* pD = (double*)&n;
```


Lektion 13 – Einstieg C++-Zeiger 133

```

    *pD = 13.0;

    // Ausgabe des Ergebnisses
    cout << »upper = » << upper << »\n«;
    cout << »n    = » << n    << »\n«;
    cout << »lower = » << lower << »\n«;

    return 0;
}

```

Ausgabe:

```

darüber  = 0
n         = 0
darunter = 0

double wird zugewiesen
darüber  = 1076494336
n         = 0
darunter = 0
Press any key to continue

```

Die ersten drei Zeilen in `main( )` deklarieren drei `int`-Variablen auf die bekannte Weise. Wir nehmen hier an, dass diese drei Variablen hintereinander im Speicher angeordnet werden.

Die drei folgenden Zeilen geben die Werte der drei Variablen aus. Es ist nicht überraschend, dass alle Variablen Null sind. Die Zuweisung `*pD = 13.0;` speichert den `double`-Wert 13.0 in der `int`-Variablen `n`. Die drei Ausgabezeilen zeigen die Werte der Variablen nach dieser Zuweisung.

Nachdem der `double`-Wert 13.0 der `int`-Variablen `n` zugewiesen wurde, ist `n` unverändert, aber die benachbarte Variable `upper` ist mit »Müll« gefüllt.

**Das Folgende castet einen Zeiger von einem Typ in einen anderen:**

```
double* pD = (double*)&n;
```

Hierbei ist `&n` vom Typ `int*`, wohingegen die Variable `pD` vom Typ `double*` ist. Der Cast `(double*)` ändert den Typ des Werts `&n` in den Wert von `pD`, in gleicher Weise castet

```
double d = (double)n;
```

den `int`-Wert in `n` in einen `double`-Wert.

13.4 Übergabe von Zeigern an Funktionen

Einer der Nutzen von Zeigervariablen ist die Übergabe von Argumenten an Funktionen. Um zu verstehen, warum das wichtig ist, müssen Sie verstehen, wie Argumente an Funktionen übergeben werden.

13.4.1 Wertübergabe

Sie werden bemerkt haben, dass es normalerweise unmöglich ist, den Wert einer Variablen innerhalb der Funktion zu ändern, an die die Variable übergeben wurde. Betrachten Sie das folgende Code-segment:

134 Samstagnachmittag

```

void fn(int nArg)
{
    nArg = 10;
    // der Wert von nArg ist hier 10
}

void parent(void)
{
    int n1 = 0;
    fn(n1);
    // der Wert von n1 ist hier 0
}

```

Die Funktion `parent( )` initialisiert die `int`-Variable `n1` mit 0. Der Wert von `n1` wird an `fn( )` übergeben. Bei Eintritt in die Funktion ist `nArg` gleich 10. `fn( )` ändert den Wert von `nArg`, bevor sie zu `parent( )` zurückkehrt. Vielleicht überrascht es Sie, dass der Wert von `n1` bei Rückkehr zu `parent( )` immer noch 0 ist.

Der Grund dafür ist, dass C++ nicht die Variable selbst an die Funktion übergibt. Stattdessen übergibt C++ den Wert, den die Variable zum Zeitpunkt des Aufrufes hat. D.h. der Ausdruck wird ausgewertet, selbst wenn es nur ein Variablenname ist, und das Ergebnis wird übergeben. Den Wert einer Variable an eine Funktion zu übergeben, wird Wertübergabe genannt.



Wenn man locker sagt »übergib die Variable `x` an die Funktion `fn( )`«, meint man damit eigentlich »übergib den Wert des Ausdrucks `x`«.

13.4.2 Übergabe von Zeigerwerten

Wie jeder andere elementare Typ, wird ein Zeigerargument als Wert übergeben.

```

void fn(int* pnArg)
{
    *pnArg = 10;
}

void parent(void)
{
    int n = 0;
    fn(&n);      // das übergibt die Adresse von n
                // der Wert von n ist jetzt 10
}

```

In diesem Fall wird die Adresse von `n` an die Funktion `fn( )` übergeben und nicht der Wert von `n`. Der entscheidende Unterschied ist offensichtlich, wenn Sie die Zuweisung in `fn( )` betrachten.

Lassen Sie uns zu unserem früheren Beispiel zurückkehren. Nehmen Sie an, dass `n` an Adresse `0x102` gespeichert ist. Nicht der Wert 10, sondern der »Wert« `0x106` wird durch den Aufruf `fn(&n)` übergeben. Innerhalb von `fn( )` speichert die Zuweisung `*pnArg = 10` den Wert 10 in der `int`-Variablen, die sich an Adresse `0x102` befindet, wodurch der Wert 0 überschrieben wird. Bei Rückkehr zu `parent( )` ist der Wert von `n` gleich 10, weil `n` nur ein anderer Name für `0x102` ist.

13.4.3 Referenzübergabe

C++ stellt eine kürzere Methode bereit, Variablen zu übergeben, ohne Zeiger verwenden zu müssen. Im folgenden Beispiel wird die Variable `n` als Referenz übergeben. Bei der *Referenzübergabe* gibt die Funktion `parent()` eine Referenz auf die Variable anstelle ihres Wertes. Referenz ist ein anderes Wort für Adresse.

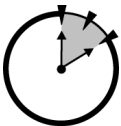
```
void fn(int& nArg)
{
    nArg = 10;
}

void parent(void)
{
    int n = 0;
    fn(n);          // Übergabe von n als Referenz
                   // hier hat n den Wert 10
}
```

In diesem Fall wird eine Referenz auf `n` an `fn()` übergeben und nicht der Wert von `n`. Die Funktion `fn()` speichert den Wert 10 an der `int`-Stelle, die durch `nArg` referenziert wird.



Beachten Sie, dass Referenz kein wirklicher Typ ist. Somit ist der volle Funktionsname `fn(int)` und nicht `fn(int&)`.



10 Min.

13.5 Heap-Speicher

Genauso wie es möglich ist, einen Zeiger an eine Funktion zu übergeben, ist es für eine Funktion möglich, einen Zeiger zurückzugeben. Eine Funktion, die die Adresse von einem `double` zurückgibt, würde wie folgt deklariert:

```
double* fn(void);
```

Man muss allerdings sehr vorsichtig mit der Rückgabe von Zeigern sein. Um zu verstehen warum, müssen Sie etwas mehr über Geltungsbereiche von Variablen wissen.

13.5.1 Geltungsbereich

C++-Variablen haben zusätzlich zu ihrem Wert und ihrem Typ eine Eigenschaft, die als Geltungsbereich bezeichnet wird. Der Geltungsbereich ist der Bereich, in dem die Variable definiert ist. Betrachten Sie den folgenden Codeschnipsel:

```
// die folgende Variable kann von allen
// Funktionen verwendet werden und ist
// so lange definiert, wie das Programm
// läuft (globaler Geltungsbereich)
int nGlobal;

// die folgende Variabale nChild ist nur in der
// Funktion verfügbar und existiert nur so lange,
```

136 Samstagnachmittag

```
// wie C++ die Funktion child( ) ausführt oder eine
// Funktion, die von child( ) aufgerufen wird
// (Funktionsgeltungsbereich)
void child(void)
{
    int nChild;
}

// die folgende Variable nParent hat
// Funktionsgeltungsbereich
void parent(void)
{
    int nParent = 0;
    fn();

    int nLater = 0;
    nParent = nLater;
}

int main(int nArgs, char* pArgs[])
{
    parent();
}
```

Die Ausführung beginnt bei `main( )`. Die Funktion `main( )` ruft sofort `parent( )` auf. Die erste Sache, die der Prozessor in `parent( )` sieht, ist die Deklaration von `nParent`. An dieser Stelle betritt `nParent` seinen Geltungsbereich – d.h. `nParent` ist definiert und steht im Rest der Funktion `parent( )` zur Verfügung.

Die zweite Anweisung in `parent( )` ist der Aufruf von `child( )`. Auch `child( )` deklariert eine lokale Variable, diesmal `nChild`. Die Variable `nChild` ist innerhalb des Geltungsbereiches von `child( )`. Technisch gesehen ist `nParent` nicht innerhalb des Geltungsbereiches von `child( )`, weil `child( )` keinen Zugriff auf `nParent` hat. Die Variable `nParent` existiert aber weiterhin.

Wenn `child( )` verlassen wird, verlässt die Variable `nChild` ihren Geltungsbereich. `nChild` wird dadurch nicht nur nicht mehr zugreifbar, sondern existiert gar nicht mehr. (Der Speicher, der von `nChild` belegt wurde, wird an einen allgemeinen Pool zurückgegeben, um für andere Dinge verwendet zu werden.)

Wenn `parent( )` mit der Ausführung fortfährt, betritt die Variable `nLater` bei ihrer Deklaration ihren Geltungsbereich. Wenn `parent( )` zu `main( )` zurückkehrt, verlassen beide, `nParent` und `nLater` ihren Geltungsbereich.

Der Programmierer kann Variablen außerhalb jeder Funktion deklarieren. Eine globale Variable ist eine Variable, die außerhalb jeder Funktion deklariert ist. Eine solche Variable bleibt in ihrem Geltungsbereich für die gesamte Dauer des Programms.

Weil `nGlobal` in diesem Beispiel global deklariert wurde, steht diese Variable allen drei Funktionen zur Verfügung und bleibt für die gesamte Dauer des Programms verfügbar.

13.5.2 Das Geltungsbereichsproblem

Das folgende Codesegment kann ohne Fehler kompiliert werden, aber arbeitet nicht korrekt:

```
double* child(void)
{
    double dLocalVariable;
    return &dLocalVariable;
}

void parent(void)
{
    double* pdLocal;
    pdLocal = child();
    *pdLocal = 1.0;
}
```

Das Problem ist, dass `dLocalVariable` nur innerhalb des Geltungsbereiches der Funktion `child( )` definiert ist. Somit existiert die Variable gar nicht mehr, wenn die Adresse von `dLocalVariable` von `child( )` zurückgegeben wird. Der Speicher, der von `dLocalVariable` belegt wurde, wird möglicherweise bereits für etwas anderes verwendet.

Das ist ein häufiger Fehler, weil er sich auf verschiedene Arten einschleichen kann. Unglücklicherweise lässt dieser Fehler das Programm nicht sofort anhalten. In der Tat kann das Programm in den meisten Fällen korrekt weiterarbeiten, so lange, wie der vorher von `dLocalVariable` belegte Speicher nicht anders verwendet wird. Solche sprunghaften Probleme sind am schwierigsten zu lösen.

13.5.3 Die Heap-Lösung

Das Problem mit dem Geltungsbereich tritt auf, weil C++ den lokal definierten Speicher freigibt, bevor der Programmierer fertig ist. Was benötigt wird, ist ein Speicherbereich, der vom Programmierer verwaltet wird. Der Programmierer kann Speicher allozieren, und ihn auch wieder zurückgeben wenn er mag. Solch ein Speicherbereich wird Heap genannt.

Der *Heap* ist ein Segment des Speichers, das explizit vom Programm kontrolliert wird. Heapspeicher wird über das Kommando `new` alloziert, gefolgt von dem Typ des Objektes, das alloziert werden soll. Z.B. alloziert das folgende eine `double`-Variable vom Heap:

```
double* child(void)
{
    double* pdLocalVariable = new double;
    return pdLocalVariable;
}
```

Obwohl die Variable `pdLocalVariable` ihren Gültigkeitsbereich verlässt, wenn die Funktion `child( )` zurückkehrt, passiert dies nicht mit dem Speicher, auf den `pdLocalVariable` verweist.

Eine Speicherstelle, die von `new` zurückgegeben wird, verlässt ihren Gültigkeitsbereich erst, wenn sie explizit an den Heap mittels den Kommandos `delete` zurückgegeben wird.

```
void parent(void)
{
    // child() gibt die Adresse eines Blocks
    // Speicher vom Heap zurück
    double* pdMyDouble = child();
}
```

138 Samstagnachmittag

```
// speichere dort einen Wert
*pdMyDouble = 1.1;

// ...

// gib jetzt den Speicher an den Heap zurück
delete pdMyDouble;
pdMyDouble = 0;

// ...
}
```

**0 Min.**

Hierbei wird der Zeiger, der von `child4` zurückgegeben wird, zur Speicherung eines `double`-Wertes verwendet. Nachdem die Funktion mit der Speicherstelle fertig ist, wird sie an den Heap zurückgegeben. Den Zeiger nach dem `delete` auf 0 zu setzen ist nicht notwendig, aber eine gute Idee. Wenn der Programmierer versehentlich versucht, etwas in `*pdMyDouble` zu speichern, nachdem bereits `delete` ausgeführt wurde, stürzt das Programm sofort ab.



Ein Programm, das sofort abstürzt, wenn ein Fehler aufgetreten ist, ist viel einfacher zu korrigieren, als ein Programm, das im Falle eines Fehlers sprunghaft ist.

Zeiger auf Funktionen

Zusätzlich zu den Zeigern auf elementare Typen ist es möglich, Zeiger auf Funktionen zu deklarieren. Z.B. deklariert die folgende Zeile einen Zeiger auf eine Funktion, die ein `int`-Argument übernimmt und einen `double`-Wert zurückgibt.

```
double (*pFN)(int);
```

Es gibt eine Vielzahl von Einsatzmöglichkeiten solcher Variablen; die Feinheiten der Funktionszeiger gehen jedoch über den Umfang dieses Buches hinaus.

Zusammenfassung

Zeigervariablen sind ein mächtiger, wenn auch gefährlicher Mechanismus, um auf Objekte über ihre Speicheradresse zuzugreifen. Das ist wahrscheinlich das entscheidende Feature, das die Dominanz von C, und später von C++, gegenüber anderen Programmiersprachen erklärt.

- Zeigervariablen werden durch Hinzufügen eines `'*'` zum Variablentyp deklariert. Der Stern kann irgendwo zwischen dem Variablennamen und dem Basistyp stehen. Es ist aber am sinnvollsten, den Stern ans Ende des Variablentyps zu setzen.
- Der Operator `&` liefert die Adresse eines Objektes zurück, während der Operator `*` das Objekt zurückgibt, auf das eine Adressen- oder Zeigervariable verweist.

Lektion 13 – Einstieg C++-Zeiger 139

- Variablentypen wie `int*` sind eigene Variablentypen, und sind nicht äquivalent mit `int`. Der Adressenoperator `&` konvertiert einen Typ wie z.B. `int` in einen Zeigertyp wie z.B. `int*`. Der Operator `*` konvertiert einen Zeigertyp wie z.B. `int*` in den Basistyp, wie z.B. `int`. Ein Zeigertyp kann mit einem gewissen Risiko in einen anderen Zeigertyp konvertiert werden.

Folgende Sitzungen stellen weitere Wege dar, wie Zeiger die Trickkiste von C++ bereichern können.

Selbsttest

1. Wenn eine Variable `x` den Wert 10 enthält, und an der Adresse 0x100 gespeichert ist, was ist der Wert von `x`? Was ist der Wert von `&x`? (Siehe »Einführung in Zeigervariablen«)
2. Wenn `x` ein `int` ist, was ist der Typ von `&x`? (Siehe »Typen von Zeigern«)
3. Warum sollten Sie einen Zeiger an eine Funktion übergeben? (Siehe »Übergabe von Zeigerwerten«)
4. Was ist Heapspeicher und wie erhalten Sie Zugriff darauf? (Siehe »Heapspeicher«)

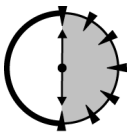


Lektion 14

Mehr über Zeiger

Checkliste

- ☒ Mathematische Operationen auf Zeichenzeigern einführen
- ☒ Die Beziehung von Zeigern und Arrays untersuchen
- ☒ Die Beziehung zur Beschleunigung von Programmen einsetzen
- ☒ Zeigeroperationen auf verschiedene Zeigertypen erweitern
- ☒ Die Argumente von `main( )` im C++- Programmtemplate erklären



30 Min.

Die Zeigertypen, die in Sitzung 13 eingeführt wurden, haben einige interessante Operationen ermöglicht. Die Adresse einer Variablen zu speichern, und dann diese Adresse mehr oder weniger wie die Variable selbst zu benutzen, ist schon ein interessanter Partytrick, aber sein Nutzen ist begrenzt, außer bei der permanenten Modifizierung von Variablen, die an eine Funktion übergeben wurden.

Was Zeiger interessant macht, ist die Fähigkeit, mathematische Operationen ausführen zu können. Sicher, weil die Multiplikation zweier Adressen keinen Sinn macht, ist sie auch nicht erlaubt. Dass zwei Adressen jedoch miteinander verglichen werden können und ein Integeroffset zu einer Adresse addiert werden kann, eröffnet interessante Möglichkeiten, die hier untersucht werden.

14.1 Zeiger und Arrays

Einige der Operatoren, die auf Integerzahlen anwendbar sind, können auch auf Zeigertypen angewendet werden. Dieser Abschnitt untersucht deren Auswirkungen auf Zeiger und die Arraytypen, die wir bisher studiert haben.

14.1.1 Operationen auf Zeigern

Tabelle 14-1: Drei Operationen, die für Zeiger definiert sind

Operation	Ergebnis	Bedeutung
pointer + offset	Zeiger	berechne die Adresse offset viele Einträge von Zeiger pointer entfernt
pointer – offset	Zeiger	das Gegenteil zur Addition
pointer2 – pointer1	Offset	berechne den Abstand zwischen den Zeigern pointer2 und pointer1

(Obwohl nicht in Tabelle 14-1 aufgelistet, sind die abgeleiteten Operatoren, wie z.B. `+=offset` und `pointer++` auch als Variation der Addition definiert.)

Das einfache Speichermodell, das in Sitzung 13 zur Erklärung des Zeigerkonzeptes verwendet wurde, ist auch hier nützlich, um die Wirkungsweise der Operatoren zu erklären. Betrachten Sie ein Array von 32 1-Bit-Zeichen, das wir `cArray` nennen wollen. Wenn das erste Byte des Array an Adresse 0x110 gespeichert wird, dann würde das Array den Speicher von 0x110 bis 0x12f belegen. Das Element `cArray[0]` befindet sich an Adresse 0x110, `cArray[1]` an Adresse 0x111, `cArray[2]` an Adresse 0x112 usw.

Nehmen Sie nun an, dass sich ein Zeiger `ptr` an der Adresse 0x102 befindet. Nachdem die folgende Anweisung

```
ptr = &cArray[0];
```

ausgeführt wurde, enthält `ptr` den Wert 0x110. Dies wird in Abbildung 14.1 gezeigt.

Die Addition einer Integerzahl als Offset zum Zeiger ist so definiert, dass die Beziehungen in Tabelle 14-2 wahr sind. Abbildung 14.2 zeigt außerdem, warum die Addition eines Offset `n` zu `ptr` die Adresse des `n`-ten Elementes von `cArray` berechnet.

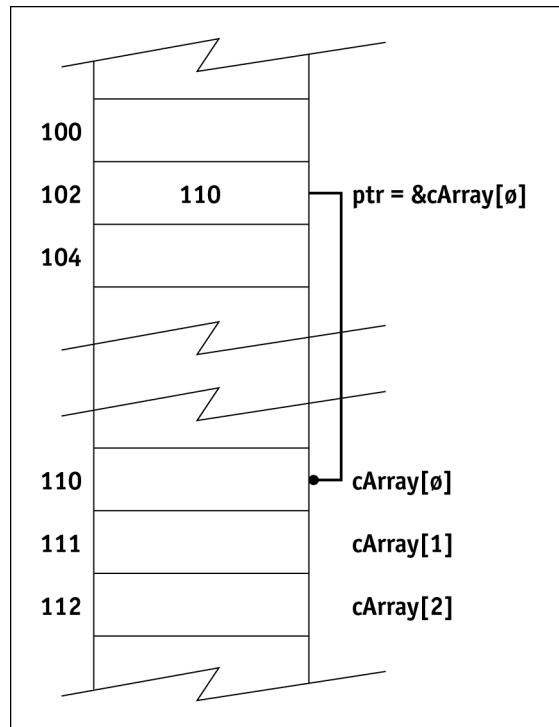
142 **Samstagnachmittag**


Abbildung 14.1: Nach der Zuweisung `ptr = &cArray[0]` zeigt der Zeiger `ptr` auf den Anfang des Arrays `cArray`.

Tabelle 14-2: Zeigeroffsets und Arrays

Offset	Ergebnis	Entspricht ...
+0	0x110	cArray[0]
+1	0x111	cArray[1]
+2	0x112	cArray[2]
...	...	...
+n	0x110+n	cArray[n]

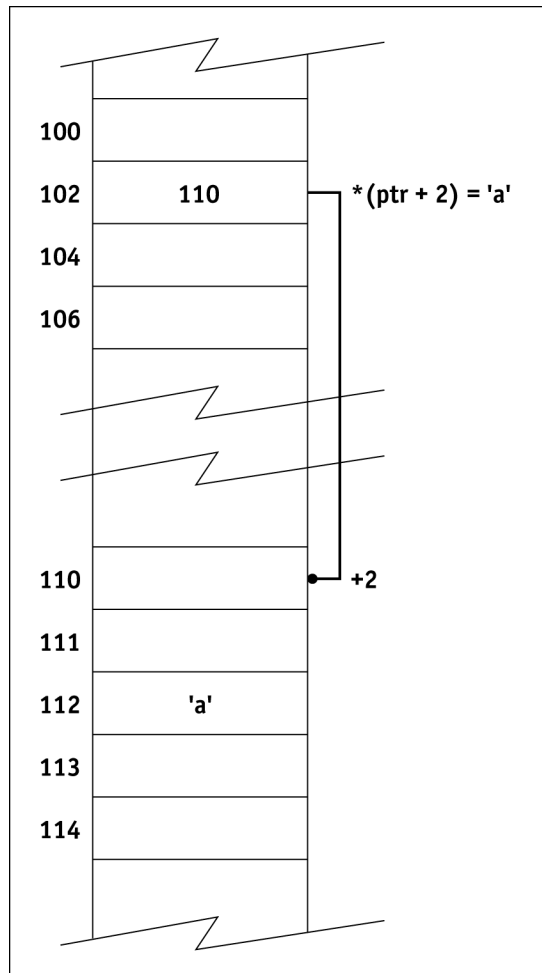


Abbildung 14.2: Der Ausdruck $ptr + i$ hat als Wert die Adresse von $cArray[i]$.

Wenn also

```
char* prt = &cArray[0];
```

gegeben ist, so entspricht

```
*(ptr + n)
```

dem Array-Element

```
cArray[n].
```



Weil $*$ eine höhere Priorität hat als die Addition, addiert $*ptr + n$ zum Zeiger ptr n Zeichen. Die Klammern werden benötigt, um zu erzwingen, dass die Addition vor dem Operator $*$ ausgeführt wird. Der Ausdruck $*(ptr + n)$ greift auf das Zeichen zu, das von ptr aus n Zeichen weiter steht.

144 Samstagnachmittag

In der Tat ist der Zusammenhang der beiden Ausdrücke so stark, dass C++ `array[n]` als nicht mehr, als nur eine vereinfachte Version von `*(ptr + n)` ansieht, wobei `ptr` auf das erste Element von `array` zeigt.

`array[n] -> interpretiert C++ als -> *(&array[0] + n)`

Um die Assoziation zu vervollständigen, verwendet C++ eine weitere Kurzform. Gegeben

```
char cArray[20];
```

dann ist

```
carray == &cArray[0]
```

d.h., der Name des Arrays ohne einen Index repräsentiert die Adresse des Array selber. Wir können also die Assoziation weiter vereinfachen:

`array[n] -> interpretiert C++ als -> *(array + n)`

Das ist eine mächtige Aussage. Z.B. könnte die Funktion `displayArray()` aus Sitzung 11, die den Inhalt eines `int`-Array ausgibt, so geschrieben werden:

```
// displayArray - zeige die Elemente eines Array
//                der Länge nSize
void displayArray(int nArray[], int nSize)
{
    cout << »Der Wert des Array ist:\n«;

    // zeige auf das erste Element von nArray
    int* pArray = nArray;
    while(nSize--)
    {
        cout.width(3);

        // gib Integer aus, worauf pArray zeigt ...
        cout << i << »: » << *pArray << »\n«;

        // ... und bewege Zeiger zum nächsten
        // Element von nArray
        pArray++;
    }
    cout << »\n«;
}
```

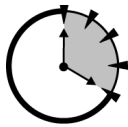
Die neue Funktion `displayArray()` beginnt damit, einen Zeiger `pArray` auf das `int`-Array zu erzeugen, der auf das erste Element von `nArray` zeigt.



Gemäß unserer Konvention weist das `p` auf einen Zeiger hin.

Die Funktion durchläuft dann eine Schleife über jedes Element des Array (wobei `nSize` als die Anzahl der Elemente im Array verwendet wird). In jedem Schleifendurchlauf gibt die Funktion `displayArray()` die entsprechende Integerzahl aus, d.h. das `int`-Element, auf das `pArray` zeigt, bevor der Zeiger zum nächsten Element in `nArray` inkrementiert wird.

Diese Verwendung von Zeigern zum Zugriff auf Arrays wird nirgendwo sonst so häufig verwendet wie bei Zeichenarrays.



20 Min.

14.1.2 Zeichenarrays

Sitzung 11 hat auch erklärt, dass C++ Null-terminierte Zeichenarrays wie einen Datentyp verwendet. C++-Programmierer verwenden oft Zeichenzeiger, um solche Zeichenketten zu manipulieren. Der folgende Beispielcode vergleicht diese Technik mit der früheren Technik der Array-Indizierung.

Zeiger und Array-basierte Manipulation von Zeichenketten

Die Funktion `concatString( )` wurde im Beispiel `Concatenate` in Sitzung 11 deklariert.

```
void concatString(char szTarget[], char szSource[]);
```

Die Prototypdeklaration beschreibt die Typen der Argumente, die die Funktion entgegennimmt, sowie den Rückgabetyt. Diese Deklaration sieht wie die Definition der Funktion aus, nur ohne Body.

Um die Null am Ende des Array `szTarget` zu finden, iteriert die Funktion `concatString( )` durch die Zeichenkette `szTarget` mit der folgenden `while`-Schleife:

```
void concatString(char szTarget[], char szSource[])
{
    // finde das Ende der ersten Zeichenkette
    int nTargetIndex = 0;
    while(szTarget[nTargetIndex])
    {
        nTargetIndex++;
    }

    // ...
}
```

Unter Verwendung der Beziehung zwischen Zeigern und Arrays könnte die Funktion `concatString( )` auch folgenden Prototyp besitzen:

```
void concatString(char* pszTarget, char* pszSource);
```

Das `z` weist auf eine durch 0 (null) abgeschlossene Zeichenkette hin.

Die Zeigerversion von `concatString( )`, die im Programm `ConcatenatePtr` enthalten ist, sieht dann wie folgt aus:

```
void concatString(char* pszTarget, char* pszSource)
{
    // finde das Ende der ersten Zeichenkette
    while(*pszTarget)
    {
        pszTarget++;
    }

    // ...
}
```

Die `while`-Schleife in der Arrayversion von `concatString( )` wurde verlassen, sobald `szTarget[nTargetIndex]` gleich 0 war. Diese Version nun iteriert durch das Array, indem `pszTarget` in jedem Schleifendurchlauf inkrementiert wird, bis das Zeichen, auf das `pszTarget` zeigt, gleich null ist.

146

Samstagnachmittag



Der Ausdruck `ptr++` ist eine Kurzform von `ptr = ptr + 1`.

Wenn die `while`-Schleife verlassen wurde, zeigt `pszTarget` auf das Nullzeichen am Ende der Zeichenkette `szTarget`.



Es ist nicht mehr richtig zu sagen »das Array, auf das `pszTarget` zeigt«, weil `pszTarget` nicht mehr auf den Anfang des Array zeigt.

Das vollständige Beispiel `concatString()`

Und hier ist das vollständige Programm `ConcatenatePtr`:

```

1. // ConcatenatePtr - verbindet zwei Zeichenketten
2. // mit » - « in der Mitte unter
3. // Verwendung von Zeigern statt
4. // Arrayindizes
5. #include <stdio.h>
6. #include <iostream.h>
7.
8. void concatString(char* pszTarget, char* pszSource);
9.
10. int main(int nArg, char* pszArgs[])
11. {
12.     // lies erste Zeichenkette ...
13.     char szString1[256];
14.     cout << »Zeichenkette #1:<<
15.     cin.getline(szString1, 128);
16.
17.     // ... nun die zweite Zeichenkette ...
18.     char szString2[128];
19.     cout << »Zeichenkette #2:<<
20.     cin.getline(szString2, 128);
21.
22.     // ... füge » - « an die erste ...
23.     concatString(szString1, » - «);
24.
25.     // ... füge jetzt die zweite an ...
26.     concatString(szString1, szString2);
27.
28.     // ... und stelle das Ergebnis dar
29.     cout << »\n<< szString1 << »\n<<
30.
31.     return 0;
32. }
33.
34. // concatString - fügt *pszSource an das Ende
35. // von *pszTarget an
36. void concatString(char* pszTarget, char* pszSource)

```

```

37. {
38.     // finde das Ende der ersten Zeichenkette
39.     while(*pszTarget)
40.     {
41.         pszTarget++;
42.     }
43.
44.     // hänge die zweite ans Ende der ersten
45.     // (kopiere auch die Null des Quellarrays -
46.     // dadurch wird das Zielarray mit einem
47.     // Nullzeichen abgeschlossen)
48.     while(*pszTarget++ = *pszSource++)
49.     {
50.     }
51.

```

Die Funktion `main( )` des Programms unterscheidet sich nicht von ihrer Array-basierten Cousine. Die Funktion `concatString( )` ist jedoch signifikant verschieden.

Wie bereits erwähnt, basiert die äquivalente Deklaration von `concatString( )` nun auf Zeigern vom Typ `char*`. Zusätzlich sucht die erste `while`-Schleife innerhalb von `concatString( )` nach dem abschließenden Null-Zeichen am Ende des Arrays `pszTarget`.

Die extrem kompakte Schleife, die dann folgt, kopiert das Array `pszSource` an das Ende des Array `pszTarget`. Die `while`-Klausel macht die ganze Arbeit, indem sie die folgenden Dinge ausführt:

1. Hole das Zeichen, auf das `pszSource` zeigt.
2. Inkrementiere `pszSource` zum nächsten Zeichen.
3. Speichere das Zeichen an der durch `pszTarget` gegebenen Position.
4. Inkrementiere `pszTarget` zum nächsten Zeichen.
5. Führe den Body der Schleife so lange aus, bis das Zeichen 0 (null) ist.

Nachdem der leere Rumpf der `while`-Schleife verlassen wurde, wird die Kontrolle an die `while( )`-Klausel selber zurückgegeben. Die Schleife wird so lange wiederholt, bis das Zeichen, das nach `*pszTarget` kopiert wurde, gleich dem Nullzeichen ist.

Warum Arrayzeiger?

Die manchmal kryptische Natur von Zeiger-basierter Manipulation von Zeichenketten kann den Leser schnell zur Frage »warum?« führen. D.h. welchen Vorteil bietet die `char*`-basierte Zeigerversion von `concatString( )` gegenüber der doch leichter lesbaren Indexversion?



Die Zeigerversion von `concatString( )` kommt in C++-Programmen häufiger vor als die Array-Version aus Sitzung 11.

Die Antwort ist teilweise historisch und teilweise menschlicher Natur. So kompliziert sie auch für den menschlichen Leser aussehen mag, kann eine Anweisung wie in Zeile 48 in eine unglaublich kleine Anzahl von Maschineninstruktionen überführt werden. Ältere Computerprozessoren waren nicht so schnell wie die heutigen. Als C, der Vorgänger von C++, vor etwa 30 Jahren entwickelt wurde, war es toll, einige Maschineninstruktionen einsparen zu können. Das gab C einen großen Vorteil

148 Samstagnachmittag

gegenüber anderen Sprachen aus dieser Zeit, insbesondere gegenüber FORTRAN, die keine Zeigerarithmetik enthielt.

Außerdem mögen es die Programmierer, clevere Programme zu schreiben, damit die Sache nicht langweilig wird. Wenn C++-Programmierer erst einmal gelernt haben, wie man kompakte und kryptische, aber effiziente Anweisungen schreibt, gibt es kein Mittel, sie wieder zur Suche in Arrays mittels Index zurückzubringen.

**Tipp**

Erzeugen Sie keine komplexen C++-Ausdrücke mit der Absicht, effizienteren Code zu erzeugen. Es gibt keinen offensichtlichen Zusammenhang zwischen der Anzahl der C++-Anweisungen und der Anzahl der Maschineninstruktionen. Z.B. könnten die beiden folgenden Ausdrücke die gleiche Anzahl von Maschineninstruktionen erzeugen:

```
*pszArray1++ = '\0';
```

```
*pszArray2 = '0';
```

```
pszArray2 = pszArray2 + 1;
```

Früher, als die Compiler noch einfacher gebaut waren, hätte die erste Version sicherlich weniger Instruktionen erzeugt.

14.1.3 Operationen auf unterschiedlichen Zeigertypen

Die beiden Beispiele von Zeigermanipulationen, die bisher gezeigt wurden, `concatString(char*, char*)` und `displayArray(int*)`, unterscheiden sich grundlegend in zwei Punkten:

Es ist für Sie nicht sehr schwer, sich selbst davon zu überzeugen, dass `szTarget + n` auf `szTarget[n]` zeigt, wenn Sie berücksichtigen, dass jedes Zeichen in `szTarget` ein einziges Byte belegt. Wenn also `szTarget` an Adresse `0x100` gespeichert ist, dann steht das sechste Element an Adresse `0x105` ($0x100 + 5$ ist gleich $0x105$).

**Hinweis**

Weil C++-Arrays bei 0 zu zählen beginnen, ist `szTarget[5]` das sechste Element des Array.

Es ist nicht offensichtlich, dass Zeigeraddition auch für `nArray` funktioniert, da jedes Element von `nArray` ein `int` ist und damit 4 Bytes belegt. Wenn das erste Element von `nArray` an Adresse `0x100` steht, dann steht das sechste Element an Adresse `0x114` ($0x100 + (5 * 4) = 0x114$).

Glücklicherweise zeigt in C++ `array + n` auf das Element `array[n]`, unabhängig davon, wie groß ein einzelnes Element von `array` ist.

**Hinweis**

Eine gute Parallele bieten Häuserblocks in einer Stadt. Wenn alle Adressen in jeder Straße fortlaufend ohne Lücken, nummeriert wären, dann wäre die Hausnummer 1605 das sechste Haus in Block 1600. Um den Postboten nicht zu sehr zu verwirren, wird diese Beziehung eingehalten, unabhängig von der Größe der Häuser.

14.1.4 Unterschiede zwischen Zeigern und Arrays

Außer den äquivalenten Typen, gibt es einige Unterschiede zwischen einem Array und einem Zeiger. Zum einen alloziert ein Array Speicher für die Daten, ein Zeiger tut das nicht:

```
void arrayVsPointer()
{
    // alloziere Speicher für 128 Zeichen
    char cArray[128];

    // alloziere Speicher für einen Zeiger
    char* pArray;
}
```

Hier belegt `cArray` 128 Bytes, das ist der Speicher, der für 128 Zeichen benötigt wird. `pArray` belegt nur 4 Bytes, das ist der Speicher, der für einen Zeiger benötigt wird.

Die folgende Funktion funktioniert nicht:

```
void arrayVsPointer()
{
    // greife auf Elemente mit Array zu
    char cArray[128];
    cArray[10] = '0';
    *(cArray + 10) = '0';

    // greife auf ein 'Element' des Arrays
    // zu, das nicht existiert
    char* pArray;
    pArray[10] = '0';
    *(pArray + 10) = '0';
}
```

Der Ausdruck `cArray[10]` und `*(cArray + 10)` sind äquivalent und zulässig. Die beiden Ausdrücke, die `pArray` enthalten, machen keinen Sinn. Während sie in C++ zulässig sind, enthält das nicht initialisierte `pArray` einen zufälligen Wert. Somit versucht das zweite Anweisungspaar ein Nullzeichen irgendwo in den Speicher zu schreiben.



Diese Art Fehler wird im Allgemeinen von der CPU abgefangen und resultiert dann im gefürchteten Fehler einer Segmentverletzung, den Sie hin und wieder bei Ihren Lieblingsprogrammen antreffen.

Ein zweiter Unterschied ist, dass `cArray` eine Konstante ist, was auf `pArray` nicht zutrifft. Somit arbeitet die folgende `for`-Schleife, die das Array `cArray` initialisieren soll, nicht korrekt:

```
void arrayVsPointer()
{
    char cArray[10];
    for (int i = 0; i < 10; i++)
    {
        *cArray = '\0';    // das macht Sinn ...
        cArray++;          // ... das nicht
    }
}
```

150 Samstagnachmittag

Der Ausdruck `cArray++` macht nicht mehr Sinn als `10++`. Die korrekte Version sieht so aus:

```
void arrayVsPointer()
{
    char cArray[10];
    char* pArray = cArray;
    for (int i = 0; i < 10; i++)
    {
        *pArray = '\0';    // das funktioniert
        pArray++;
    }
}
```

14.2 Argumente eines Programms

Arrays von Zeigern sind ein anderer Typ Array, der von besonderem Interesse ist. Dieser Abschnitt untersucht, wie Sie diese Arrays einsetzen können, um Ihre Programme einfacher zu machen.

14.2.1 Arrays von Zeigern

Weil Arrays Daten beliebigen Typs enthalten können, ist es möglich, Zeiger in Arrays zu speichern. Das Folgende deklariert ein Array von Zeigern auf `int`:

```
int* pnInts[10];
```

Gegeben die obige Deklaration, ist `pnInts[0]` ein Zeiger auf einen `int`-Wert. Somit ist das Folgende wahr:

```
void fn()
{
    int n1;
    int* pnInts[3];
    pnInts[0] = &n1;
    *pnInts[0] = 1;
}
```

oder

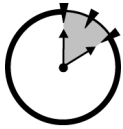
```
void fn()
{
    int n1, n2, n3;
    int* pnInts[3] = {&n1, &n2, &n3};
    for (int i = 0; i < 3; i++)
    {
        *pnInts[i] = 0;
    }
}
```

oder sogar

```
void fn()
{
    int* pnInts[3] = {(new int),
                     (new int),
                     (new int)};
    for (int i = 0; i < 3; i++)
    {
        *pnInts[i] = 0;
    }
}
```

Das Letztere deklariert drei `int`-Objekte vom Heap.

Der häufigste Einsatz von Zeigerarrays ist das Anlegen eines Array von Zeichenketten. Die folgenden zwei Beispiele zeigen, warum Arrays von Zeichenketten nützlich sind.



10 Min.

14.2.2 Arrays von Zeichenketten

Betrachten Sie eine Funktion, die den Namen eines Monats zurückgibt, der zu einer Integerzahl gehört. Wenn das Programm z.B. den Wert 1 erhält, gibt es die Zeichenkette »Januar« zurück.

Die Funktion könnte wie folgt geschrieben werden:

```
// int2month() - gib den Namen des Monats zurück
char* int2month(int nMonth)
{
    char* pszReturnValue;

    switch(nMonth)
    {
        case 1: pszReturnValue = »Januar«;
                break;
        case 2: pszReturnValue = »Februar«;
                break;
        case 3: pszReturnValue = »März«;
                break;
        // ... usw ...
        default: pszReturnValue = »invalid«;
    }
    return pszReturnValue;
}
```

Wenn 1 übergeben wird, geht die Kontrolle an die erste `case`-Anweisung über und die Funktion würde einen Zeiger auf die Zeichenkette »Januar« zurückgeben; wenn eine 2 übergeben wird, kommt »Februar« zurück usw.



Hinweis

Das `switch()`-Kontrollkommando ist vergleichbar mit einer Folge von `if`-Anweisungen.

Eine elegantere Lösung nutzt den Integerwert als Index für ein Array von Zeigern auf die Monatsnamen. Praktisch sieht das so aus:

```
// int2month() - gib den Namen des Monats zurück
char* int2month(int nMonth)
{
    // überprüfe den Wert auf Gültigkeit
    if (nMonth < 1 || nMonth > 12)
    {
        return »ungültig«;
    }

    // nMonth ist gültig - gib Monatsnamen zurück
```

152 Samstagnachmittag

```

char* pszMonths[] = {»invalid«,
                    »Januar«,
                    »Februar«,
                    »März«,
                    »April«,
                    »Mai«,
                    »Juni«,
                    »Juli«,
                    »August«,
                    »September«,
                    »Oktober«,
                    »November«,
                    »Dezember«};

return pszMonths[nMonth];
}

```

Hierbei überprüft die Funktion `int2month( )` zuerst, ob der Wert von `nMonth` zwischen 1 und 12 ist (die `default`-Klausel der `switch`-Anweisung hat das für uns im vorhergehenden Beispiel erledigt). Wenn `nMonth` zulässig ist, benutzt die Funktion diesen Wert als Offset für das Array, das die Monatsnamen enthält.

14.2.3 Die Argumente von `main( )`

Sie haben bereits einen anderen Einsatz eines Zeigerarray auf Zeichenketten gesehen: Die Argumente der Funktion `main( )`.

Die Argumente eines Programms sind die Zeichenketten, die beim Aufruf hinter dem Programmnamen angegeben werden. Nehmen Sie z.B. an, dass ich das folgende Kommando hinter dem MS-DOS-Prompt eingegeben habe:

```
MyProgram file.txt /w
```

MS-DOS führt das Programm, das in der Datei `MyProgram.exe` enthalten ist, aus und übergibt ihm die Argumente `file.txt` und `/w`.



*Der Nutzen des Begriffs **Argument** ist ein wenig verwirrend. Die Argumente eines Programms und die Argumente einer C++-Funktion folgen einer unterschiedlichen Syntax, aber die Bedeutung ist die gleiche.*

Betrachten Sie das folgende Beispielprogramm:

```

// PrintArgs - schreibt die Argumente des Programms
//              in die Standardausgabe
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{
    // gib Kopfzeile aus
    cout << »Argument von « << pszArgs[0] << »\n«;
}

```

```

// nun schreibe die Argumente raus
for (int i = 1; i < nArg; i++)
{
    cout << i << »:« << pszArgs[i] << »\n«;
}

// das war es
cout << »Das war es\n«;
return 0;
}

```

Wie immer akzeptiert die Funktion `main( )` zwei Argumente. Das erste ist ein `int` und trägt den Namen `nArgs`. Diese Variable enthält die Anzahl der Argumente, die an das Programm übergeben wurden. Das zweite Argument ist ein Array von Zeigern vom Typ `char*`, das ich `pszArgs` genannt habe. Jedes dieser `char*`-Elemente zeigt auf ein Argument, das dem Programm übergeben wurde.

Betrachten Sie das Programm `PrintArgs`. Wenn ich das Programm aufrufe mit

```
PrintArgs arg1 arg2 arg3 /w
```

von der Kommandozeile eines MS-DOS-Fensters aus, wäre `nArgs` gleich 5 (eins für jedes Argument). Das erste Argument ist der Name des Programms selber. Somit zeigt `pszArgs[0]` auf `PrintArgs`. Die restlichen Elemente in `pszArgs` zeigen auf die Programmargumente. Das Element `pszArgs[1]` zeigt auf `arg1`, `pszArgs[2]` zeigt auf `arg2` usw. Weil MS-DOS `/w` keine besondere Bedeutung beimisst, wird diese Zeichenkette in gleicher Weise als ein Argument an das Programm übergeben.



Das Gleiche gilt nicht für die Richtungszeichen »<«, »>« und »|«. Diese haben unter MS-DOS eine besondere Bedeutung und werden nicht als Argument an das Programm übergeben.

Es gibt verschiedene Wege, Argumente an eine Funktion zu übergeben. Der einfachste Weg ist, das Programm vom MS-DOS-Prompt aus aufzurufen. Beide Debugger, von Visual C++ und von GNU C++, stellen einen Mechanismus bereit, um Argumente während des Debuggens zu übergeben.

In Visual C++, wählen Sie das Debug-Feld in der Dialog-Box »Project Settings« aus. Geben Sie Ihre Argumente in das Eingabefenster »Program Arguments« ein wie in Abbildung 14.3 zu sehen ist. Das nächste Mal, wenn Sie Ihr Programm starten, übergibt Visual C++ diese Argumente.

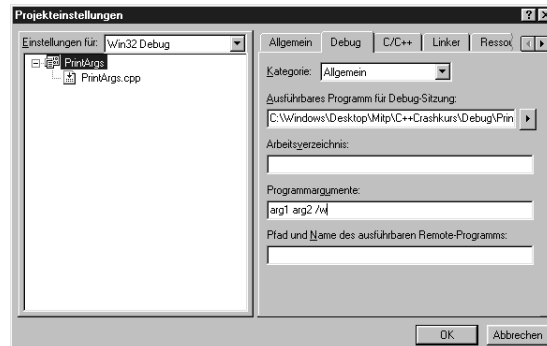
154 Samstagnachmittag

Abbildung 14.3: Visual C++ verwendet Project Settings zur Übergabe von Argumenten an das Programm während des Debuggens.

In `rhide` wählen Sie »Argumente...« im Menü »Start«. Geben sie die Argumente im Fenster ein. Dies ist in Abbildung 14.4 zu sehen.

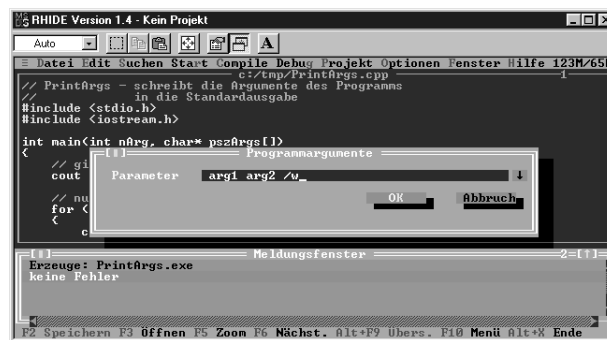


Abbildung 14.4: In `rhide` stehen die Programmargumente im Start-Menü.



0 Min.

Zusammenfassung

Alle Programmiersprachen basieren die Indizierung von Arrays auf einfachen mathematischen Operationen auf Zeigern. Durch die Möglichkeit für den Programmierer, auf diese Art Operation direkt zuzugreifen, gibt C++ dem Programmierer eine große semantische Freiheit. Der C++-Programmierer kann die Beziehung zwischen der Manipulation von Arrays und Zeigern untersuchen und zu seinem Vorteil nutzen.

In dieser Sitzung haben wir gesehen, dass

- die Indizierung auf Arrays einfache mathematische Operationen auf Zeigern beinhaltet. C++ ist praktisch einzigartig darin, dass der Programmierer diese Operationen selber ausführen kann.
- Zeigeroperationen auf Zeichenarrays boten das größte Potenzial zur Leistungssteigerung bei den frühen C- und C++-Compilern. Ob das immer noch der Fall ist, darüber lässt sich streiten. Jedenfalls sind Zeichenzeiger Teil des täglichen Lebens geworden.

Lektion 14 – Mehr über Zeiger**155**

- C++ passt die Zeigerarithmetik an die unterschiedliche Größe der Objekte an, auf die Zeiger verweisen. Somit vergrößert die Inkrementierung eines `char`-Zeigers dessen Wert um 1, während die Inkrementierung eines `double`-Zeigers zu einer Vergrößerung seines Wertes um 8 führt. Die Inkrementierung eines Zeigers auf Klassenobjekte kann zu einer Vergrößerung von Hunderten von Bytes führen.
- Arrays von Zeigern können signifikant die Effizienz eines Programms erhöhen, das einen `int`-Wert in eine Konstante eines anderen Typs verwandelt, wie z.B. eine Zeichenkette oder ein Bitfeld.
- Argumente eines Programms werden an die Funktion `main()` als Array von Zeigern auf Zeichenketten übergeben.

Selbsttest

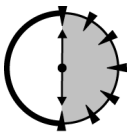
1. Wenn das erste Element eines Array von Zeichen `c[]` an Adresse `0x100` steht, was ist die Adresse von `c[2]`? (Siehe »Operationen auf Zeigern«)
2. Was ist das Indexäquivalent zum Zeigerausdruck `*(c + 2)`? (Siehe »Zeiger und Array-basierte Manipulation von Zeichenketten«)
3. Was ist der Sinn der beiden Argumente von `main()`? (Siehe »Die Argumente von `main()`«)



Zeiger auf Objekte

Checkliste

- ☒ Zeiger auf Objekte deklarieren und verwenden
- ☒ Objekte durch Zeiger übergeben
- ☒ Objekte vom Heap allozieren
- ☒ Verkettete Listen erzeugen und manipulieren
- ☒ Verkettete Listen von Objekten und Objektarrays vergleichen



30 Min.

Sitzung 12 demonstrierte, wie Arrays und Klassenstrukturen in Arrays von Objekten kombiniert werden können, um eine Reihe von Problemen zu lösen. In gleicher Weise löst die Einführung von Zeigern auf Objekte einige Probleme, die von Objektarrays nicht so leicht gelöst werden können.

15.1 Zeiger auf Objekte

Ein Zeiger auf eine vom Programmierer definierte Struktur arbeitet im Wesentlichen so, wie Zeiger auf die elementaren Typen:

```
int* pInt;
class MyClass
{
public:
    int n1;
    char c2;
};
MyClass mc;
MyClass* pMS = &mc;
```


Lektion 15 – Zeiger auf Objekte 157



Der Typ von `pMS` ist »Zeiger auf `MyClass`«, was auch als `MyClass*` ausgedrückt werden kann.

Auf Elemente eines solchen Objektes kann wie folgt zugegriffen werden:

```
(*pMS).n1 = 1;
(*pMS).c2 = '\0';
```

In Wörtern besagt der erste Ausdruck »weise 1 dem Element `n1` des `MS`-Objektes zu, auf das `pMS` verweist.«



Die Klammern sind notwendig, weil ».« eine höhere Priorität hat als »*«. Der Ausdruck `*mc.pN1` bedeutet »die Integerzahl, auf die das `pN1`-Element des Objektes `mc` verweist«.

Genauso wie C++ eine Kurzform für den Gebrauch von Arrays bereitstellt, definiert C++ bequemere Operatoren, um auf die Elemente eines Objektes zuzugreifen. Der Operator `->` ist wie folgt definiert:

```
(*pMS).n1 ist äquivalent zu pMS->n1
```

Der Pfeiloperator wird fast ausschließlich verwendet, weil das so leichter zu lesen ist. Die beiden Formen sind jedoch völlig äquivalent.

Teil 3 – Samstagnachmittag
Lektion 15

15.1.1 Übergabe von Objekten

Ein Zeiger auf ein Klassenobjekt kann an eine Funktion in der gleichen Weise übergeben werden wie einfache Zeigertypen.

```
// PassObjectPtr - demonstriert Funktionen, die einen
//                Zeiger auf ein Objekt erwarten
#include <stdio.h>
#include <iostream.h>

// MyClass - eine Testklasse ohne Bedeutung
class MyClass
{
public:
    int n1;
    int n2;
};

// myFunc - Version mit Wertübergabe
void myFunc(MyClass mc)
{
    cout << »In myFunc(MyClass)\n«;
    mc.n1 = 1;
    mc.n2 = 2;
}
```

158 Samstagnachmittag

```
// myFunc - Version mit Zeigerübergabe
void myFunc(MyClass* pMS)
{
    cout << »In myFunc(MyClass*)\n«;
    pMS->n1 = 1;
    pMS->n2 = 2;
}

int main(int nArg, char* pszArgs[])
{
    // Definiere ein Dummy-Object
    MyClass mc = {0, 0};
    cout << »Anfangswert = \n«;
    cout << »n1 = » << mc.n1 << »\n«;

    // übergib als Wert
    myFunc(mc);
    cout << »Ergebnis = \n«;
    cout << »n1 = » << mc.n1 << »\n«;

    // übergib als Zeiger
    myFunc(&mc);
    cout << »Ergebnis = \n«;
    cout << »n1 = » << mc.n1 << »\n«;
    return 0;
}
```

Das Hauptprogramm erzeugt ein Objekt aus der Klasse `MyClass`. Das Objekt wird zuerst an die Funktion `myFunc(MyClass)` übergeben, und dann wird die Adresse des Objektes an die Funktion `myFunc(MyClass*)` übergeben. Beide Funktionen ändern den Wert des Objektes – nur die Änderungen, die innerhalb von `myFunc(MyClass*)` durchgeführt wurden, bleiben erhalten.

Im Aufruf von `myFunc(MyClass)` macht C++ eine Kopie des Objektes. Änderungen am Objekt `mc` in dieser Funktion bleiben in `main( )` nicht erhalten. Der Aufruf von `myFunc(MyClass*)` übergibt die Adresse auf das ursprüngliche Objekt in `main( )`. Das Objekt enthält alle gemachten Änderungen, wenn die Kontrolle an `main( )` zurückgegeben wird.

Dieser Vergleich von Kopie und Original ist das Gleiche, wie der Vergleich der beiden Funktionen `fn(int)` und `fn(int*)`.



Abgesehen vom Erhalt von Änderungen kann die Übergabe eines 4-Byte-Zeigers wesentlich effizienter sein, als die Kopie eines Objektes.

Lektion 15 – Zeiger auf Objekte 159**15.1.2 Referenzen**

Sie können Referenzen verwenden, um C++ einige Zeigermanipulationen durchführen zu lassen.

```
// myFunc - mc behält Änderungen in
//          aufrufender Funktion
void myFunc(MyClass& mc)
{
    mc.n1 = 1;
    mc.n2 = 2;
}

int main(int nArgs, char* pszArgs[])
{
    MyClass mc;
    myFunc(mc);
    // ...
}
```



Sie haben dieses Feature bereits gesehen. Das Beispiel `ClassData` in Sitzung 12 verwendete eine Referenz auf ein Klassenobjekt im Aufruf `getData(NameData-Set&)`, um die gelesenen Daten an den Aufrufenden zurückzugeben.

15.1.3 Rückgabe an den Heap

Man muss sehr vorsichtig sein, keine Referenz auf ein Objekt, das lokal definiert wurde, zurückzugeben:

```
MyClass* myFunc()
{
    MyClass mc;
    MyClass* pMC = &mc;
    return pMC;
}
```

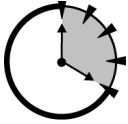
Wenn `myFunc()` zurückkehrt, verlässt das Objekt `mc` seinen Gültigkeitsbereich. Der Zeiger, der von `myFunc()` zurückgegeben wird, ist nicht gültig in der aufrufenden Funktion. (Siehe Sitzung 13 zu Details.)

Das Objekt vom Heap zu allozieren, löst das Problem:

```
MyClass* myFunc()
{
    MyClass* pMC = new MyClass;
    return pMC;
}
```



Der Heap wird verwendet, um Objekte in verschiedenen Situationen zu allozieren.



15.2 Die Datenstruktur Array

Als ein Container von Objekten hat das Array eine Reihe von Vorteilen, insbesondere die Fähigkeit, auf ein Element schnell und effizient zugreifen zu können:

20 Min.

```
MyClass mc[100]; // alloziere Platz für 100 Einträge
mc[n];           // Zugriff auf den n-ten Eintrag
```

Doch das Array hat auch Nachteile:

Arrays haben eine fest Länge. Sie können die Anzahl der Arrayelemente zwar zur Laufzeit bestimmen, aber wenn Sie das Array erst einmal angelegt haben, können Sie seine Größe nicht mehr verändern.

```
void fn(int nSize)
{
    // alloziere ein Objekt, um n Objekte aus
    // der Klasse MyClass zu speichern
    MyClass* pMC = new MyClass[n];

    // die Größe des Arrays ist jetzt fest und
    // kann nicht mehr geändert werden

    // ...
}
```

Zusätzlich muss jeder Eintrag im Array vom gleichen Typ sein. Es ist nicht möglich, Objekte der Klassen `MyClass` und `YourClass` im gleichen Array zu speichern.

Schließlich ist es schwierig, ein Objekt mitten in das Array einzufügen. Um ein Objekt hinzuzufügen oder zu löschen, muss das Programm angrenzende Objekte nach oben oder nach unten kopieren, um eine Lücke zu schaffen oder zu schließen.

Es gibt Alternativen zu Arrays, die diese Einschränkungen nicht besitzen. Die bekannteste ist die verkettete Liste.

15.3 Verkettete Listen

Die verkettete Liste verwendet das gleiche Prinzip wie bei der Übung »wir geben uns die Hände, um die Straße zu überqueren«, als Sie noch ein Kind waren. Jedes Objekt enthält einen Verweis auf das nächste Objekt in der Kette. Der »Lehrer«, auch als Kopfzeiger bekannt, zeigt auf das erste Element der Liste.

Eine verkettete Liste ist wie folgt deklariert:

```
class LinkableClass
{
public:
    LinkableClass* pNext;

    // andere Elemente der Klasse
};
```

Lektion 15 – Zeiger auf Objekte 161

Hierbei zeigt pNext auf den nächsten Eintrag in der Liste. Das sehen Sie in Abbildung 15.1.

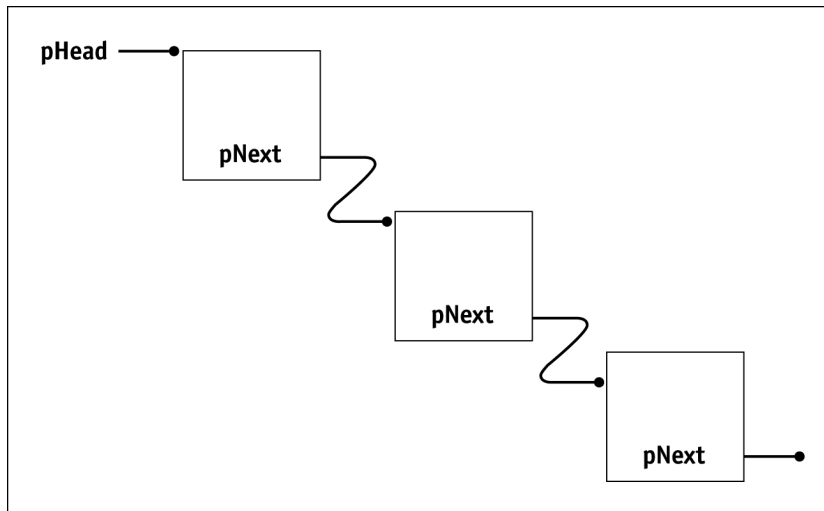


Abbildung 15.1: Eine verkettete Liste besteht aus einer Anzahl von Objekten; jedes Objekt verweist auf das nächste Objekt in der Liste.

Der Kopfzeiger ist einfach ein Zeiger von Typ `LinkableClass*`:

```
LinkableClass* pHead = (LinkableClass*)0;
```



Initialisieren Sie jeden Zeiger mit 0, der im Kontext von Zeigern auch als null bezeichnet wird. Dieser Wert wird als Nullzeiger bezeichnet. In jedem Fall wird der Zugriff auf die Adresse 0 immer zum Anhalten des Programms führen.



Der Cast von 0 als Typ `int` nach `LinkableClass*` ist nicht nötig. C++ interpretiert 0 als beliebigen Typ, als eine Art »universeller Zeiger«. Ich finde jedoch, dass es ein guter Stil ist.

15.3.1 Anfügen am Kopf der verketteten Liste

Um zu sehen, wie verkettete Listen in der Praxis arbeiten, betrachten Sie die folgende einfache Beispielfunktion, die das ihr übergebene Argument an den Anfang der Liste anfügt:

```
void addHead(LinkableClass* pLC)
{
    pLC->pNext = pHead;
    pHead = pLC;
}
```

Dieser Prozess wird in Abbildung 15.2 grafisch dargestellt. Nach der ersten Zeile zeigt das Objekt *pLC auf das erste Objekt der Liste (das gleiche, auf das pHead zeigt), dargestellt als Schritt A. Nach der zweiten Anweisung zeigt der Kopfzeiger auf das übergebene Objekt *pLC, dargestellt als Schritt B.

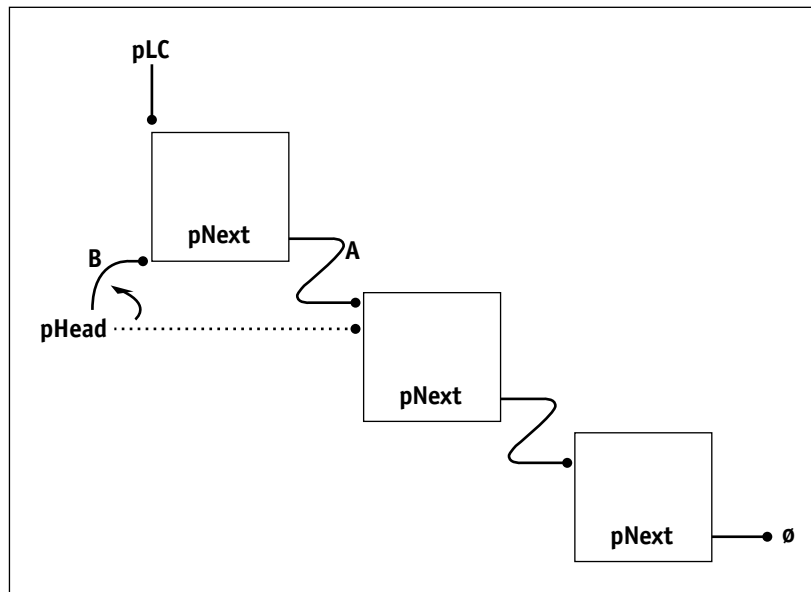


Abbildung 15.2: Ein Element wird in zwei Schritten am Kopf der Liste eingefügt.

15.3.2 Andere Operationen auf verketteten Listen

Ein Objekt am Kopf einer Liste einzufügen ist die einfachste der Operationen auf verketteten Listen. Ein Element am Ende der Liste einzufügen ist viel trickreicher.

```
void addTail(LinkableClass* pLC)
{
    // beginne mit einem Zeiger auf den Anfang
    // der verketteten Liste
    LinkableClass* pCurrent = pHead;

    // iteriere durch die Liste, bis wir das
    // letzte Element der Liste finden - das ist das
    // Element, dessen Zeiger pNext gleich null ist
    while(pCurrent->pNext != (LinkableClass*)0)
    {
        // bewege pCurrent zum nächsten Eintrag
        pCurrent = pCurrent->pNext;
    }

    // lasse das Objekt auf LC verweisen
    pCurrent->pNext = pLC;

    // stelle sicher, dass LC's pNext-Zeiger null
}
```

Lektion 15 – Zeiger auf Objekte 163

```
// ist, wodurch LC als letztes Element in der
// Liste markiert wird
pLC->pNext = (LinkableClass*)0;
}
```

Die Funktion `addTail( )` beginnt mit einer Iteration, um das Element zu finden, dessen `pNext`-Zeiger gleich null ist – das ist das letzte Element in der Liste. Wenn dieses Element gefunden ist, verkettet `addTail( )` das Objekt `*pLC` mit dem Ende der Liste.

(Tatsächlich enthält die Funktion `addTail( )` so, wie wir sie geschrieben haben, einen Bug. Ein spezieller Test muss hinzugefügt werden, um festzustellen, ob `pHead` selbst null ist, was anzeigen würde, dass die Liste leer war).

Die Funktion `remove( )` ist ähnlich. Die Funktion entfernt das spezifizierte Objekt aus der Liste und gibt 1 zurück, wenn dies erfolgreich war, sonst 0.

```
int remove(LinkableClass* pLC)
{
    LinkableClass* pCurrent = pHead;

    // wenn die Liste leer ist, ist *pLC
    // offensichtlich nicht in der Liste
    if (pCurrent == (LinkableClass*)0)
    {
        return 0;
    }

    // iteriere durch die Schleife und suche
    // nach dem Element bis zum Ende der Liste
    while(pCurrent->pNext)
    {
        // wenn der nächste Eintrag *pLC ist ...
        if (pLC == pCurrent->pNext)
        {
            // ...dann soll der aktuelle Eintrag
            // auf dessen nächsten Eintrag verweisen
            pCurrent->pNext = pLC->pNext;

            // nicht absolut notwendig, aber entferne
            // das nächste Objekt aus *pLC, um nicht
            // verwirrt zu werden
            pLC->pNext = (LinkableClass*)0;
            return 1;
        }
    }
    return 0;
}
```

Die Funktion `remove( )` überprüft zuerst, ob die Liste auch nicht leer ist – wenn sie leer ist, gibt die Funktion den Fehlercode zurück, da offensichtlich das Objekt `*pLC` nicht in der Liste enthalten ist. Wenn die Liste nicht leer ist, iteriert `remove( )` durch alle Elemente, bis das Objekt gefunden wird, das auf `*pLC` verweist. Wenn dieses Objekt gefunden wird, setzt `remove( )` den Zeiger `pCurrent->pNext` an `*pLC` vorbei. Dieser Prozess wird in Abbildung 15.3 grafisch veranschaulicht.

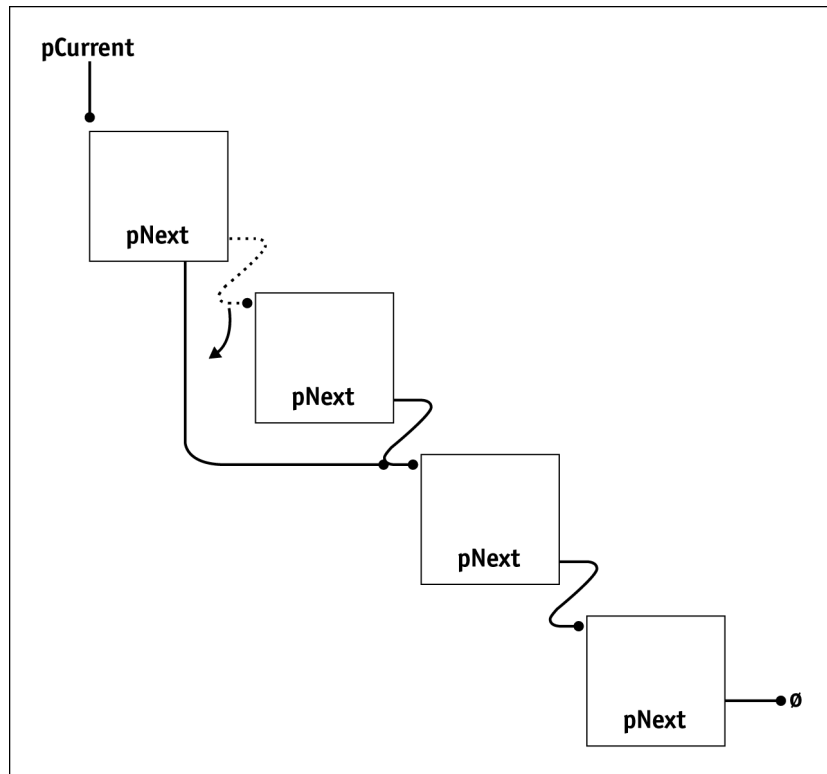
164 Samstagnachmittag

Abbildung 15.3: »Umgehe« einen Eintrag, um ihn aus der Liste zu entfernen

15.3.3 Eigenschaften verketteter Listen

Verkettete Listen sind all das, was Arrays nicht sind. Verkettete Listen können nach Belieben erweitert und verkleinert werden, indem Objekte eingefügt oder entfernt werden. Das Einfügen eines Objektes in die Mitte der verketteten Liste ist schnell und einfach – bereits eingefügte Elemente müssen nicht an eine andere Stelle kopiert werden. In gleicher Weise ist das Sortieren von Elementen in einer verketteten Liste schneller durchzuführen als in einem Array. Array-Elemente sind direkt über einen Index ansprechbar – eine damit vergleichbare Eigenschaft besitzen die verketteten Listen nicht. Programme müssen manchmal die gesamte Liste durchsuchen, um einen bestimmten Eintrag zu finden.

**10 Min.**

15.4 Ein Programm mit verkettetem NameData

Das Programm `LinkedListData`, das Sie im Folgenden finden, ist eine Version des Array-basierten Programms `ClassData` aus Sitzung 12, die eine verkettete Liste verwendet.

```
// LinkedListData - speichere Namensdaten in einer
//                      verketteten Liste von Objekten
#include <stdio.h>
#include <iostream.h>
#include <string.h>

// NameDataSet - speichere Vorname, Name und
//                      Sozialversicherungsnummer
class NameDataSet
{
public:
    char szFirstName[128];
    char szLastName [128];
    int  nSocialSecurity;

    // der Verweis auf das nächste Listenelement
    NameDataSet* pNext;
};

// der Zeiger auf den ersten Eintrag der Liste
NameDataSet* pHead = 0;

// addTail - füge ein neues Element der Liste an
void addTail(NameDataSet* pNDS)
{
    // stelle sicher, dass der Zeigern von *pDNS null
    // ist, weil das zum letzten Listenelement wird
    pNDS->pNext = 0;

    // wenn die Liste leer ist, dann zeige einfach
    // mit dem Kopfzeiger auf den aktuellen Eintrag
    // und fertig
    if (pHead == 0)
    {
        pHead = pNDS;
        return;
    }

    // ansonsten finde das letzte Element der Liste
    NameDataSet* pCurrent = pHead;
    while(pCurrent->pNext)
    {
        pCurrent = pCurrent->pNext;
    }

    // jetzt füge den aktuellen Eintrag diesem an
    pCurrent->pNext = pNDS;
}

// getData - lies einen neuen Namen und eine
```

166 Samstagnachmittag

```
// Sozialversicherungsnummer; gib null
// zurück, wenn nichts mehr zu lesen ist
NameDataSet* getData()
{
    // neuer Eintrag, der zu füllen ist
    NameDataSet* pNDS = new NameDataSet;

    // lies den Vornamen
    cout << »\nVorname:<<
    cin >> pNDS->szFirstName;

    // wenn der Vorname 'ende' oder 'ENDE' ist...
    if ((strcmp(pNDS->szFirstName, »ende<<) == 0)
        ||
        (strcmp(pNDS->szFirstName, »ENDE<<) == 0))
    {
        // ... lösche das immer noch leere Objekt ...
        delete pNDS;

        // ... gib eine null zurück für Eingabeende
        return 0;
    }

    // lies die übrigen Elemente
    cout << »Nachname:<<
    cin >> pNDS->szLastName;

    cout << »Sozialversicherungsnummer:<<
    cin >> pNDS->nSocialSecurity;

    // Zeiger auf nächstes Element auf null setzen
    pNDS->pNext = 0;

    // gib die Adresse auf das neue Element zurück
    return pNDS;
}

// displayData - Ausgabe des Datensatzes
// auf den pDNS zeigt
void displayData(NameDataSet* pNDS)
{
    cout << pNDS->szFirstName
        << » »
        << pNDS->szLastName
        << »/<<
        << pNDS->nSocialSecurity
        << »\n<<
}

int main(int nArg, char* pszArgs[])
{
    cout << »Lies Vornamen, Nachnamen und\n<<
        << »Sozialversicherungsnummer\n<<
        << »Geben Sie 'ende' als Vorname ein, um\n<<
        << »das Programm zu beenden\n<<
```

Lektion 15 – Zeiger auf Objekte 167

```

// erzeuge (weiteres) NameDataSet-Objekt
NameDataSet* pNDS;
while (pNDS = getData())
{
    // füge es an die Liste der
    // NameDataSet-Objekte an
    addTail(pNDS);
}

// um die Objekte anzuzeigen, iteriere durch die
// Liste (stoppe, wenn die nächste Adresse
// null ist)
cout << »Entries:\n«;
pNDS = pHead;
while(pNDS)
{
    // Anzeige des aktuellen Eintrags
    displayData(pNDS);

    // gehe zum nächsten Eintrag
    pNDS = pNDS->pNext;
}
return 0;
}

```

Obwohl es in gewisser Hinsicht lang ist, ist das Programm `LinkedListData` doch recht einfach. Die Funktion `main()` beginnt mit dem Aufruf von `getData()`, um einen `NameDataSet`-Eintrag vom Benutzer zu bekommen. Wenn der Benutzer `ende` eingibt, gibt `getData()` `null` zurück. `main()` ruft dann `addTail()` auf, um den Eintrag, der von `getData()` zurückgegeben wurde, an das Ende der verketteten Liste anzufügen.

Sobald es vom Benutzer keine weiteren `NameDataSet`-Objekte gibt, iteriert `main()` durch die Liste, und zeigt jedes Objekt mittels der Funktion `displayData()` an.

Die Funktion `getData()` alloziert zuerst ein leeres `NameDataSet`-Objekt von Heap. `getData()` liest dann den Vornamen des einzufügenden Eintrags. Wenn der Benutzer als Vornamen `ende` oder `ENDE` eingibt, löscht die Funktion das Objekt, und gibt `null` an den Aufrufenden zurück. `getData()` fährt mit dem Lesen des Nachnamens und der Sozialversicherungsnummer fort. Schließlich setzt `getData()` den Zeiger `pNext` auf `null`, bevor die Funktion zurückkehrt.



Lassen Sie Zeiger niemals uninitialisiert. Wenden Sie die alte Programmierregel an: »Im Zweifelsfalle ausnullen«.

Die Funktion `addTail()`, die hier auftaucht, ist der Funktion `addTail()` sehr ähnlich, die bereits in diesem Kapitel dargestellt wurde. Anders als die ältere Version, überprüft diese Version von `addTail()`, ob die Liste leer ist, bevor sie startet. Wenn `pHead` `null` ist, dann setzt `addTail()` den Zeiger `pHead` auf den aktuellen Eintrag und terminiert.

Die Funktion `displayData()` ist eine Zeiger-basierte Version der früheren Funktionen `displayData()`.



0 Min.

15.5 Andere Container

Ein *Container* ist eine Struktur, die entworfen wurde, um Objekte zu enthalten. Arrays und verkettete Listen sind spezielle Container. Der Heap ist auch eine Form Container; er enthält einen separaten Speicherblock, der dem Programm zur Verfügung steht.

Sie haben möglicherweise von anderen Containern gehört, wie z.B. der FIFO (first-in-first-out) und der LIFO (last-in-first-out); der Container LIFO wird auch als Stack (Stapel) bezeichnet. Diese stellen zwei Funktionen bereit, jeweils eine für das Einfügen und das Löschen von Objekten. Die FIFO entfernt das älteste Objekt, während die LIFO das jüngste Objekt entfernt.

Zusammenfassung

Zeiger auf Klassenobjekte machen es möglich, den Wert von Klassenobjekten innerhalb von Funktionen zu verändern. Eine Referenz auf ein Klassenobjekt zu übergeben, ist bedeutend schneller, als ein Klassenobjekt als Wert zu übergeben. Ein Zeigerelement in eine Klasse einzufügen, ermöglicht eine Verkettung der Objekte in einer verketteten Liste. Die Struktur der verketteten Liste bietet mehrere Vorteile gegenüber Arrays, während andererseits Effizienz eingebüßt wird.

- Zeiger auf Klassenobjekte arbeiten im Wesentlichen wie Zeiger auf andere Datentypen. Dies umfasst die Fähigkeit, Objekte als Referenz an eine Funktion zu übergeben.
- Ein Zeiger auf ein lokales Objekt ist nicht mehr gültig, wenn die Kontrolle von der Funktion zurückgegeben wurde. Objekte, die vom Heap alloziert wurden, haben keine solche Beschränkung ihres Gültigkeitsbereiches und können daher von Funktion zu Funktion übergeben werden. Es obliegt jedoch dem Programmierer, Objekte, die vom Heap alloziert wurden, an den Heap zurückzugeben; geschieht dies nicht, sind fatale und schwer zu findende Speicherlöcher die Folge.
- Objekte können in einer verketteten Liste miteinander verbunden werden, wenn ihre Klasse einen Zeiger auf ein Objekt ihres eigenen Typs enthält. Es ist einfach, Elemente in die verkettete Liste einzufügen, und Elemente aus der verketteten Liste zu löschen. Obwohl wir das hier nicht gezeigt haben, ist das Sortieren einer verketteten Liste einfacher als das Sortieren eines Array. Objektzeiger sind auch nützlich bei der Erzeugung anderer Container, die hier nicht vorgestellt wurden.

Selbsttest

1. Gegeben sei die folgende Klasse:

```
class MyClass
{
    int n;
}
MyClass* pM;
```

Wie würden Sie das Datenelement `n` vom Zeiger `pM` referenzieren? (Siehe »Zeiger auf Objekte«)

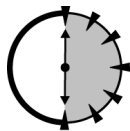
2. Was ist ein Container? (Siehe »Andere Container«)
3. Was ist eine verkettete Liste? (Siehe »Verkettete Listen«)
4. Was ist ein Kopfzeiger? (Siehe »Verkettete Listen«)

Debuggen II



Checkliste

- ☒ Schrittweise durch ein Programm
- ☒ Haltepunkte setzen
- ☒ Variablen ansehen und modifizieren
- ☒ Ein Programm mit einem Debugger debuggen



30 Min.

Sitzung 10 hat eine Technik zum Debuggen von Programmen vorgestellt, die auf der Ausgabe von Schlüsselinformationen auf `cout` basiert. Wir haben diese so genannte Technik der Ausgabeanweisungen verwendet, um das zugegebenermaßen einfache Beispielprogramm `ErrorProgram` zu debuggen.

Für kleinere Programme arbeitet diese Technik gut. Probleme mit dieser Technik treten erst auf, wenn der Umfang der Programme über den hier dargestellten Beispielprogramme hinausgeht.

In größeren Programmen weiß der Programmierer oft nicht, wo er Ausgabeanweisungen hinsetzen muss. Ein strenger Zyklus von Einfügen von Ausgabeanweisungen, Erzeugen des Programms, Ausführen des Programms, Einfügen von Ausgabeanweisungen usw. ist nervig. Um Ausgabeanweisungen zu verändern, muss das Programm stets neu erzeugt werden. Bei einem großen Programm kann selbst die Zeit für die Erzeugung schon beachtlich sein.

Ein zweiter, ausgeklügelterer Ansatz basiert auf einem separaten Werkzeug, dem Debugger. Dieser Zugang vermeidet viele der Nachteile, die in der Technik der Ausgabeanweisungen enthalten sind. Diese Sitzung führt Sie in die Verwendung des Debuggers ein, indem wir den Bug in einem kleinen Programm finden.



Ein großer Teil dieses Buches ist dem Studium der Programmierfähigkeiten gewidmet, die durch Zeigervariablen ermöglicht werden. Zeiger haben jedoch auch ihren Preis: Zeigerfehler sind leicht zu begehen, und extrem schwierig zu finden. Die Technik der Ausgabeanweisungen taugt für das Finden und Entfernen von Zeigerfehlern nicht. Nur ein guter Debugger kann bei solchen Fehlern helfen.

170 Samstagnachmittag**16.1 Welcher Debugger?**

Anders als bei der Programmiersprache C++, die über Herstellergrenzen hinweg standardisiert ist, hat jeder Debugger seine eigenen Kommandos. Glücklicherweise bieten die meisten Debugger die gleichen elementaren Kommandos. Die Kommandos, die wir benötigen, sind sowohl in Visual C++ als auch in der `rhide`-Umgebung von GNU C++ enthalten. Beide Umgebungen bieten diese Basis-kommandos über Pulldown-Menüs an. Beide Debugger bieten auch den schnellen Zugriff auf die wichtigsten Debuggerfunktionen über Tastenkombinationen. Tabelle 16-1 listet diese Kommandos für beide Umgebungen auf.

Im Rest dieser Sitzung werden ich Debugkommandos über ihren Namen ansprechen. Verwenden Sie Tabelle 16-1 um die entsprechende Tastenkombination zu finden.

Tabelle 16-1: Debuggerkommandos für Visual C++ und GNU C++

Kommando	Visual C++	GNU C++ (rhide)
Build	Shift+F8	F9
Step In	F11	F7
Step Over	F10	F8
View Variable	siehe Text	Ctl+F4
Set Breakpoint	F9	Ctl+F8
Add Watch	siehe Text	Ctl+F7
Go	F5	Ctl+F9
View User Screen	Klicken auf Programmfenster	Alt+F5
Program Reset	Shift+F5	Ctl+F2

Um Verwirrung hinsichtlich der kleinen Unterschiede der beiden Debugger zu vermeiden, beschreibe ich den Debugprozess, den ich mit `rhide` verfolgt habe. Danach debugge ich das gleiche Programm noch einmal mit dem Debugger von Visual C++.

16.2 Das Testprogramm

Ich habe das folgende Programm geschrieben, das einen Bug (Fehler) enthält. Das Schreiben fehlerhafter Programme ist für mich nicht schwierig, weil meine Programme fast nie beim ersten Mal laufen.



Die Datei ist auf der beiliegenden CD-ROM unter dem Namen `Concatenate(Error).cpp` zu finden.

```
// Concatenate - verbinde zwei Zeichenketten
//          mit » - « in der Mitte
//          (diese Version stürzt ab)
#include <stdio.h>
#include <iostream.h>
void concatString(char szTarget[], char szSource[]);

int main(int nArg, char* pszArgs[])
{
    cout << »Dieses Programm verbindet zwei Zeichenketten\n«;
    cout << »(Diese Version stürzt ab)\n\n«;

    // lese erste Zeichenkette ...
    char szString1[256];
    cout << »Zeichenkette #1:«;
    cin.getline(szString1, 128);

    // ... nun die zweite Zeichenkette ...
    char szString2[128];
    cout << »Zeichenkette #2:«;
    cin.getline(szString2, 128);

    // ... füge » - « an die erste an ...
    concatString(szString1, » - «);

    // ... nun füge zweite an ...
    concatString(szString1, szString2);

    // ... und zeige das Resultat
    cout << »\n« << szString1 << »\n«;

    return 0;
}

// concatString - fügt die Zeichenkette szSource
//          an das Ende von szTarget an
void concatString(char szTarget[], char szSource[])
{
    int nTargetIndex;
    int nSourceIndex;

    // finde das Ende der ersten Zeichenkette
    while(szTarget[++nTargetIndex])
    {
    }

    // füge die zweite ans Ende der ersten an
    while(szSource[nSourceIndex])
    {
        szTarget[nTargetIndex] =
            szSource[nSourceIndex];
        nTargetIndex++;
        nSourceIndex++;
    }
}
```

172 Samstagnachmittag

Das Programm kann ohne Fehler erzeugt werden. Ich führe das Programm aus. Das Programm fragt nach Zeichenkette #1, und ich gebe »das ist eine Zeichenkette« ein. Für Zeichenkette #2 gebe ich DAS IST EINE ZEICHENKETTE ein. Aber anstatt die korrekte Ausgabe zu erzeugen, bricht das Programm mit Fehlercode 0xff ab. Ich drücke OK. Der Debugger versucht, mir ein wenig Trost zu spenden, indem er das Fenster unterhalb des Eingabefensters öffnet, wie in Abbildung 16.1 zu sehen ist.

Die erste Zeile im Nachrichtenfenster zeigt an, dass rhide denkt, dass der Fehler in Zeile 46 des Moduls Concatenate(error1) aufgetreten ist. Außerdem wurde die Funktion, die abgestürzt ist, von Zeile 29 im gleichen Modul aufgerufen. Das weist scheinbar darauf hin, dass die initiale while-Schleife innerhalb von concatString() fehlerhaft ist.

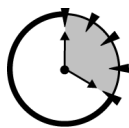


Abbildung 16.1: Der rhide-Debugger gibt einen Hinweis auf die Fehlerquelle, wenn ein Programm abstürzt.

Weil ich in dieser Anweisung keinen Fehler finden kann, mache ich den Debugger zu meinem Helfen.



Tatsächlich sehe ich das Problem, ausgehend von den Informationen, die rhide mir liefert. Doch gehen Sie weiter mit mir durch.



20 Min.

16.3 Einzelschritte durch ein Programm

Ich drücke Step Over, um mit dem Debuggen des Programms zu beginnen. rhide öffnet ein MS-DOS-Fenster, als wenn es das Programm ausführen wollte. Doch bevor das Programm mit der Ausführung beginnen kann, schließt der Debugger das Programmfenster und zeigt das Editierfenster des Programms an, in dem die erste ausführbare Zeile des Programms markiert ist.

Eine *ausführbare Anweisung* ist eine Anweisung, die keine Deklaration oder Kommentar ist. Eine ausführbare Anweisung ist eine Anweisung, die beim Kompilieren Maschinencode erzeugt.

Der Debugger hat tatsächlich das Programm ausgeführt bis zur ersten Zeile der Funktion main() und dann dem Programm die Kontrolle entzogen. Der Debugger wartet auf Sie, zu entscheiden, wie es weitergehen soll.

Indem ich mehrfach Step Over drücke, kann ich das Programm ausführen, bis es zum Absturz kommt. Das sagt mir viel darüber, was falsch gelaufen ist.

Ein Programm zeilenweise auszuführen wird auch als *Einzelschrittausführung* eines Programms bezeichnet.

Als ich Step Over für das Kommando `cin.getline( )` ausführe, bekommt der Debugger die Kontrolle vom MS-DOS-Fenster nicht wie üblich zurück. Stattdessen scheint das Programm beim Prompt eingefroren zu sein und darauf zu warten, dass die erste Zeichenkette eingegeben wird.

In der Nachbetrachtung merke ich, dass der Debugger die Kontrolle vom Programm erst dann zurückbekommt, wenn die C++-Anweisung ausgeführt ist – eine Anweisung, die den Aufruf `getline( )` enthält, ist erst dann ausgeführt, wenn ich einen Text über die Tastatur eingegeben habe.

Ich gebe eine Zeile Text ein und drücke die Enter-Taste. Der rhide-Debugger hält das Programm bei der nächsten Anweisung an: `cout << »Zeichenkette #2:«`. Wieder führe ich einen Einzelschritt aus, indem ich die zweite Zeile Text eingebe als Antwort auf den zweiten Aufruf von `getline( )`.



Wenn der Debugger anzuhalten scheint, ohne zurückzukommen, wenn Sie in Einzelschritten durch ein Programm gehen, wartet Ihr Programm darauf, dass etwas Bestimmtes passiert. Am wahrscheinlichsten ist, dass das Programm auf eine Eingabe wartet, entweder von Ihnen oder von einem externen Device.

Schließlich gehe ich im Einzelschrittmodus durch den Aufruf von `concatString( )`, wie in Abbildung 16.2 zu sehen ist. Als ich Step Over für den Aufruf versuche, stürzt das Programm wie zuvor ab.

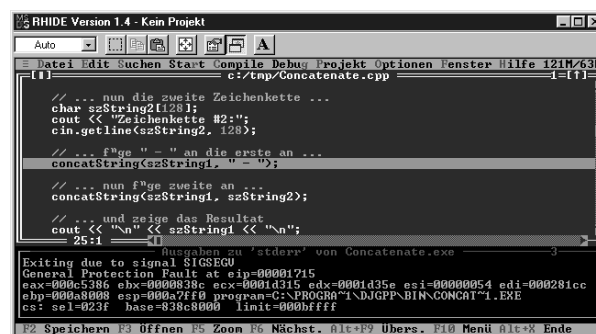


Abbildung 16.2: Etwas in der Funktion `concatString( )` verursacht den Absturz.

Das sagt mir nicht mehr als ich schon vorher wusste. Was ich brauche, ist die Möglichkeit, Einzelschritte in der Funktion auszuführen, statt einen Schritt über die Funktion hinweg zu machen.

16.4 Einzelschritte in eine Funktion hinein

Ich entscheide mich für einen weiteren Versuch. Erst drücke ich Reset, um den Debugger am Anfang des Programms zu starten.



Denken Sie immer daran, Program Reset zu drücken, bevor Sie erneut beginnen. Es tut nicht weh, den Knopf oft zu drücken. Sie können sich daran gewöhnen, Program Reset jedes Mal auszuführen, bevor Sie den Debugger starten.

174 Samstagnachmittag

Wieder gehe ich in Einzelschritten durch das Programm unter Verwendung von Step Over, bis ich den Aufruf von `concatString( )` erreiche. Diesmal rufe ich nicht Step Over, sondern Step In auf, um in die Funktion zu gelangen. Sofort bewegt sich die Markierung zur ersten ausführbaren Zeile innerhalb von `concatString( )` wie in Abbildung 16.3 zu sehen ist.



Es gibt keinen Unterschied zwischen Step Over und Step In, wenn kein Funktionsaufruf ausgeführt wird.

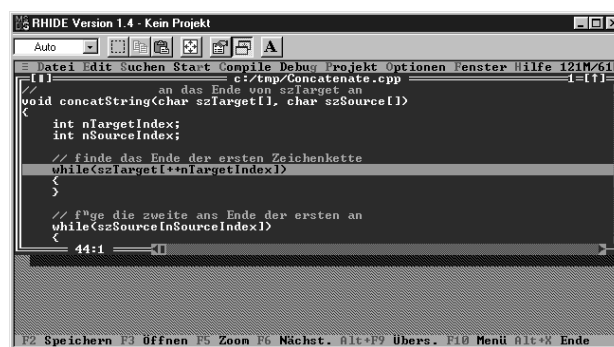


Abbildung 16.3: Das Kommando Step In bewegt die Kontrolle zur ersten ausführbaren Zeile von `concatString( )`.



Wenn Sie Step In versehentlich für eine Funktion ausgeführt haben, kann es sein, dass der Debugger Sie nach der Quelldatei einer Datei fragen wird, von der Sie vorher noch nie gehört haben. Das ist die Datei des Bibliothekmoduls, das die Funktion enthält, in die Sie hineingehen wollten. Drücken Sie Cancel, und Sie erhalten eine Liste von Maschineninstruktionen, die selbst für die härtesten Techniker nicht sehr hilfreich ist. Um wieder in einen gesunden Zustand zu kommen, öffnen Sie das Eingabefenster, setzen einen Haltepunkt, wie im nächsten Abschnitt beschrieben, auf die Anweisung direkt nach dem Aufruf und drücken Go.

Mit großer Hoffnung drücke ich Step Over, um die erste Anweisung in der Funktion auszuführen. Der `rhide`-Debugger antwortet mit der Meldung eines Segmentfehlers wie in Abbildung 16.4 zu sehen ist.



Abbildung 16.4: Ein Einzelschritt auf der ersten Zeile von `concatString()` erzeugt einen Segmentfehler.



Ein Segmentfehler zeigt im Allgemeinen an, dass ein Programm auf ein ungültiges Speichersegment zugegriffen hat, entweder weil ein Zeiger ungültig geworden ist, oder ein Array außerhalb seiner Grenzen adressiert wurde. Um es interessanter zu machen, lassen Sie uns annehmen, dass ich das nicht weiß.

Jetzt weiß ich sicher, dass etwas in der `while`-Schleife nicht korrekt ist, und dass bereits die erste Ausführung der Schleife zum Absturz führt. Um herauszufinden, was schiefgeht, muss ich das Programm unmittelbar vor der fehlerhaften Zeile anhalten.

16.5 Verwendung von Haltepunkten

Wieder drücke ich Program Reset, um den Debugger auf den Anfang des Programms zurückzubringen. Ich könnte wieder in Einzelschritten durch das Programm gehen, bis ich auf die `while`-Schleife treffe. Stattdessen wähle ich eine Abkürzung. Ich platziere den Cursor auf der `while`-Anweisung und führe das Kommando Set Breakpoint aus. Der Editor markiert die Anweisung rot, wie in Abbildung 16.5 zu sehen ist.

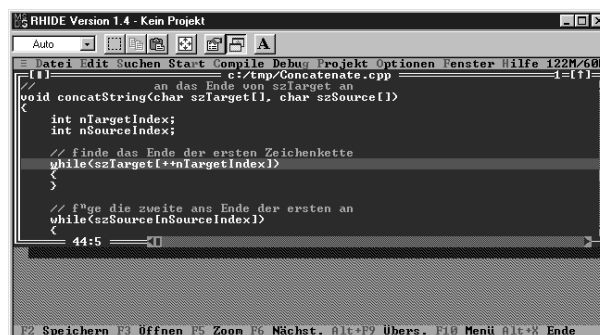
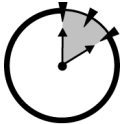


Abbildung 16.5: Das Kommando Set Breakpoint.

Ein Haltepunkt teilt dem Debugger mit, bei dieser Anweisung anzuhalten, wenn die Kontrolle jemals dorthin gelangt. Ein Haltepunkt lässt ein Programm wie gewohnt ablaufen bis zu diesem Punkt, an dem wir die Kontrolle übernehmen möchten. Haltepunkte sind nützlich, wenn wir wissen, wo wir anhalten möchten, oder wenn wir das Programm normal ausführen möchten, bis es Zeit zum Anhalten ist.

176 Samstagnachmittag

Nachdem ich den Haltepunkt gesetzt habe, drücke ich Go. Das Programm scheint normal ausgeführt zu werden bis zum Aufruf von `while`. An diesem Punkt springt das Programm zurück zum Debugger.

**10 Min.****16.6 Ansehen und Modifizieren von Variablen**

Es macht nicht viel Sinn, die `while`-Anweisung wieder auszuführen – ich weiß ja, dass sie zum Absturz führen wird. Ich brauche mehr Informationen darüber, was das Programm macht, um festzustellen, was zum Absturz führt. Z.B. möchte ich gerne den Inhalt der Variablen `nTargetIndex` unmittelbar vor der Ausführung der `while`-Schleife sehen.

Zuerst führe ich einen Doppelklick auf dem Variablennamen `nTargetIndex` aus. Dann drücke ich View Variable. Ein Fenster erscheint, mit dem Namen `nTargetIndex` im oberen Feld. Ich drücke Eval, um den aktuellen Wert der Variable herauszufinden. Das Ergebnis, das Sie in Abbildung 1.6 finden, macht offensichtlich keinen Sinn.

Bei einem Blick zurück in den C++-Code stelle ich fest, dass ich die Variablen `nTargetIndex` und `nSourceIndex` nicht initialisiert habe. Um das zu überprüfen, gebe ich 0 im Fenster New Value ein und drücke Change. Ich führe das Gleiche für `nSourceIndex` aus. Ich schließe das Fenster und drücke Step Over, um die Ausführung fortzusetzen.

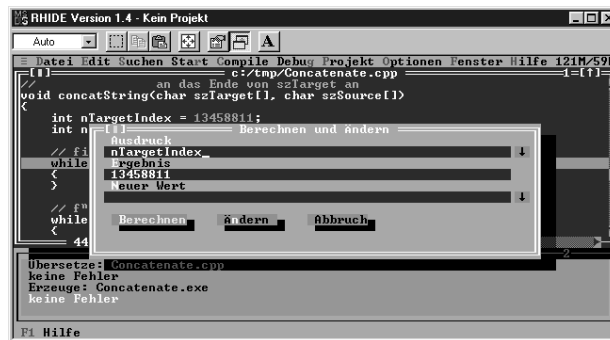


Abbildung 16.6: Dieses Fenster erlaubt es dem Programmierer, gültige Variablen anzusehen und ihren Wert zu verändern.

Mit den nun initialisierten Indexvariablen gehe ich in Einzelschritten durch die `while`-Schleife hindurch. Jeder Aufruf von Step Over oder Step In führt eine Iteration der `while`-Schleife aus. Weil der Cursor nach dem Aufruf da steht, wo er vorher auch war, tritt scheinbar keine Veränderung auf; nach einem Schleifendurchlauf hat `nTargetIndex` jedoch den Wert 1.

Weil ich mir nicht die Arbeit machen möchte, den Wert von `nTargetIndex` nach jeder Iteration zu überprüfen, führe ich einen Doppelklick auf `nTargetIndex` aus, und führe das Kommando Add Watch aus. Es erscheint ein Fenster mit der Variable `nTargetIndex` und dem Wert 1 rechts daneben. Ich drücke mehrmals Step In, und in jeder Iteration wird der Wert von `nTargetIndex` um eins erhöht. Nach einigen Iterationen wird die Kontrolle an eine Anweisung außerhalb der Schleife übergeben.

Lektion 16 – Debuggen II 177

Ich setze einen Haltepunkt auf die schließende Klammer der Funktion `concatString()` und drücke Go. Das Programm hält unmittelbar vor der Rückkehr der Funktion an.

Um die erzeugte Zeichenkette zu überprüfen, führe ich einen Doppelklick auf `szTarget` aus und drücke View Variable. Die Ergebnisse, die in Abbildung 16.7 zu sehen sind, sind nicht so, wie ich es erwartet habe.



Die 0x6ccee0 ist die Adresse der Zeichenkette im Speicher. Diese Information kann hilfreich sein, wenn es um Zeiger geht. Diese Information könnte z.B. sehr hilfreich sein beim Debuggen einer Anwendung, die verkettete Listen verwendet.

Es sieht so aus, als ob die Zielzeichenkette nicht verändert worden wäre, obwohl ich genau weiß, dass die zweite `while`-Schleife ausgeführt wurde. Mit einer kleinen Chance, dass die zweite Zeichenkette doch da ist, sehe ich hinter der initialen Zeichenkette nach. `szTarget + 27` sollte die Adresse des ersten Zeichens nach der Null der Zeichenkette »DIES IST EINE ZEICHENKETTE«, die ich eingegeben habe, sein. Und tatsächlich, das »-« steht da, gefolgt von einem »d«, das korrekt zu sein scheint. Das ist in Abbildung 16.8 zu sehen.



Abbildung 16.7: Die Zielzeichenkette scheint nach Rückkehr der Funktion `concatString()` nicht verändert zu sein.

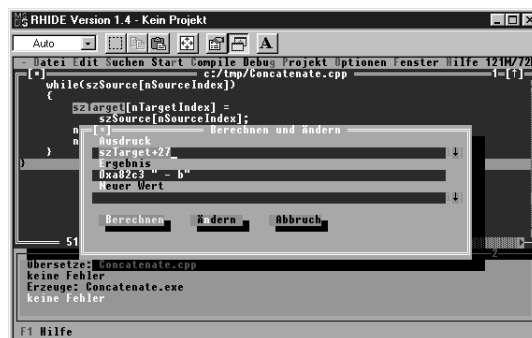


Abbildung 16.8: Die Quellzeichenkette scheint an einer falschen Stelle an die Zielzeichenkette angefügt worden zu sein.

178 Samstagnachmittag

Nach reiflicher Überlegung ist es offensichtlich, dass `szSource` hinter dem abschließenden `Null`-zeichen an `szTarget` angefügt wurde. Zusätzlich ist klar, dass die resultierende Ergebniszeichenkette überhaupt nicht abgeschlossen wurde (daher das »D« am Ende).



Eine Zeichenkette hinter dem Nullzeichen zu verändern, oder das terminierende Nullzeichen zu vergessen, sind die häufigsten Fehler beim Umgang mit Zeichenketten.

Weil ich jetzt zwei Fehler kenne, drücke ich Program Reset und berichtige die Funktion `concatString()`, so dass die zweite Zeichenkette an der richtigen Stelle eingefügt wird und die Ergebniszeichenkette mit einem Nullzeichen abgeschlossen wird. Die geänderte Funktion `concatString()` sieht wie folgt aus:

```
void concatString(char szTarget[], char szSource[])
{
    int nTargetIndex = 0;
    int nSourceIndex = 0;

    // finde das Ende der ersten Zeichenkette
    while(szTarget[nTargetIndex])
    {
        nTargetIndex++;
    }

    // füge die zweite ans Ende der ersten an
    while(szSource[nSourceIndex])
    {
        szTarget[nTargetIndex] =
            szSource[nSourceIndex];
        nTargetIndex++;
        nSourceIndex++;
    }

    // terminiere die Zeichenkette
    szTarget[nTargetIndex] = '\0';
}
```

Weil ich vermute, dass es noch ein weiteres Problem gibt, beobachte ich `szTarget` und `nTargetIndex`, während die zweite Schleife ausgeführt wird. Nun wird die Zeichenkette korrekt ans Ende der Zielzeichenkette kopiert, wie in Abbildung 16.9 zu sehen ist. (Abbildung 16.9 zeigt den zweiten Aufruf von `concatString()`, weil er besser zu verstehen ist.)



Sie müssen das wirklich selber ausführen. Das ist der einzige Weg, wie Sie ein Gefühl dafür bekommen können, wie hübsch es ist, einer Zeichenkette beim Wachsen zuzusehen, während die andere Zeichenkette in jeder Iteration der Schleife schrumpft.

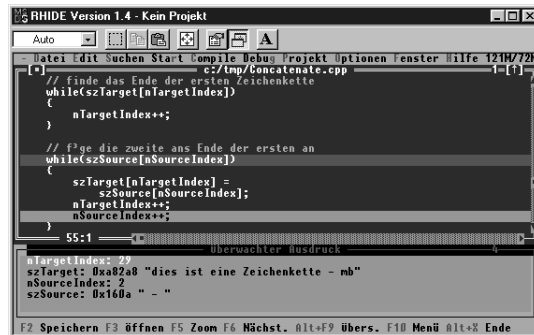


Abbildung 16.9: Zeigt wie die Quellzeichenkette an das Ende der Zielzeichenkette gehängt wird.

Bei der erneuten Untersuchung der Zeichenkette unmittelbar vor dem Anfügen der terminierenden Null, stelle ich fest, dass die Zeichenkette `szTarget` korrekt ist mit Ausnahme des Extrazeichens am Ende, wie in Abbildung 16.10 zu sehen ist.

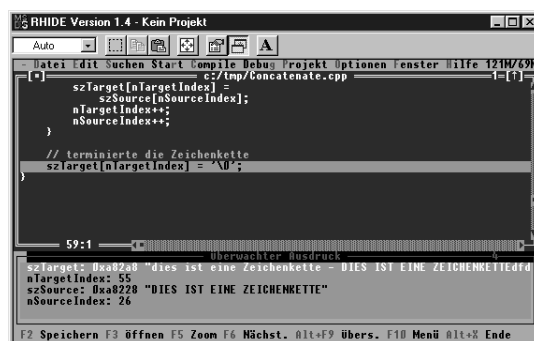


Abbildung 16.10: Vor dem Hinzufügen des Nullzeichens enthält die Ergebniszeichenkette am Ende weitere Zeichen.

Sobald ich Step Over drücke, fügt das Programm das Nullzeichen an und die »Rauschzeichen« verschwinden aus dem Fenster.

16.7 Verwendung des Visual C++ Debuggers

Die Schritte, um das Programm `Concatenate` mit den Debugger von Visual C++ zu debuggen, sind ähnlich zu den Schritten, die wir mit `rhide` ausgeführt haben. Der Hauptunterschied ist jedoch, dass der Debugger von Visual C++ das auszuführende Programm in einem separaten MS-DOS-Fenster öffnet, statt als Teil des Debuggers selber. Wenn Sie Go drücken, erscheint in der Programmleiste von Windows ein neues Icon, das den Namen des Programms zeigt, in diesem Fall `Concatenate`. Der Programmierer kann das Benutzerfenster ansehen, indem er das Programmfenster auswählt.

180 Samstagnachmittag

Ein zweiter Unterschied ist die Art und Weise, mit der der Visual C++-Debugger das Ansehen von Variablen löst. Wenn die Ausführung an einem Haltepunkt gestoppt ist, kann der Programmierer einfach den Cursor über eine Variable bewegen. Wenn die Variable im Gültigkeitsbereich ist, zeigt der Debugger ihren Wert in einem kleinen Fenster, wie in Abbildung 16.11 zu sehen ist.

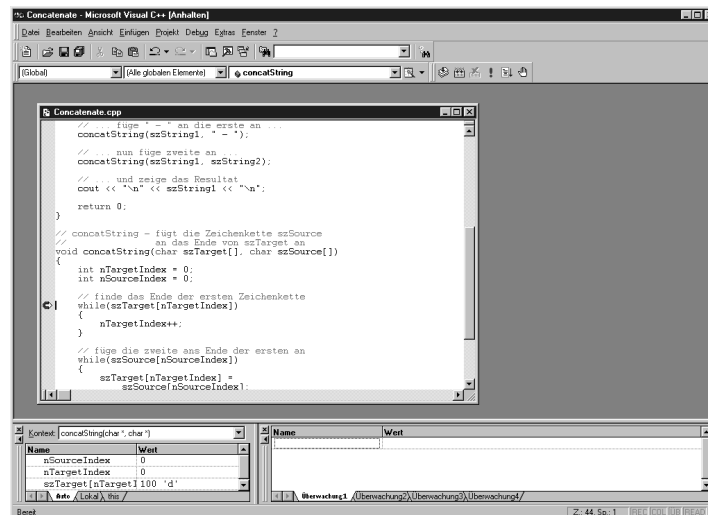


Abbildung 16.11: Visual C++ zeigt den Wert einer Variable, wenn der Cursor über sie bewegt wird.



0 Min.



Hinweis

Zusätzlich bietet der Debugger von Visual C++ einen bequemen Weg, um lokale Variablen (das sind Variable, die lokal in einer Funktion deklariert sind) anzusehen. Wählen Sie View, dann Debug Windows und schließlich Variables. Vom Fenster Variables aus wählen Sie die Locals-Schaltfläche. Alternativ können Sie Alt+4 drücken. Dieses Fenster markiert sogar die Variablen, die seit dem letzten Haltepunkt verändert wurden. Abbildung 16.12 zeigt dieses Fenster, während Einzelschritte durch das Kopieren der Quelle in die Zielzeichenkette ausgeführt werden.

Das Zeichen »|« am Ende von szTarget spiegelt die Tatsache wieder, dass die Zeichenkette noch nicht terminiert wurde.

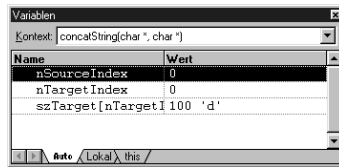


Abbildung 16.12: Dieses Fenster des Visual C++-Debuggers zeigt den Wert von Variablen, während Einzelschritte ausgeführt werden.

Zusammenfassung

Lassen Sie uns den Debugger-Zugang zum Finden eines Problems vergleichen mit dem Zugang über Ausgabeanweisungen, den wir in Sitzung 10 eingeführt haben. Der Debugger-Zugang ist nicht einfach zu erlernen. Ich bin mir sicher, dass Ihnen viele der hier eingegebenen Kommandos fremd vorgekommen sind. Wenn Sie sich jedoch erst einmal an den Debugger gewöhnt haben, können Sie ihn nutzen, um viel über Ihr Programm zu erfahren. Die Fähigkeit, langsam durch Ihr Programm durchzugehen, während Sie Variablen ansehen und verändern können, ist ein mächtiges Werkzeug.



Ich bevorzuge es, den Debugger beim ersten Aufruf eines neuen Programms aufzurufen. Schrittweise durch ein Programm durchzugehen, schafft ein gutes Verständnis dafür, was wirklich ausgeführt wird.

Ich war gezwungen, das Programm mehrmals auszuführen, als ich den Debugger verwendet habe. Das Programm musste ich jedoch nur einmal kompilieren und erzeugen, obwohl ich mehr als einen Fehler gefunden habe. Das ist ein großer Vorteil, wenn Sie ein großes Programm debuggen, das einige Minuten für seine Erzeugung benötigt.



Ich habe schon an Projekten mitgearbeitet, bei denen der Computer die ganze Nacht damit beschäftigt war, das System neu zu erzeugen. Während das eher die Ausnahme ist, sind 5 bis 30 Minuten für die Erzeugung realer Applikationen nicht außergewöhnlich.

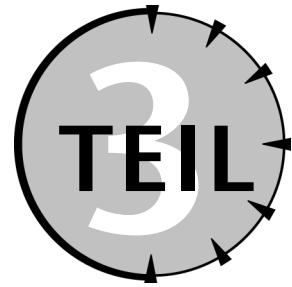
Schließlich gibt der Debugger Ihnen Zugriff auf Informationen, die Sie nicht so einfach sehen könnten, wenn Sie den Zugang der Ausgabeanweisungen wählen. Z.B. ist das Ansehen eines Zeigerinhaltes einfach mit einem Debugger. Es ist zwar möglich, doch sehr umständlich, immer wieder Adresseninformationen auszugeben.

182 *Samstagnachmittag*

Selbsttest

1. Was ist der Unterschied zwischen Step Over und Step In? (Siehe »Einzelschritte in eine Funktion hinein«)
2. Was ist ein Haltepunkt? (Siehe »Verwendung von Haltepunkten«)
3. Was ist eine Watch? (Siehe »Ansehen und Modifizieren von Variablen«)

Samstagnachmittag – Zusammenfassung



1. Definieren Sie eine Studentenkasse, die den Nachnamen des Studenten, den Grad (erster Grad, zweiter Grad, usw.) und den mittleren Grad speichern kann.
2. Schreiben Sie eine Funktion, um ein Studentenobjekt zu lesen und seine Informationen auszugeben.
3. Geben Sie drei Grade ein, und mitteln Sie diese, bevor Sie das Objekt wieder ausgeben.

Hinweise:

a. Verwenden Sie eine Abbildung von Zahlen auf Grade. Z.B. sei 1 der erste Grad, 2 der zweite Grad, usw.

b. Der mittlere Grad ist eine Gleitkommazahl.

4. Wenn die folgenden Variablen im Speicher ohne Zwischenräume angeordnet sind, wie viele Bytes Speicher beansprucht jede Variable in einem Programm, das mit Visual C++ oder GNU C++ erzeugt wurde:

a. `int n1; long l1; double d1;`

b. `int nArray[20];`

c. `int* pnPt1; double* pdPt2;`

5. Betrachten Sie die folgende Funktion:

```
void fn(int n1)
{
    int* pnVar1 = new int;
    *pnVar1 = n1;
}
```

- a. Kann die Funktion kompiliert werden?
- b. Was ist falsch an ihr?
- c. Warum könnte die Funktion Probleme verursachen?
- d. Was macht diese Art Problem so schwer auffindbar?

6. Beschreiben Sie die Speicheranordnung von `double dArray[3]`. Nehmen Sie an, dass das Array bei Adresse `0x100` beginnt.
7. Unter Verwendung des Arrays in Aufgabe 6, beschreiben Sie den Effekt des Folgenden:

```
double dArray[3];
double* pdPtr = &dArray[1];
*pdPtr = 1.0; // Zuweisung #1
int* pnPtr = (int*)&dArray[2];
*pnPtr = 2; // Zuweisung #2
```

8. Schreiben Sie eine Funktion `LinkableClass* removeHead( )`, die das erste Element einer Liste von `LinkableClass`-Objekten entfernt, und an den Aufrufenden zurückgibt.

Hinweise:

- Vergessen Sie nicht, dass die Liste bereits leer sein könnte.
 - Geben Sie null zurück, wenn die Liste leer ist.
 - Wen Sie Schwierigkeiten mit leeren Listen haben, fangen Sie damit an, dass Sie die Liste als nicht leer annehmen. Nachdem Ihre Funktion fertig ist, versuchen Sie den Spezialfall einzubauen, dass die Liste leer ist.
9. Schreiben Sie eine Funktion `LinkableClass* returnPrevious(LinkableClass* pTarget)`, die den Vorgänger von `pTarget` in einer verketteten Liste zurückgibt, d.h. den Eintrag in der Liste, der auf `pTarget` verweist. Geben Sie null zurück, wenn die Liste leer ist, oder `pTarget` nicht gefunden wurde. Denken Sie wieder daran, auf das Ende der Liste zu achten.
10. Schreiben Sie eine Funktion `LinkableClass* returnTail( )`, die den letzten Eintrag entfernt und an den Aufrufenden zurückgibt.

Hinweis: Erinnern Sie sich daran, dass der `pNext`-Zeiger des letzten Elementes gleich `null` ist.

Zusatzaufgabe: Schreiben Sie eine Funktion `LinkableClass* removeTail( )`, die das letzte Element der Liste entfernt, und dieses Element zurückliefert.

Hinweise:

- Versuchen Sie, die Funktion `returnPrevious( )` zu verwenden. Sie sollte in der Lage sein, die meiste Arbeit zu erledigen.
- Wenn der Vorgängereintrag des letzten Elements null ist, dann hat die Liste nur einen Eintrag.

- 11. Updaten Sie das Programm Concatenate mit der folgenden Zeigerversion von `concatString`, nachdem Sie mittels eines Debuggers (GNU C++ oder Visual C++) den darin enthaltenen Fehler entfernt haben:**

```
void concatString(char* pszTarget, char* pszSource)
{
    // hänge die zweite ans Ende der ersten
    while(*pszTarget)
    {
        *pszTarget++ = *pszSource++;
    }

    // terminiere die Zeichenkette
    *pszTarget = '\0';
}
```

Samstagabend

Teil 4

Lektion 17

Objektprogrammierung

Lektion 18

Aktive Klassen

Lektion 19

Erhalten der Klassenintegrität

Lektion 20

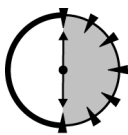
Klassenkonstruktoren II

Objekt- programmierung



Checkliste

- ☒ Objekte in der realen Welt identifizieren
- ☒ Objekte in Klassen klassifizieren
- ☒ Objektorientierter und funktionaler Ansatz im Vergleich
- ☒ Nachos herstellen



30 Min.

Beispiele für Objektprogrammierung können im täglichen Leben gefunden werden. In der Tat sind Objekte überall. Direkt vor mir steht ein Stuhl, ein Tisch, ein Computer und ein halbgeessenes Brötchen. Objektprogrammierung wendet diese Konzepte auf die Welt des Programmierens an.

17.1 Abstraktion und Mikrowellen

Manchmal, wenn mein Sohn und ich Fußball schauen, bereite ich mir Unmengen von Nachos zu, die ich für fünf Minuten in die Mikrowelle stelle. (Nachos sind eine Spezialität mit Chips, Bohnen, Käse und Jalapenos).

Um die Mikrowelle zu benutzen, öffne ich die Tür, stelle die Sachen rein und drücke ein paar Knöpfe. Nach ein paar Minuten sind die Nachos fertig.

Das klingt nicht sehr profund, aber denken Sie einige Minuten über alle die Dinge nach, die ich nicht tue, um die Mikrowelle zu benutzen:

- Ich schaue nicht in das Innere der Mikrowelle; ich schaue nicht das Listing des Codes an, den der zentrale Prozessor ausführt; ich studiere auch nicht den Schaltplan des Gerätes.
- Ich ändere nichts an der Mikrowelle, um sie zum Laufen zu bringen, auch nichts an ihrer Verkabelung. Die Mikrowelle hat ein Interface, das mich alles ausführen lässt, was ich brauche – die Frontplatte mit all den Knöpfen und der kleinen Zeitanzeige.
- Ich schreibe nicht das Programm neu, das auf dem kleinen Prozessor in der Mikrowelle läuft, selbst wenn ich beim letzten Mal was anderes gekocht habe als dieses Mal.

188 Samstagabend

- Selbst wenn ich ein Mikrowellenentwickler wäre, und alle Dinge über die internen Abläufe kennen würde, ihre Software eingeschlossen, würde ich nicht darüber nachdenken, wenn ich die Mikrowelle benutze.

Das sind keine profunden Beobachtungen. Über so vieles können wir nicht gleichzeitig nachdenken. Um die Anzahl der Dinge zu reduzieren, mit denen wir uns beschäftigen müssen, arbeiten wir auf einem bestimmten Detailniveau.

Mit objektorientierten (OO) Begriffen ausgedrückt, wird der Level, auf dem wir arbeiten, als *Abstraktionslevel* bezeichnet. Wenn wir mit Nachos arbeiten, betrachte ich meine Mikrowelle als Box. So lange ich die Mikrowelle über ihr Interface verwende (die Knopfplatte), sollte nichts, was ich tue, die Mikrowelle dahin bringen, dass sie

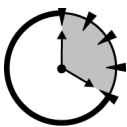
1. in einen inkonsistenten Zustand gerät und abstürzt
2. oder, was viel schlimmer wäre, mein Nacho in eine schwarze, brennende Masse verwandelt
3. oder, was am schlimmsten wäre, Feuer fängt und das Haus in Brand setzt.

17.1.1 Funktionale Nachos

Nehmen Sie an, ich sollte meinen Sohn bitten, einen Algorithmus zu schreiben, wie sein Vater Nachos macht. Nachdem er verstanden hätte, was ich von ihm will, würde er vermutlich etwas in der Art schreiben »öffne eine Dose Bohnen, reibe etwas Käse, schneide die Jalapenos« usw. Wenn es dann zu dem Teil mit der Mikrowelle käme, würde er etwas wie »fünf Minuten in der Mikrowelle kochen« schreiben.

Diese Beschreibung ist klar und vollständig. Aber es ist nicht die Art, in der ein funktionaler Programmierer ein Programm zur Herstellung von Nachos schreiben würde. Funktionale Programmierer leben in einer Welt, die frei ist von Objekten wie Mikrowellen und anderen Dingen. Sie kümmern sich um Flussdiagramme mit Myriaden von Pfaden. In einer funktionalen Lösung des Problems, Nachos herzustellen, würde die Kontrolle durch meinen Finger auf die Frontplatte fließen und dann in die internen Schaltungen der Mikrowelle. Dann würde die Kontrolle durch die komplexen logischen Pfade gehen, die dazu da sind, die Mikrowellenenergie anzuschalten und den Sound »fertig, hol mich raus« zu erzeugen.

In einer Welt wie dieser, ist es schwierig, von einem Abstraktionslevel zu sprechen. Es gibt keine Objekte und keine Abstraktion, hinter denen inhärente Komplexität versteckt werden könnte.

**20 Min.****17.1.2 Objektorientierte Nachos**

Bei objektorientierter Vorgehensweise, Nachos herzustellen, würden wir erst die Typen der Objekte in diesem Problem identifizieren: Chips, Bohnen, Käse und eine Mikrowelle. Dann würden wir mit der Aufgabe beginnen, diese Objekte in Software zu modellieren, ohne die Details zu berücksichtigen, wie sie im Programm verwendet werden.

Wenn wir Code auf Objektebene schreiben, arbeiten (und denken) wir auf einem Abstraktionsniveau der Basisobjekte. Wir müssen darüber nachdenken, eine nützliche Mikrowelle zu erstellen, wir müssen aber noch nicht über den logischen Prozess der Nacho-Zubereitung nachdenken. Schließlich haben die Designer der Mikrowelle auch nicht über das spezielle Problem nachgedacht, mir einen Snack zuzubereiten. Sie waren nur mit dem Problem befasst, eine nützliche Mikrowelle zu entwerfen und zu bauen.

Lektion 17 – Objektprogrammierung 189

Nachdem wir die Objekte erfolgreich codiert und getestet haben, können wir uns dem nächsten Abstraktionslevel zuwenden. Wir können auf dem Level der Nacho-Herstellung denken und nicht mehr auf Mikrowellenlevel. An dieser Stelle können wir die Anweisungen meines Sohnes leicht in C++-Code überführen.



Tatsächlich können wir die Leiter weiter hochsteigen. Der nächste Level nach oben könnte Dinge enthalten wie aufstehen, zur Arbeit gehen, nach Hause kommen, essen, ausruhen und schlafen, wobei der Verzehr von Nachos irgendwo in den Bereich von essen und ausruhen gehören würde.

17.2 Klassifizierung und Mikrowellen

Wesentlich im Konzept der Abstraktion ist die Klassifizierung. Wenn ich meinen Sohn fragen würde »Was ist eine Mikrowelle?« würde er wahrscheinlich sagen »Es ist Ofen, der ...« Wenn ich dann fragen würde »Was ist ein Ofen?« könnte er antworten »Es ist ein Gerät in der Küche, das ...« Ich könnte weitere Fragen stellen, bis wir schließlich bei der Antwort ankommen »Es ist ein Ding«, was eine andere Form von »Es ist ein Objekt« ist.

Mein Sohn versteht, dass unsere spezielle Mikrowelle ein Beispiel ist für den Typ der Dinge, die als Mikrowellen bezeichnet werden. Er versteht auch, dass die Mikrowelle ein spezieller Typ Ofen ist, und dieser wiederum ein spezielles Küchengerät ist.

Technisch gesagt, ist meine Mikrowelle eine *Instanz* der Klasse *Mikrowelle*. Die Klasse *Mikrowelle* ist eine *Unterklasse* der Klasse *Ofen*, und die Klasse *Ofen* ist eine *Superklasse* der Klasse *Mikrowelle*.

Menschen klassifizieren. Alles in unserer Welt ist in Klassen eingeteilt. Wir tun das, um die Anzahl der Dinge, die wir uns merken müssen, klein zu halten. Erinnern Sie sich z.B. daran, wann Sie zum ersten Mal den Ford-basierten Jaguar (oder den neuen Neon) gesehen haben. Die Werbung nannte den Jaguar »revolutionär, eine neue Art Auto«. Aber Sie und ich wissen, dass das nicht stimmt. Ich mag das Aussehen des Jaguars – ich mag es sogar sehr – aber es ist nur ein Auto. Als solches teilt es alle (oder zumindest die meisten) Eigenschaften mit anderen Autos. Er hat ein Lenkrad, Sitze, einen Motor, Bremsen, usw. Ich wette, ich könnte sogar einen Jaguar ohne Hilfe fahren.

Ich will den wenigen Platz, den ich in diesem Buch zur Verfügung habe, nicht mit all den Dingen verschwenden, die ein Jaguar mit anderen Autos gleich hat. Ich muss nur an den Satz »ein Jaguar ist ein Auto, das ...« denken und die wenigen Dinge, die einzigartig für Jaguar sind. Autos sind eine Unterklasse von bereiften Fahrzeugen, von denen es auch andere gibt, wie z.B. LKW und Pickups. Vielleicht sind bereifte Fahrzeuge eine Unterklasse von Fahrzeugen, die auch Boote und Flugzeuge einschließt. Das lässt sich beliebig fortsetzen.

17.2.1 Warum solche Objekte bilden?

Es scheint leichter zu sein, eine Mikrowelle zu entwerfen und zu bauen, speziell für unser Problem als ein separates, allgemeineres Ofenobjekt. Stellen Sie sich z.B. vor, ich wollte eine Mikrowelle bauen, um Nachos, und nur Nachos, darin zuzubereiten. Diese Mikrowelle bräuchte dann keine Frontplatte, sondern nur einen Startknopf. Sie kochen Nachos immer gleich lang. Wir könnten uns

190 Samstagabend

den ganzen Unsinn z.B. zum Auftauen, sparen. Die Mikrowelle könnte winzig sein. Sie müsste nur einen kleinen Teller aufnehmen können. Nur sehr wenig Platz würde so für Nachos verschwendet.

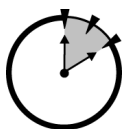
In diesem Sinne, lassen Sie uns das Konzept »Mikrowellenofen« vergessen, Alles, was wir wirklich brauchen, sind die Interna eines Ofens. In einem Rezept fassen wir dann alle Instruktionen zusammen, um ihn zum Laufen zu bringen: »Legen Sie die Nachos in die Box. Verbinden Sie das rote und das schwarze Kabel. Sie hören ein leichtes Summen«. So etwas in der Art.

Wie dem auch sei, der funktionale Ansatz hat einige Probleme:

- **Zu komplex.** Wir wollen die Details des Ofenbaus nicht vermischen mit den Details der Nachozubereitung. Wenn wir die Details nicht finden und aus der Fülle der Details herausziehen können, um sie separat zu bearbeiten, müssen wir immer mit der Gesamtkomplexität des Problems arbeiten.
- **Nicht flexibel.** Wenn wir den Mikrowellenofen durch einen anderen Typ Ofen ersetzen müssen, könnten wir das tun, solange der neue Ofen das gleiche Interface wie der alte Ofen hat. Ohne ein klar definiertes Interface wird es unmöglich, einen Objekttyp sauber zu entfernen und durch einen anderen zu ersetzen.
- **Nicht wiederverwendbar.** Öfen werden verwendet, um verschiedene Gerichte zuzubereiten. Wir müssen nicht jedes Mal einen neuen Ofen kreieren, wenn wir ein neues Rezept haben. Wenn ein Problem gelöst ist, wäre es schön, die Lösung auch in zukünftigen Programmen wiederverwenden zu können.



Es kostet mehr, ein generisches Objekt zu schreiben. Es wäre billiger, eine Mikrowelle nur für Nachos zu bauen. Wir könnten auf die teure Zeitzuhr und Knöpfe verzichten, die für Nachos nicht benötigt werden. Nachdem wir ein generisches Objekt in mehr als einer Anwendung verwendet haben, kommt der Vorteil einer etwas teureren generischen Klasse gegenüber der wiederholten Entwicklung billigerer Klassen für jede neue Applikation zum Tragen.



10 Min.

17.2.2 Selbstenthaltende Klassen

Lassen Sie uns reflektieren, was wir gelernt haben. Im objektorientierten Zugang der Programmierung

- identifiziert der Programmierer die Klassen, die benötigt werden, um ein Problem zu lösen. Ich wusste gleich, dass ich einen Ofen brauche, um Nachos zuzubereiten.
- Der Programmierer erzeugt selbstenthaltende Klassen, die den Anforderungen des Problems entsprechen, und er kümmert sich nicht um die Details der Gesamtanwendung.
- Der Programmierer schreibt die Anwendung unter Verwendung der gerade erzeugten Klassen, ohne darüber nachzudenken, wie sie intern funktionieren.

Ein integraler Bestandteil dieses Programmiermodells ist, dass eine Klasse für sich selber verantwortlich ist. Eine Klasse sollte jederzeit in einem definierten Zustand sein. Es sollte nicht möglich sein, das Programm zum Absturz zu bringen, indem man einer Klasse ungültige Daten oder eine ungültige Folge gültiger Daten übergibt.

Lektion 17 – Objektprogrammierung**191**

Paradigma ist ein anderes Wort für Programmiermodell.

Viele der Features von C++, die in den folgende Kapiteln beschrieben werden, behandeln die Fähigkeit von Klassen, sich selber gegen fehlerhafte Programme zu schützen, die nur auf einen Absturz warten.



Zusammenfassung

In dieser Sitzung haben Sie die fundamentalen Konzepte der objektorientierten Programmierung gesehen.

0 Min.

- Objektorientierte Programme bestehen aus locker verbundenen Objekten.
- Jede Klasse repräsentiert ein Konzept der realen Welt.
- Objektorientierte Klassen werden so geschrieben, dass sie in gewisser Hinsicht unabhängig sind von dem Programm, das sie benutzt.
- Objektorientierte Klassen sind verantwortlich für ihr eigenes Wohlbefinden.

Selbsttest

1. Was bedeutet der Begriff `Abstraktionslevel`? (Siehe »Abstraktion und Mikrowellen«)
2. Was ist mit `Klassifizierung` gemeint? (Siehe »Klassifizierung und Mikrowellen«)
3. Was sind die drei fundamentalen Probleme des funktionalen Programmiermodells, die vom objektorientierten Programmiermodell zu lösen versucht werden? (Siehe »Warum solche Objekte bilden?«)
4. Wie stellt man Nachos her? (Siehe »Siehe »Abstraktion und Mikrowellen«)

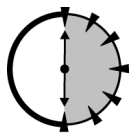
Teil 4 – Samstagabend
Lektion 17



Aktive Klassen

Checkliste

- ☒ Klassen durch Elementfunktionen in aktive Agenten verwandeln
- ☒ Elementfunktionen mit Namen versehen
- ☒ Elementfunktionen innerhalb und außerhalb der Klasse definieren
- ☒ Elementfunktionen aufrufen
- ☒ Klassendefinitionen in `#include`-Dateien sammeln
- ☒ Auf `this` zugreifen



30 Min.

Objekte in der realen Welt sind unabhängige Agenten (abgesehen von ihrer Abhängigkeit von Strom, Luft usw.). Eine Klasse sollte so unabhängig wie möglich sein. Es ist für eine struct-Struktur unmöglich, von ihrer Umgebung unabhängig zu sein. Die Funktionen, die ihre Daten manipulieren, müssen außerhalb der Struktur definiert sein. Aktive Klassen haben die Fähigkeit, diese Manipulationsfunktionen in sich selbst zu bündeln.

18.1 Klassenrückblick

Ein Klasse ermöglicht die Gruppierung von verwandten Datenelement in einer Einheit. Z.B. könnten wir eine Klasse `Student` wie folgt definieren:

```
class Student
{
public:
    int    nSemesterHours;    // Semesterstunden bis
                               // zur Graduierung
    float dAverage;          // mittlerer Grad
};
```

Jede Instanz von `Student` enthält ihre eigenen zwei Datenelemente:

```
void fn(void)
{
    Student s1;
    Student s2;
    s1.nSemesterHours = 1; // ist nicht gleich ...
    s2.nSemesterHours = 2; // ... mit diesem
}
```

Die beiden `nSemesterHours` sind verschieden, weil sie zu verschiedenen Studenten gehören (`s1` und `s2`).

Eine Klasse mit nichts als Datenelementen wird auch als *Struktur* bezeichnet und wird über das Schlüsselwort `struct` definiert, das von der Programmiersprache C herkommt. Eine `struct` ist identisch mit einer Klasse, außer dass das Schlüsselwort `public` nicht benötigt wird.

Es ist möglich, Zeiger auf Klassenobjekte zu definieren, und diese Objekte vom Heap zu allozieren, wie das folgende Beispiel zeigt:

```
void fn()
{
    Student* pS1 = new Student;
    pS1->nSemesterHours = 1;
}
```

Es ist möglich, Klassenobjekte an Funktionen zu übergeben:

```
void studentFunc(Student s);
void studentFunc(Student* pS);

void fn()
{
    Student s1 = {12, 4.0};
    Student* pS = new Student;

    studentFunc (s1); // ruft studentFunc(Student)
    studentFunc (ps); // ruft studentFunc(Student*)
}
```

18.2 Grenzen von `struct`

Klassen, die nur Datenelemente enthalten, haben entscheidende Grenzen.

Programme existieren in der realen Welt. D.h., jedes nicht triviale Programm ist dafür entworfen, eine Funktion der realen Welt bereitzustellen. Meistens, aber nicht immer, haben diese Funktionen eine Analogie, die manuell ausgeführt werden könnte. Z.B. könnte ein Programm den mittleren Grad eines Studenten ausrechnen. Das könnte per Hand mit Papier und Bleistift durchgeführt werden, es ist aber elektronisch viel einfacher und schneller.

Je genauer ein Programm das Leben widerspiegelt, um so einfacher ist das Programm zu verstehen. Wenn es einen Typ Ding gibt, das »Student« genannt wird, dann wäre es schön, eine Klasse `Student` zu haben, die alle wesentlichen Eigenschaften von Studenten besitzt. Eine Instanz der Klasse `Student` würde dann einem Studenten entsprechen.

Eine kleine Klasse `Student`, die alle Informationen beschreibt, die zur Berechnung des mittleren Grades eines Studenten benötigt werden, finden Sie am Anfang dieser Sitzung. Das Problem mit

194 Samstagabend

dieser Klasse ist, dass sie nur passive Eigenschaften von Studenten enthält. D.h., ein Student hat eine Eigenschaft für die Anzahl der Semester und seinen mittleren Grad. (Ein Student hat auch einen Namen, eine Sozialversicherungsnummer usw., aber es ist in Ordnung, diese Eigenschaften wegzulassen, wenn sie nicht zu dem Problem gehören, das wir lösen möchten.) Studenten beginnen Vorlesungen, brechen Vorlesungen ab und beenden Vorlesungen erfolgreich. Das sind aktive Eigenschaften der Klasse.

18.2.1 Eine funktionale Lösung

Sicher, es ist möglich, einer Klasse aktive Eigenschaften zuzufügen, indem man eine Menge von Funktionen für die Klasse schreibt:

```
// Definiere eine Klasse, um die passiven
// Eigenschaften eines Studenten zu speichern
class Student
{
public:
    int    nSemesterHours; // Semesterstunden bis
                        // zur Graduierung
    float  dAverage;      // mittlerer Grad
};

// eine Kursklasse
class Course
{
public:
    char*  pszName;
    int    nCourseNumber;
    int    nNumHours;
};

// definiere eine Menge von Funktionen, um die
// aktiven Eigenschaften eines Studenten zu
// beschreiben
void startCourse(Student* pS, Course* pC);
void dropCourse(Student* pS, int nCourseNumber);
void completeCourse(Student* pS, int nCourseNumber);
```

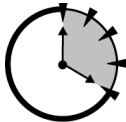
Diese Lösung funktioniert – in der Tat ist das die Lösung in nicht objektorientierten Programmiersprachen wie C. Es gibt jedoch ein Problem mit dieser Lösung.

In der Art und Weise, in der dieser Auszug geschrieben ist, hat Student nur passive Eigenschaften. Es gibt da ein nebulöses »Ding«, das aktive Agenten hat, wie `startCourse( )`, das auf Objekten aus der Klasse Student arbeitet (und Objekten aus der Klasse Course). Dieses nebulöse Ding hat keine eigenen Dateneigenschaften. Obwohl sie funktioniert, spiegelt diese Beschreibung nicht die Realität wider.

Wir würden gerne die aktiven Eigenschaften aus diesem undefinierten Ding herausnehmen und sie der Klasse Student selber hinzufügen, so dass jede Instanz der Klasse Student mit einem kompletten Satz von Eigenschaften ausgerüstet ist.



Das Hinzufügen aktiver Eigenschaften zu einer Klasse, statt diese aus der Klasse herauszulassen, scheint Ihnen vielleicht nebensächlich zu sein, es wird aber im weiteren Verlauf dieses Buches immer wichtiger.



18.3 Definition einer aktiven Klasse

Die aktiven Eigenschaften eines Studenten könnten wie folgt der Klasse `Student` zugefügt werden:

20 Min.

```
class Student
{
public:
    // die passiven Eigenschaften der Klasse
    int  nSemesterHours; // Semesterstunden bis
                          // zur Graduierung
    float dAverage;      // mittlerer Grad

    // die aktiven Eigenschaften der Klasse
    float startCourse(Course*);
    void  dropCourse(int nCourseNumber);
    void  completeCourse(int nCourseNumber);
};
```

Die Funktion `startCourse(Course*)` ist eine Eigenschaft der Klasse, wie `nSemesterHours` und `dAverage`.



Eine Funktion, die Element einer Klasse ist, wird Elementfunktion genannt. Aus historischen Gründen, die nur sehr wenig mit C++ zu tun haben, werden Elementfunktionen auch Methoden genannt. Wahrscheinlich weil der Begriff »Methode« der verwirrendere der beiden Begriffe ist, wird er am häufigsten verwendet.



Es gibt keinen Namen für Funktionen oder Daten, die nicht Element einer Klasse sind. Ich bezeichne sie als Nichtelemente. Alle Funktionen, die sie bisher gesehen haben, sind Nichtelemente gewesen, weil sie nicht zu einer Klasse gehören.



C++ kümmert sich nicht um die Ordnung der Elementfunktionen innerhalb einer Klasse. Die Datenelemente können vor oder nach den Elementfunktionen stehen, sie können auch miteinander vermischt werden. Ich selber bevorzuge es, die Datenelemente vor die Elementfunktionen zu platzieren.

196 Samstagabend**18.3.1 Namengebung für Elementfunktionen**

Der vollständige Name der Funktion `startCourse(Course*)` ist `Student::startCourse(Course*)`. Der Klassenname am Anfang zeigt an, dass die Funktion ein Element der Klasse `Student` ist. (Der Klassenname wird dem erweiterten Namen der Funktion zugefügt, wie die Argumente zum Namen einer überladenen Funktion hinzugefügt wurden.) Wir könnten weitere Funktionen mit Namen `startCourse( )` haben, die Elemente in anderen Klassen wären, wie z.B. `Teacher::startCourse( )`. Eine Funktion `startCourse( )` ohne einen Klassennamen am Anfang, ist eine herkömmliche Nicht-elementfunktion.



Tatsächlich ist der vollständige Name der Nichtelementfunktion `addCourse( )` gleich `::addCourse( )`. Die beiden Doppelpunkte `::` ohne Klassennamen auf ihrer linken Seite weisen ausdrücklich darauf hin, dass es sich um eine Nichtelementfunktion handelt.

Datenelemente verhalten sich nicht anders als Elementfunktionen in Bezug auf erweiterte Namen. Außerhalb einer Struktur ist es nicht ausreichend, nur `nSemesterHours` zu verwenden. Das Datenelement `nSemesterHours` macht nur im Kontext der Klasse `Student` Sinn. Der erweiterte Name von `nSemesterHours` ist `Student::nSemesterHours`.

Der Operator `::` wird auch *Bereichsauflösungsoperator* genannt, weil er die Klasse identifiziert, zu der ein Element gehört. Der Operator `::` kann mit einer Nichtelementfunktion und mit einem leeren Strukturnamen aufgerufen werden. Die Nichtelementfunktion `startCourse( )` sollte als `::startCourse( )` angesprochen werden.

Der Operator ist optional, außer wenn es zwei Funktionen mit gleichem Namen gibt. Hier ein Beispiel dazu:

```
float startCourse(Course*);

class Student
{
public:
    int    nSemesterHours; // Semesterstunden bis zur
                        // Graduierung
    float dAverage;       // mittlerer Grad

    // beginne neuen Kurs
    float startCourse(Course* pCourse)
    {
        // ... was auch immer ...

        startCourse(pCourse); // globale Funktion(?)

        // ... weitere Anweisungen ...
    }
};
```

Wir wollen, dass die Elementfunktion `Student::startCourse( )` die Nichtelementfunktion `::startCourse( )` aufruft. Ohne den Operator `::` bezieht sich ein Aufruf von `startCourse( )` jedoch auf `Student::startCourse( )`. Das führt dazu, dass sich die Funktion selber aufruft. Das

Hinzufügen des Operators `::` an den Anfang des Aufrufs leitet den Aufruf wie gewünscht an die globale Funktion weiter:

```
class Student
{
public:
    int    nSemesterHours; // Semesterstunden bis zur
                        // Graduierung
    float dAverage;        // mittlerer Grad

    // starte neuen Kurs
    float startCourse(Course* pCourse)
    {
        ::startCourse(pCourse); // globale Funktion
    }
};
```

Der vollständig erweiterte Name einer Nichtelementfunktion enthält also nicht nur die Argumente, wie wir in Sitzung 9 gesehen haben, sondern auch den Namen der Klasse, zu der die Funktion gehört – der ist leer für Nichtelementfunktionen.

18.4 Definition einer Elementfunktion einer Klasse

Eine Elementfunktion kann entweder in der Klasse oder separat definiert werden. Betrachten Sie die folgende Definition der Methode `addCourse(int, float)` innerhalb der Klasse:

```
class Student
{
public:
    int    nSemesterHours; // Semesterstunden bis zur
                        // Graduierung
    float dAverage;        // mittlerer Grad

    // füge einen absolvierten Kurs ein
    float addCourse(int hours, float grade)
    {
        float weighted;
        weighted = nSemesterHours * dAverage;

        // füge den neuen Kurs ein
        nSemesterHours += hours;
        weighted += grade * hours;
        dAverage = weighted / nSemesterHours;
        return dAverage;
    }
};
```

Der Code von `addCourse(int, float)` sieht nicht anders aus als der jeder anderen Funktion, außer dass er eingebettet ist in die Klasse.

Elementfunktionen, die innerhalb der Klasse definiert sind, werden standardmäßig als Inline-Funktionen behandelt (s.u.). Hauptsächlich ist dies der Fall, weil Elementfunktionen, die in der Klasse definiert sind, sehr kurz sind, und kurze Funktionen die primären Kandidaten für eine Behandlung als Inline-Funktion sind.

198 Samstagabend**Inline-Funktionen**

Normalerweise veranlasst eine Funktionsdefinition den C++-Compiler, Maschinencode an einer bestimmten Stelle des ausführbaren Programms zu platzieren. Jedes Mal, wenn die Funktion aufgerufen wird, fügt C++ eine Art Sprung an die Stelle ein, an der die Funktion gespeichert ist. Wenn die Funktion durchlaufen wurde, geht die Kontrolle zurück zu dem Punkt, von wo aus die Funktion aufgerufen wurde.

C++ definiert einen speziellen Typ Funktion, der als *Inline*-Funktion bezeichnet wird. Wenn eine Inline-Funktion aufgerufen wird, generiert der C++-Compiler den Code direkt an dieser Stelle. Jeder Aufruf der Inline-Funktion erhält seine eigene Kopie des Maschinencodes.

Inline-Funktionen werden schneller ausgeführt, weil der Computer nicht an eine andere Stelle springen und Initialisierungen ausführen muss, um die Verarbeitung fortzuführen. Denken Sie jedoch daran, dass Inline-Funktionen mehr Speicher benötigen. Wenn eine Inline-Funktion zehnmal aufgerufen wird, befinden sich 10 Kopien des Maschinencodes im Speicher.

Weil der Unterschied zwischen Inline-Funktionen und konventionellen Funktionen, die manchmal auch als Outline-Funktionen bezeichnet werden, in der Ausführungsgeschwindigkeit klein ist, sind nur kleine Funktionen Kandidaten für eine Inline-Behandlung. Zusätzlich zwingen bestimmte Konstruktionen eine Inline-Funktion zu einer Behandlung als Outline-Funktion.

18.5 Schreiben von Elementfunktionen außerhalb einer Klasse

Für längere Funktionen führt das direkte Einfügen des Funktionscodes zu sehr langen und schwerfälligen Klassendefinitionen. Um dies zu verhindern, erlaubt es uns C++, Elementfunktionen außerhalb der Klasse zu definieren.

Unsere Methode `addCourse()`, die außerhalb der Klasse definiert ist, sieht dann so aus:

```
class Student
{
public:
    int    nSemesterHours; // Semesterstunden bis zur
                        // Graduierung
    float dAverage;        // mittlerer Grad

    // füge einen absolvierten Kurs ein
    float addCourse(int hours, float grade);
};
float Student::addCourse(int hours, float grade)
{
    float weighted;
    weighted = nSemesterHours * dAverage;

    // füge den neuen Kurs ein
    nSemesterHours += hours;
    weighted += grade * hours;
    dAverage = weighted / nSemesterHours;
    return dAverage;
}
```

Lektion 18 – Aktive Klassen 199

Hier sehen wir, dass die Klassendefinition nichts weiter als eine *Prototypdeklaration* der Funktion `addCourse()` enthält. Die tatsächliche *Funktionsdefinition* steht separat.



Eine Deklaration definiert den Typ einer Sache. Eine Definition definiert den Inhalt einer Sache.

Die Analogie mit einer Prototypdeklaration ist exakt. Die Deklaration in der Struktur ist eine Prototypdeklaration und ist, wie alle Prototypdeklarationen, notwendig.

Als die Funktion innerhalb der Klasse `Student` definiert wurde, war es nicht nötig, dass der Klassenname im Namen der Funktion enthalten ist; es wurde der Name der enthaltenden Klasse angenommen. Wenn die Funktion alleine steht, wird der Klassenname benötigt. Es ist wie bei mir zu Hause. Meine Frau ruft mich nur bei meinem Vornamen (vorausgesetzt ich bin nicht in der Hundehütte). Innerhalb der Familie wird der Nachname angenommen. Außerhalb der Familie (und meinem Bekanntenkreis) rufen mich andere mit meinem vollen Namen.

18.5.1 Include-Dateien

Es ist üblich, Klassendefinitionen und Funktionsprototypen in eine Datei zu schreiben, die die Endung `.h` trägt, getrennt von der `.cpp`-Datei, die die tatsächlichen Funktionsdefinitionen enthält. Die `.h`-Datei wird dann in der `.cpp`-Datei »included« (=eingebunden), wie im Folgenden zu sehen ist.

Die Datei `student.h` wird am besten so definiert:

```
class Student
{
public:
    int    nSemesterHours; // Semesterstunden bis zur
                        // Graduierung
    float dAverage;        // mittlerer Grad

    // füge einen absolvierten Kurs ein
    float addCourse(int hours, float grade);
};
```

Die Datei `student.cpp` sieht wie folgt aus:

```
#include »student.h«;
float Student::addCourse(int hours, float grade)
{
    float weighted;
    weighted = nSemesterHours * dAverage;

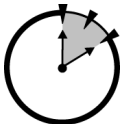
    // füge den neuen Kurs ein
    nSemesterHours += hours;
    weighted += grade * hours;
    dAverage = weighted / nSemesterHours;
    return dAverage;
}
```

Die Direktive `#include` besagt »ersetze diese Direktive durch den Inhalt der Datei `student.h`«.



Die Direktive `#include` hat nicht das Format einer C++-Anweisung, weil es von einem separaten Interpreter verarbeitet wird, der vor dem C++-Compiler ausgeführt wird.

Klassendefinitionen und Funktionsprototypen in Include-Dateien zu packen, ermöglichen es mehreren C++-Modulen, die gleichen Informationen einzubinden, ohne sie wiederholen zu müssen. Das reduziert den Aufwand und, was noch wichtiger ist, es reduziert die Wahrscheinlichkeit, dass mehrere Quelldateien nicht mehr synchron sind.



18.6 Aufruf einer Elementfunktion

Bevor wir uns ansehen, wie Elementfunktionen aufgerufen werden, lassen Sie uns daran erinnern, wie ein Datenelement referenziert wird:

10 Min.

```
#include >student.h<
Student s;
void fn(void)
{
    // Zugriff auf Datenelemente von s
    s.nSemesterHours = 10;
    s.dAverage        = 3.0;
}
```

Wir müssen ein Objekt zusammen mit dem Elementnamen spezifizieren, wenn wir ein Objektelement ansprechen wollen. In anderen Worten, das Folgende macht keinen Sinn:

```
#include >student.h<
void fn(void)
{
    Student s;

    // Zugriff auf die Datenelemente von s
    // keiner der Zugriffe ist zulässig
    nSemesterHours = 10; // Element welches Objekt
                        // aus welcher Klasse?
    Student::nSemesterHours = 10;
                        // ok, ich weiß die Klasse,
                        // aber ich weiß nicht
                        // welches Objekt
    s.nSemesterHours = 10; // das ist in Ordnung
}
```

Lektion 18 – Aktive Klassen 201

Elementfunktionen werden mit einem Objekt aufgerufen, wie Datenelemente angesprochen werden:

```
void fn()
{
    Student s;

    // referenziere die Datenelemente der Klasse
    s.nSemesterHours = 10;
    s.dAverage       = 3.0;

    // Zugriff auf die Elementfunktion
    s.addCourse(3, 4.0);
}
```

Eine Elementfunktion ohne ein Objekt aufzurufen, macht nicht mehr Sinn, als ein Datenelement ohne ein Objekt anzusprechen. Die Syntax für den Aufruf einer Elementfunktion sieht aus wie eine Kreuzung der Syntax für Zugriffe auf Datenelemente und dem Aufruf konventioneller Funktionen.

18.6.1 Aufruf einer Elementfunktion über einen Zeiger

Die gleiche Parallele wie für die Objekte selber kann auch für Zeiger auf Objekte gezogen werden. Das Folgende referenziert ein Datenelement eines Objektes über einen Zeiger:

```
#include >>student.h<<

void someFn(Student *pS)
{
    // Zugriff auf die Datenelemente der Klasse
    pS->nSemesterHours = 10;
    pS->dAverage       = 3.0;

    // Zugriff auf die Elementfunktion
    // (d.h. Aufruf dieser Funktion)
    pS->addCourse(3, 4.0);
}

int main()
{
    Student s;

    someFn(&s);
    return 0;
}
```

Eine Elementfunktion mit einer Referenz auf ein Objekt aufzurufen, erscheint identisch mit der Benutzung des Objektes selber. Erinnern Sie sich daran, dass bei der Übergabe oder Rückgabe einer Referenz als Argument einer Funktion C++ nur die Adresse des Objektes übergibt. Bei Verwendung einer Referenz dereferenziert C++ die Adresse automatisch, wie das folgende Beispiel zeigt:

202 Samstagabend

```
#include >student.h<<

// das Gleiche wie eben, nur dieses
// Mal mit Referenzen
void someFn(Student &refS)
{
    refS.nSemesterHours = 10;
    refS.dAverage       = 3.0;
    refS.addCourse(3, 4.0); // Aufruf Elementfunktion
}

Student s;
int main()
{
    someFn(s);
    return 0;
}
```

18.6.2 Zugriff auf andere Elemente von einer Elementfunktion aus

Es ist klar, warum Sie keine Elemente einer Klasse ohne ein Objekt ansprechen können. Sie müssen z.B. wissen, welches `dAverage` von welchem `Student`-Objekt? Schauen Sie sich nochmals die Definition der Elementfunktion `Student::addCourse( )` an. Diese Funktion greift auf Klassenelemente zu, ohne explizit ein Objekt zu referenzieren, was in direktem Widerspruch dazu steht, was ich gerade gesagt habe.

Sie können ein Element einer Klasse nicht ohne ein Objekt ansprechen; innerhalb einer Elementfunktion jedoch wird das Objekt als das referenzierte Objekt angenommen. Es ist leichter, das an einem Beispiel zu sehen:

```
#include >student.h<<
float Student::addCourse(int hours, float grade)
{
    float weighted;
    weighted = nSemesterHours * dAverage;

    // füge den neuen Kurs ein
    nSemesterHours += hours;
    weighted += hours * grade;
    dAverage = weighted / nSemesterHours;
    return dAverage;
}

int main(int nArgs, char* pArgs[])
{
    Student s;
    Student t;

    s.addCourse(3, 4.0); // Note 4
    t.addCourse(3, 2.5); // Note 2-
    return 0;
}
```

Lektion 18 – Aktive Klassen 203

Wenn `addCourse()` mit dem Objekt `s` aufgerufen wird, beziehen sich alle anderen unqualifizierten Elemente innerhalb von `addCourse()` auf `s`. Somit wird `nSemesterHours` zu `s.nSemesterHours`, `dAverage` wird zu `s.dAverage`. Im Aufruf `t.addCourse()` in der nächsten Zeile beziehen sich die gleichen Referenzen auf `t.nSemesterHours` und `t.dAverage`.

Das Objekt, mit dem die Elementfunktion aufgerufen wird, ist das »aktuelle« Objekt, und alle unqualifizierten Referenzen auf Klassenelemente beziehen sich auf dieses Objekt. Oder anders gesagt, beziehen sich unqualifizierte Referenzen auf Klassenelemente auf das aktuelle Objekt.

Wie wissen Elementfunktionen, was das aktuelle Objekt ist? Es ist keine Magie – die Adresse des Objektes wird an die Elementfunktion übergeben als implizites und nicht sichtbares erstes Argument. In anderen Worten findet die folgende Konvertierung statt:

```
s.addCourse(3, 2.5)
```

wird zu

```
Student::addCourse(&s, 3, 2.5);
```

(Sie können diese interpretative Syntax so nicht verwenden; sie ist nur ein Weg, um zu verstehen, was C++ tut.)

Innerhalb der Funktion hat dieser implizite Zeiger auf das aktuelle Objekt einen Namen, für den Fall, dass Sie darauf zugreifen müssen. Dieser versteckte Objektzeiger heißt `this` (=dieses). Der Typ von `this` ist immer ein Zeiger auf ein Objekt der entsprechenden Klasse. Innerhalb der Klasse `Student` ist `this` vom Typ `Student*`.

Jedes Mal, wenn eine Elementfunktion auf ein anderes Element der gleichen Klasse verweist, ohne explizit ein Objekt zu benennen, nimmt C++ an, dass `this` gemeint ist. Sie können `this` auch explizit verwenden. Wir hätten `Student::addCourse()` auch so schreiben können:

```
#include >>student.h<<
float Student::addCourse(int hours, float grade)
{
    float weighted;

    // referenziere 'this' explicit
    weighted = this->nSemesterHours * this->dAverage;

    // gleiche Rechnung mit 'this' implizit
    weighted = nSemesterHours * dAverage;

    // füge den neuen Kurs ein
    this->nSemesterHours += hours;
    weighted += hours * grade;
    this->dAverage = weighted / this->nSemesterHours;
    return this->dAverage;
}
```

Ob wir `this` explizit hinschreiben oder es implizit lassen, wie wir es bereits getan haben, hat den gleichen Effekt.

204 Samstagabend**18.7 Überladen von Elementfunktionen**

Elementfunktionen können überladen werden in der gleichen Weise wie konventionelle Funktionen. Erinnern Sie sich jedoch daran, dass der Klassenname Teil des erweiterten Namens ist. Somit sind die folgenden Funktionen legal:

```
class Student
{
public:
    // grade - aktueller mittlerer Grad
    float grade();

    // grade - setze Grad und gib vorigen Wert zurück
    float grade(float newGrade);

    // ... Datenelemente und alles weitere ...
};

class Slope
{
public:
    // grade - Grad des Gefälles
    float grade();

    // ... hier steht der Rest ...
};

// grade - gibt Zeichenäquivalent eines Wertes zurück
char grade(float value);

int main(int nArgs, char* pArgs[])
{
    Student s;
    Slope o;

    // rufe verschiedene Varianten von grade() auf
    s.grade(3.5);           // Student::grade(float)
    float v = s.grade();    // Student::grade()
    char c = grade(v);      // ::grade(float)
    float m = o.grade();    // Slope::grade()
    return 0;
}
```

Jeder Aufruf von `main( )` aus ist im Kommentar mit dem erweiterten Namen der aufgerufenen Funktion versehen.

Wenn überladene Funktionen aufgerufen werden, werden sowohl die Argumente der Funktion, und der Typ des Objektes (falls vorhanden), mit dem die Funktion aufgerufen wird, verwendet, um den Aufruf unzweideutig zu machen.

Der Begriff *unzweideutig* ist ein objektorientierter Begriff, der sagen soll »entscheide zur Compilezeit, welche überladene Funktion aufgerufen werden soll«. Wir können auch sagen, dass die Aufrufe aufgelöst werden.



0 Min.

Im Beispielcode unterscheiden sich die beiden ersten Aufrufe der Elementfunktionen `Student::grade(float)` und `Student::grade( )` durch ihre Argumentlisten. Der dritte Aufruf hat kein Objekt, somit bezeichnet es unzweideutig die Nichtelementfunktion `grade(float)`. Weil der letzte Aufruf mit einem Objekt vom Typ `Slope` durchgeführt wird, muss er sich auf die Elementfunktion `Slope::grade( )` beziehen.

Zusammenfassung

Je näher Sie das Problem, das Sie mit C++ lösen wollen, modellieren können, desto leichter kann das Problem gelöst werden. Diejenigen Klassen, die nur Datenelemente enthalten, können nur passive Eigenschaften von Objekten modellieren. Das Hinzufügen von Elementfunktionen macht die Klasse mehr zu einem Objekt wie in der realen Welt, in dem Sinne, dass es nun auf die Welt »außen«, d.h. den Rest des Programms, reagieren kann. Außerdem kann die Klasse für ihr eigenes Wohlergehen verantwortlich gemacht werden, in der gleichen Weise, wie Objekte in der realen Welt sich selbst schützen.

- Elemente einer Klasse können Funktionen oder Daten sein. Solche Elementfunktionen machen eine Klasse aktiv. Der vollständige Name einer Elementfunktion schließt den Namen der Klasse ein.
- Elementfunktionen können entweder innerhalb oder außerhalb der Klasse definiert werden. Elementfunktionen, die außerhalb einer Klasse definiert sind, sind der Klasse schwerer zuzuordnen, tragen aber zu einer besseren Lesbarkeit der Klasse bei.
- Innerhalb einer Elementfunktion kann das aktuelle Objekt über das Schlüsselwort `this` referenziert werden.

Selbsttest

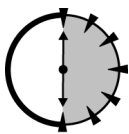
1. Was ist falsch daran, Funktionen außerhalb einer Klasse zu deklarieren, die Datenelemente der Klasse direkt manipulieren? (Siehe »Eine funktionale Lösung«)
2. Eine Funktion, die ein Element einer Klasse ist, wird wie bezeichnet? Es gibt zwei Antworten auf diese Frage. (Siehe »Definition einer aktiven Klasse«)
3. Beschreiben Sie die Bedeutung der Reihenfolge von Funktionen innerhalb einer Klasse. (Siehe »Definition einer aktiven Klasse«)
4. Wenn eine Klasse `X` ein Element `Y(int)` besitzt, was ist der vollständige Name der Funktion? (Siehe »Schreiben von Elementfunktionen außerhalb einer Klasse«)
5. Warum wird eine Include-Datei so genannt? (Siehe »Include-Dateien«)



Erhalten der Klassenintegrität

Checkliste

- ☒ Einen Konstruktor schreiben und benutzen
- ☒ Datenelemente konstruieren
- ☒ Einen Destruktor schreiben und benutzen
- ☒ Zugriff auf Datenelemente kontrollieren



30 Min.

Ein Objekt kann nicht für sein Wohlergehen verantwortlich gemacht werden, wenn es keine Kontrolle darüber hat, wie es erzeugt und verwendet wird. In dieser Sitzung untersuchen wir die Möglichkeiten in C++, die Integrität von Objekten zu erhalten.

19.1 Erzeugen und Vernichten von Objekten

C++ kann ein Objekt als Teil einer Deklaration initialisieren, z.B.:

```
class Student
{
    public:
        int    nSemesterHours;
        float dAverage;
};

void fn()
{
    Student s = {0, 0};
    //... Fortsetzung der Funktion ...
}
```

Hierbei hat `fn()` totale Kontrolle über das `Student`-Objekt.

Lektion 19 – Erhalten der Klassenintegrität 207

Wir könnten die Klasse mit einer Initialisierungsfunktion ausstatten, die von der Anwendung aufgerufen wird, sobald ein Objekt erzeugt wird. Das gibt der Klasse die Kontrolle darüber, wie ihre Datenelemente initialisiert werden. Die Lösung sieht dann so aus:

```
class Student
{
public:
    // data members
    int  nSemesterHours;
    float dAverage;

    // Elementfunktionen
    // init - initialisiere ein Objekt mit einem
    // gültigen Zustand
    void init()
    {
        semesterHours = 0;
        dAverage = 0.0;
    }
};

void fn()
{
    // erzeuge gültiges Student-Objekt
    Student s; // erzeuge das Objekt...
    s.init();  // ... und initialisiere es

    //... Fortsetzung der Funktion ...
}
```

Das Problem mit der »init«-Lösung ist, dass sich die Klasse auf die Anwendung verlassen muss, dass die Funktion `init()` aufgerufen wird. Das ist nicht die angestrebte Lösung. Was wir eigentlich wollen, ist ein Mechanismus, der ein Objekt automatisch initialisiert, wenn es erzeugt wird.

19.1.1 Der Konstruktor

C++ ermöglicht es einer Klasse, ihre Objekte zu initialisieren, indem eine spezielle Funktion bereitgestellt wird. Diese wird als Konstruktor bezeichnet.



Ein Konstruktor ist eine Elementfunktion, die automatisch aufgerufen wird, wenn ein Objekt erzeugt wird. In gleicher Weise wird ein Destruktor aufgerufen, wenn ein Objekt vernichtet wird.

C++ führt einen Aufruf eines Konstruktors immer aus, wenn ein Objekt erzeugt wird. Der Konstruktor trägt den gleichen Namen wie die Klasse. Auf diese Weise weiß der Compiler, welche Elementfunktion ein Konstruktor ist.



Die Entwickler von C++ hätten auch eine andere Regel aufstellen können, z.B. »Der Konstruktor muss `init( )` heißen.« Die Programmiersprache Java verwendet eine solche Regel. Eine andere Regel würde keinen Unterschied machen, solange wie der Compiler den Konstruktor von anderen Elementfunktionen unterscheiden kann.

Mit einem Konstruktor sieht die Klasse `Student` wie folgt aus:

```
class Student
{
    public:
        // Datenelemente
        int    nSemesterHours;
        float  dAverage;

        // Elementfunktionen
        Student()
        {
            nSemesterHours = 0;
            dAverage = 0.0;
        }
};

void fn()
{
    Student s; // erzeuge und initialisiere Objekt
    // ... Fortsetzung der Funktion ...
}
```

An der Stelle, an der `s` deklariert wird, fügt der Compiler einen Aufruf des Konstruktors `Student::Student( )` ein.

Dieser einfache Konstruktor wurde als Inline-Elementfunktion geschrieben. Konstruktoren können auch als Outline-Funktionen geschrieben werden, z.B.:

```
class Student
{
    public:
        // Datenelemente
        int    nSemesterHours;
        float  dAverage;

        // Elementfunktionen
        Student();
};

Student::Student()
{
    nSemesterHours = 0;
    dAverage = 0.0;
}

int main(int nArgc, char* pszArgs)
{
```

Lektion 19 – Erhalten der Klassenintegrität 209

```
Student s; // erzeuge und initialisiere Objekt
return 0;
}
```

Ich habe eine kleine Funktion `main( )` hinzugefügt, damit Sie das Programm ausführen können. Sie sollten mit Ihrem Debugger in Einzelschritten durch das Programm gehen, bevor sie hier fortfahren.

Wenn Sie in Einzelschritten durch dieses Beispielprogramm geben, kommt die Kontrolle schließlich zur Deklaration `Student s;`. Führen Sie Step In einmal aus, und die Kontrolle springt magischerweise nach `Student::Student( )`. Gehen Sie in Einzelschritten durch den Konstruktor. Wenn die Funktion fertig ist, kehrt die Kontrolle zur Anweisung nach der Deklaration zurück.

Mehrere Objekte können in einer Zeile deklariert werden. Gehen Sie nochmals in Einzelschritten durch die Funktion `main( )`, die wie folgt deklariert ist:

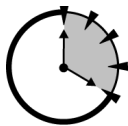
```
int main(int nArgc, char* pszArgs)
{
    Student s[5];    // erzeuge Array von Objekten
}
```

Der Konstruktor wird fünfmal aufgerufen, einmal für jedes Element des Array.



Tipp

Wenn Sie den Debugger nicht zum Laufen bringen (oder sich damit einfach nicht beschäftigen möchten), fügen Sie eine Ausgabeanweisung in den Konstruktor ein, damit Sie auf dem Bildschirm sehen können, wenn der Konstruktor aufgerufen wird. Der Effekt ist nicht so dramatisch, aber überzeugend.



Grenzen des Konstruktors

Der Konstruktor kann nur automatisch aufgerufen werden. Er kann nicht wie eine normale Elementfunktion aufgerufen werden. D.h. Sie können ihn nicht so wie in folgendem Beispiel verwenden, um ein `Student`-Objekt neu zu initialisieren:

20 Min.

```
void fn()
{
    Student s; // erzeuge und initialisiere Objekt
    // ... weitere Anweisungen ...
    s.Student(); // initialisiere Objekt erneut -
                // das funktioniert nicht
}
```

Der Konstruktor hat keinen Rückgabewert, noch nicht einmal `void`. Die Konstruktoren, die Sie hier sehen, haben alle eine `void`-Argumentliste.



Hinweis

Der Konstruktor ohne Argumente wird auch als Defaultkonstruktor bezeichnet.

210 Samstagabend

Der Konstruktor kann andere Funktionen aufrufen. Wenn wir ein Objekt neu initialisieren wollen, schreiben wir das Folgende:

```
class Student
{
public:
    // Datenelemente
    int    nSemesterHours;
    float dAverage;

    // Elementfunktionen
    // Konstruktor - initialisiert das Objekt bei
    // Erzeugung automatisch
    Student()
    {
        init();
    }

    // init - initialisiere das Objekt
    void init()
    {
        nSemesterHours = 0;
        dAverage = 0.0;
    }
};

void fn()
{
    Student s; // erzeuge und initialisiere Objekt

    // ... weitere Anweisungen ...
    s.init(); // initialisiere Objekt erneut
}
```

Hierbei ruft der Konstruktor eine allgemein verfügbare Funktion `init( )` auf, die die Initialisierung durchführt.

Konstruktion von Datenelementen

Die Datenelemente eines Objektes werden zur gleichen Zeit wie das Objekt erzeugt. Die Datenelemente werden tatsächlich in der Reihenfolge erzeugt, in der sie im Quelltext stehen, und vor dem Rest der Klasse. Betrachten Sie das Programm `ConstructMember` in Listing 19-1. Ausgabeanweisungen wurden in die Konstruktoren der einzelnen Klassen eingefügt, damit Sie sehen können, in welcher Reihenfolge die Objekte erzeugt werden.

Listing 19-1: Das Programm ConstructMembers

```
// ConstructMembers - erzeuge ein Objekt, das
//                     Datenelemente hat, die selber
//                     Objekte einer Klasse sind
#include <stdio.h>
#include <iostream.h>
class Student
{
public:
    Student()
    {
        cout << »Konstruktor Student\n«;
    }
};

class Teacher
{
public:
    Teacher()
    {
        cout << »Konstruktor Teacher\n«;
    }
};

class TutorPair
{
public:
    Student student;
    Teacher teacher;
    int      noMeetings;

    TutorPair()
    {
        cout << »Konstruktor TutorPair \n«;
        noMeetings = 0;
    }
};

int main(int nArgc, char* pArgs[])
{
    cout << »Erzeuge ein TutorPair\n«;

    TutorPair tp;

    cout << »Zurück in main\n«;
    return 0;
}
```

212 Samstagabend

Ausführen des Programms erzeugt die folgende Ausgabe:

```
Erzeuge ein TutorPair
Konstruktor Student
Konstruktor Teacher
Konstruktor TutorPair
Zurück in main
```

Bei der Erzeugung von `tp` in `main()` wird der Konstruktor von `TutorPair` automatisch aufgerufen. Bevor die Kontrolle an den Body des Konstruktors `TutorPair` übergeht, werden die Konstruktoren der beiden Elementobjekte – `student` und `teacher` – aufgerufen.

Der Konstruktor von `Student` wird zuerst aufgerufen, weil das Element `student` zuerst deklariert ist. Dann wird der Konstruktor von `Teacher` aufgerufen. Nachdem die Objekte erzeugt sind, geht die Kontrolle auf die öffnende Klammer über, und der Konstruktor von `TutorPair` darf den Rest des Objektes initialisieren.



Es würde nicht gehen, `TutorPair` für die Initialisierung von `student` und `teacher` verantwortlich zu machen. Jede Klasse ist für die Initialisierung ihrer Objekte selber zuständig.

19.1.2 Der Destruktor

So wie Objekte erzeugt werden, werden sie auch wieder vernichtet. Wenn eine Klasse einen Konstruktor zur Initialisierung hat, sollte sie auch eine spezielle Elementfunktion besitzen, die als Destruktor bezeichnet wird, um ein Objekt zu vernichten.

Der Destruktor ist eine spezielle Elementfunktion, die aufgerufen wird, wenn ein Objekt vernichtet wird, oder in C++-Sprechweise »destruiert« wird.

Eine Klasse kann in einem Konstruktor Ressourcen anlegen; diese Ressourcen müssen wieder freigegeben werden, bevor das Objekt aufhört zu existieren. Wenn der Konstruktor z.B. eine Datei öffnet, muss diese Datei wieder geschlossen werden. Oder wenn der Konstruktor Speicher vom Heap alloziert, muss dieser Speicher wieder freigegeben werden, bevor das Objekt verschwindet. Der Destruktor erlaubt es der Klasse, dieses Aufräumen automatisch durchzuführen, ohne sich auf die Anwendung verlassen zu müssen, dass die entsprechende Elementfunktion aufgerufen wird.

Das Destruktorelement hat den gleichen Namen wie die Klasse, mit einem vorangestellten Tildezeichen (~). Wie ein Konstruktor hat auch der Destruktor keinen Rückgabety. Z.B. sieht die Klasse `Student` mit einem hinzugefügten Destruktor wie folgt aus:

```
class Student
{
public:
    // Datenelemente
    int    nSemesterHours;
    float  dAverage;

    // ein Array für die einzelnen Grade
    int*   pnGrades;

    // Elementfunktionen
```


Lektion 19 – Erhalten der Klassenintegrität 213

```

// Konstruktor - aufgerufen bei Objekterzeugung;
//             initialisiert die Elemente - die

//             Allokierung eines Array vom
//             Heap eingeschlossen
Student()
{
    nSemesterHours = 0;
    dAverage = 0.0;

    // Alloziere Platz für 50 Grade
    pnGrades = new int[50];
}

// Destruktor - aufgerufen bei Vernichtung des
//             Objektes, um den Heapspeicher
//             zurückzugeben
~Student()
{
    // gib den Speicher an den Heap zurück
    delete pnGrades;
    pnGrades = 0;
}
};

```

Wenn mehr als ein Objekt vernichtet wird, werden die Destruktoren in der umgekehrten Reihenfolge der Konstruktoren aufgerufen. Das Gleiche gilt bei der Vernichtung von Objekten, die Klassenobjekte als Datenelemente enthalten. Listing 19-2 zeigt die Ausgabe des Programms aus Listing 19-1 mit hinzugefügten Destruktoren in den drei Klassen:

Listing 19-2: Ausgabe von ConstructMembers, nachdem Destruktor eingefügt wurde

```

Erzeuge ein TutorPair
Konstruktor Student
Konstruktor Teacher
Konstruktor TutorPair
Zurück in main
Destruktor TutorPair
Destruktor Teacher
Destruktor Student

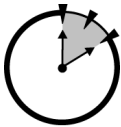
```



Das gesamte Programm ist auf der beiliegenden CD-ROM enthalten.

214 Samstagabend

Der Konstruktor von `TutorPair` wird bei der Deklaration von `tp` aufgerufen. Die `Student`- und `Teacher`-Datenobjekte werden in der Reihenfolge erzeugt, wie sie in `TutorPair` enthalten sind, bevor die Kontrolle an den Body von `TutorPair()` übergeben wird. Wenn die schließende Klammer von `main()` erreicht wird, verlässt `tp` seinen Gültigkeitsbereich. C++ ruft `~TutorPair()` auf, um `tp` zu vernichten. Nachdem der Destruktor das `TutorPair`-Objekt vernichtet hat werden die Destrukto-
ren `~Student()` und `~Teacher()` aufgerufen, um die Datenelemente zu vernichten.

**10 Min.****19.2 Zugriffskontrolle**

Ein Objekt mit einem bekannten Zustand zu initialisieren, ist nur die halbe Miete. Die andere Hälfte besteht darin, sicherzustellen, dass externe Funktionen nicht in das Objekt »eindringen« können und Unsinn mit seinen Datenelementen anstellen können.



Externen Funktionen den Zugriff auf die Datenelemente zu erlauben, ist das Gleiche, wie den Zugriff auf die Interna meiner Mikrowelle zu gestatten. Wenn ich die Mikrowelle aufmache, und die Verkabelung ändere, kann ich den Entwickler der Mikrowelle nur schwerlich für die Folgen verantwortlich machen.

19.2.1 Das Schlüsselwort `protected`

C++ erlaubt es Klassen auch, Datenelemente zu deklarieren, die für Nichtelementfunktionen nicht zugreifbar sind. C++ verwendet das Schlüsselwort `protected`, um Klassenelemente so zu markieren, dass Sie für externe Funktionen zugreifbar sind.



Ein Klassenelement ist `protected`, wenn es nur von anderen Elementen der Klasse angesprochen werden kann.



Der Gegensatz zu `protected` ist `public`. Ein `public`-Element kann von Element- und Nichtelementfunktionen angesprochen werden.

Z.B. in der folgenden Version von `Student` sind nur die Funktionen `grade(double, int)` und `grade()` für externe Funktionen zugreifbar.

```
// ProtectedMembers - demonstriert die Verwendung von
//                      protected-Elementen
#include <stdio.h>
#include <iostream.h>

// Student
class Student
{
    protected:
```

Lektion 19 – Erhalten der Klassenintegrität 215

```
double dCombinedScore;
int    nSemesterHours;

public:
    Student()
    {
        dCombinedScore = 0;
        nSemesterHours = 0;
    }

    // grade - addiere einen weiteren Grad
    double grade(double dNewGrade, int nHours)
    {
        // wenn die Werte gültig sind ...
        if (dNewGrade >= 0 && dNewGrade <= 4.0)
        {
            if (nHours > 0 && nHours <= 5)
            {
                // ...führe Update durch
                dCombinedScore += dNewGrade * nHours;
                nSemesterHours += nHours;
            }
        }
        return grade();
    }

    // grade - gib den Grad zurück
    double grade()
    {
        return dCombinedScore / nSemesterHours;
    }

    // semesterHours - gib die aktuelle Anzahl der
    // Semesterstunden zurück
    int semesterHours()
    {
        return nSemesterHours;
    }
};

int main(int nArgc, char* pszArgs[])
{
    // erzeuge ein Student-Objekt vom Heap
    Student* pS = new Student;

    // addiere ein paar Grade
    pS->grade(2.5, 3);
    pS->grade(4.0, 3);
    pS->grade(3, 3);

    // hole den aktuellen Grad
    cout << »Der Grad ist » << pS->grade() << »\n«;

    return 0;
}
```

216 Samstagabend

Diese Version von `Student` hat zwei Datenelemente. `dCombinedScore` zeigt die Summe der gewichteten Grade, während `nSemesterHours` die Gesamtsumme der Semesterstunden widerspiegelt. Die Funktion `grade(double, int)` führt ein Update für die Summe der gewichteten Grade und die Anzahl der Semesterstunden aus. Die Funktion `grade()` gibt, ihrem Namen entsprechend, den derzeitigen Grad zurück, den sie als Verhältnis der mittleren Grade und der Gesamtanzahl der Semesterstunden berechnet.



`grade(double, int)` **addiert den Effekt eines neuen Kurses zum gesamten Grad, während `grade(void)` den aktuellen Grad zurückgibt. Diese Zweiteilung, bei der eine Funktion einen Wert updatet, während die andere nur den Wert zurückgibt, ist häufig anzutreffen.**

Eine Funktion `grade()`, die den Wert eines Datenelementes zurückgibt, wird als Zugriffsfunktion bezeichnet, weil sie einen kontrollierten Zugriff auf ein Datenelement bereitstellt.

Obwohl die Funktion `grade(double, int)` nicht fehlersicher ist, zeigt Sie doch ein wenig, wie eine Klasse sich selber schützen kann. Die Funktion führt eine Reihe rudimentärer Tests aus, um sicherzustellen, dass die übergebenen Daten Sinn machen. Die Klasse `Student` weiß, dass gültige Grade zwischen 0 und 4 liegen. Außerdem weiß die Klasse, dass die Anzahl der Semesterstunden eines Kurses zwischen 0 und 5 liegen (die Obergrenze ist meine eigene Erfindung).

Die elementaren Überprüfungen, die von der Methode `grade()` durchgeführt werden, stellen eine gewisse Integrität der Daten sicher, vorausgesetzt, die Datenelemente sind für externe Funktionen nicht zugreifbar.



Es gibt noch einen weiteren Kontroll-Level, der als `private` bezeichnet wird. Der Unterschied von `protected` und `private` wird deutlich, wenn wir in Sitzung 32 die Vererbung diskutieren.

Die Elementfunktion `semesterHours()` tut nicht mehr, als den Wert von `nSemesterHours` zurückzugeben.

Eine Funktion, die nichts anderes tut, als externen Funktionen Zugriff auf die Werte von Datenelementen zu geben, wird *Zugriffsfunktion* genannt. Eine Zugriffsfunktion erlaubt es Nichtelementfunktionen, den Wert eines Datenelementes zu lesen, ohne es verändern zu können.

Eine Funktion, die Zugriff auf die `protected`-Elemente einer Klasse hat, wird als *vertrauenswürdige* Funktion bezeichnet. Alle Elementfunktionen sind vertrauenswürdig. Auch Nichtelementfunktionen können als vertrauenswürdig bezeichnet werden durch die Verwendung des Schlüsselwortes `friend` (Freund). Eine Funktion, die als Freund einer Klasse bezeichnet ist, ist vertrauenswürdig. Alle Elementfunktionen einer `friend`-Klasse sind Freunde. Der richtige Gebrauch von `friend` geht über die Zielsetzung dieses Buches hinaus.

19.2.2 Statische Datenelemente

Unabhängig davon, wie viele Elemente `protected` sind, ist unsere Klasse `LinkedList` aus Sitzung 15 doch noch verwundbar durch externe Funktionen durch den globalen Kopfzeiger. Was wir eigentlich möchten, ist diesen Zeiger unter den Schutz der Klasse zu stellen. Wie können jedoch kein normales Datenelement verwenden, weil diese für jede Instanz von `LinkedList` separat angelegt werden – es kann nur einen Kopfzeiger in einer verketteten Liste geben. C++ bietet hierfür eine Lösung mit statischen Datenelementen.

Ein *statisches Datenelement* wird nicht für alle Objekte der Klasse separat instanziiert. Alle Objekte der Klasse teilen sich das gleiche statische Element.

Die Syntax zur Deklaration statischer Datenelemente ist ein bißchen unangenehm:

```
class LinkedList
{
    protected:
        // deklariere pHead als Element der Klasse,
        // aber gleich für alle Objekte
        static LinkedList* pHead;

        // der Zeiger pNext wird für jedes Objekt
        // separat angelegt
        LinkedList* pNext;

        // addHead - fügt ein Datenelement an den Anfang
        // der Liste an
        void addHead()
        {
            // verkette den aktuellen Eintrag und
            // den Kopf der Liste
            pNext = pHead;

            // setze den Kopfzeiger auf das aktuelle
            // Objekt (this)
            pHead = this;
        }

        // ... der Rest des Programms ...
};

// alloziere jetzt einen Speicherbereich, in dem die
// statischen Elemente abgelegt werden können;
// stellen Sie sicher, dass die Objekte hier
// initialisiert werden, weil der Konstruktor
// das nicht tut
LinkedList* LinkedList::pHead = 0;
```

218 Samstagabend

Die statische Deklaration in der Klasse macht `pHead` zu einem Element, alloziert aber keinen Speicher dafür. Dies muss außerhalb der Klasse geschehen, wie oben zu sehen ist.

Die gleiche Funktion `addHead()` greift auf `pHead` zu, wie sie auf jedes andere Datenelement zugreifen würde. Zuerst lässt sie den `pNext`-Zeiger des aktuellen Objekts auf den Anfang der Liste zeigen – der Eintrag, auf den `pHead` zeigt. Danach verändert es diesen Zeiger, auf den aktuellen Eintrag zu zeigen.

Erinnern Sie sich daran, dass die Adresse des aktuellen Eintrags über das Schlüsselwort `this` angesprochen werden kann.

**Tipp**

So einfach `addHead()` ist, untersuchen Sie die Funktion dennoch genau: Alle Objekte der Klasse `LinkedList` haben das gleiche Element `pHead`, jedes Objekt hat jedoch seinen eigenen `pNext`-Zeiger.

**Hinweis**

Es ist auch möglich, eine Elementfunktion statisch zu deklarieren; in diesem Buch verwenden wir jedoch keine solchen Funktionen.

Zusammenfassung

Der Konstruktor ist eine spezielle Elementfunktion, die C++ automatisch aufruft, wenn ein Objekt erzeugt wird, ob nun eine lokale Variable ihren Gültigkeitsbereich betritt oder ob ein Objekt vom Heap alloziert wird. Der Konstruktor ist verantwortlich dafür, die Datenelemente mit einem gültigen Zustand zu initialisieren. Die Datenelemente einer Klasse werden automatisch erzeugt, bevor der Konstruktor der Klasse aufgerufen wird. C++ ruft eine spezielle Funktion auf, wenn ein Objekt vernichtet wird, die als Destruktor bezeichnet wird.

- Der Klassenkonstruktor gibt der Klasse die Kontrolle darüber, wie Objekte erzeugt werden. Das verhindert, dass Objekte ihr Leben in einem illegalen Zustand beginnen. Konstruktoren werden wie die anderen Elementfunktionen deklariert, außer dass sie den Namen der Klasse tragen und keinen Rückgabetyt besitzen (nicht einmal `void`).
- Der Klassendestruktor gibt der Klasse die Chance, Ressourcen, die von einem Konstruktor belegt wurden, wieder freizugeben. Die häufigste Ressource ist Speicher.
- Eine Funktion `protected` zu deklarieren, macht sie nicht zugreifbar für nicht vertrauenswürdige Funktionen. Elementfunktionen werden automatisch als vertrauenswürdig angesehen.

Lektion 19 – Erhalten der Klassenintegrität 219

Selbsttest

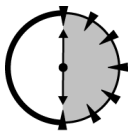
1. Was ist ein Konstruktor? (Siehe »Der Konstruktor«)
2. Was ist falsch daran, eine Funktion `init()` zur Initialisierung eines Objektes aufzurufen, wenn es angelegt wird? (Siehe »Der Konstruktor«)
3. Was ist der vollständige Name eines Konstruktors der Klasse `Teacher`? Was ist der Rückgabotyp? (Siehe »Der Konstruktor«)
4. In welcher Reihenfolge werden Datenelemente eines Objektes angelegt? (Siehe »Konstruktion von Datenelementen«)
5. Was ist der vollständige Name des Destruktors der Klasse `Teacher`? (Siehe »Der Destruktor«)
6. Was ist die Bedeutung des Schlüsselwortes `static`, wenn es in Verbindung mit Datenelementen verwendet wird? (Siehe »Statische Datenelemente«)



Lektion 20 *Klassenkonstruktoren II*

Checkliste

- ☒ Konstruktoren mit Argumenten erzeugen
- ☒ Argumente an die Konstruktoren von Datenelementen übergeben
- ☒ Konstruktionsreihenfolge der Datenelemente bestimmen
- ☒ Spezielle Eigenschaften des Kopierkonstruktors bestimmen



30 Min.

Ein sehr einfacher Konstruktor arbeitet gut in der einfachen Klasse `Student` in Sitzung 19. Ohne dass die Klasse `Student` sehr komplex wird, treten die Begrenzungen des Konstruktors nicht zu Tage. Bedenken Sie z.B., dass ein `Student` einen Namen und eine Sozialversicherungsnummer hat. Der Konstruktor aus Sitzung 19 hat keine Argumente, so hat er keine andere Wahl, als das Objekt »leer« zu initialisieren. Der Konstruktor soll ein gültiges Objekt erzeugen – ein namenloser `Student` ohne Sozialversicherungsnummer stellt sicherlich keinen gültigen Studenten dar.

20.1 Konstruktoren mit Argumenten

C++ erlaubt es dem Programmierer, Konstruktoren mit Argumenten zu definieren, wie in Listing 20-1 zu sehen ist.

Listing 20-1: Definition eines Konstruktors mit Argumenten

```
// NamedStudent - zeigt, wie Argumente im Konstruktor
//                  eine Klasse realistischer machen
#include <stdio.h>
#include <iostream.h>
#include <string.h>

// Student
class Student
{
public:
```


Lektion 20 – Klassenkonstruktoren II 221

```
Student(char *pszName, int nID)
{
    // Speichere eine Kopie des Namens:
    // allokieren vom Heap die gleiche Größe
    // wie die übergebene Zeichenkette
    int nLength = strlen(pszName) + 1;
    this->pszName = new char[nLength];

    // kopiere den übergebenen Namen in
    // den neu allozierten Puffer
    strcpy(this->pszName, pszName);

    // speichere die Studenten-ID
    this->nID = nID;
}

~Student()
{
    delete pszName;
    pszName = 0;
}

protected:
    char* pszName;
    int nID;
};

void fn()
{
    // erzeuge einen lokalen Studenten
    Student s1(»Stephen Davis«, 12345);

    // erzeuge einen Studenten vom Heap
    Student* pS2 = new Student(»Ron Davis«, 67890);

    // stelle sicher, dass aller Speicher an den
    // Heap zurückgegeben wird
    delete pS2;
}
```

Der Konstruktor `Student(char*, int)` beginnt mit der Allokierung eines Speicherblocks vom Heap, um eine Kopie der übergebenen Zeichenkette zu speichern. Die Funktion `strlen()` gibt die Anzahl der Bytes in der Zeichenkette zurück; weil `strlen()` das Nullzeichen nicht mitzählt, müssen wir 1 addieren, um die Anzahl Bytes zu erhalten, die wir vom Heap benötigen. Der Konstruktor `Student()` kopiert die übergebene Zeichenkette in diesen Speicherblock. Schließlich weist der Konstruktor den Studenten die ID zu.

Die Funktion `fn()` erzeugt zwei `Student`-Objekte, eins lokal in der Funktion und eins vom Heap. In beiden Fällen übergibt `fn()` den Namen und die ID an das `Student`-Objekt, das erzeugt wird.



In diesem Beispiel `NamedStudent` tragen die Argumente des Konstruktors `Student` die gleichen Namen wie die Datenelemente, die sie initialisieren. Das ist keine Anforderung von C++, sondern nur eine Konvention, die ich bevorzuge. Eine lokal definierte Variable hat jedoch Vorrang vor einem Datenelement mit dem gleichen Namen. Somit bezieht sich eine unqualifizierte Referenz innerhalb von `Student()` auf `pszName` auf das Argument; `this->pszName` gestattet jedoch immer, das Datenelement anzusprechen, unabhängig von allen lokal deklarierten Variablen. Während einige diese Praxis verwirrend finden, denke ich, dass sie die Verbindung zwischen den Argumenten und den Datenelementen unterstreicht. In jedem Fall sollten Sie mit dieser üblichen Praxis vertraut sein.

Konstrukturen können überladen werden, wie Funktionen mit Argumenten überladen werden können.



Erinnern Sie sich, dass das Überladen einer Funktion bedeutet, dass Sie zwei Funktionen mit dem gleichen Namen haben, die sich durch ihre unterschiedlichen Argumenttypen unterscheiden.

C++ wählt den entsprechenden Konstruktor, basierend auf den Argumenten in der Deklaration. Z.B. kann die Klasse `Student` gleichzeitig drei Konstruktoren haben, wie im folgenden Schnipsel:

```
#include <iostream.h>
#include <string.h>
class Student
{
public:
    // die verfügbaren Konstruktoren
    Student();
    Student(char* pszName);
    Student(char* pszName, int nID);
    ~Student();

protected:
    char* pszName;
    int nID;
};

// das Folgende ruft jeden Konstruktor auf
int main(int nArgs, char* pszArgs[])
{
    Student noName;
    Student freshMan(>>Smel E. Fish<<);
    Student xfer(>>Upp R. Classman<<, 1234);
    return 0;
}
```

Lektion 20 – Klassenkonstruktoren II 223

Weil das Objekt `noName` ohne Argument erscheint, wird es von dem Konstruktor `Student::Student()` erzeugt. Dieser Konstruktor wird als *Defaultkonstruktor* (oder auch *void-Konstruktor*) bezeichnet. (Ich persönlich bevorzuge den zweiten Begriff, aber der erste ist üblicher, weshalb er in diesem Buch verwendet wird.)

Es ist oft der Fall, dass der einzige Unterschied zwischen einem Konstruktor und einem anderen das Hinzufügen eines Defaultwertes für ein fehlendes Argument ist. Nehmen Sie z.B. an, dass ein `Student`, der ohne ID erzeugt wird, die ID 0 erhält. Um Extraarbeit zu vermeiden, ermöglicht es C++ dem Programmierer, einen Defaultwert zuzuweisen. Ich hätte die beiden letzten Konstrukturen wie folgt kombinieren können:

```
Student(char* pszName, int nID = 0);

Student s1(>Lynn Anderson<, 1234);
Student s2(>Stella Prater<);
```

Beide, `s1` und `s2`, werden durch den gleichen Konstruktor `Student(char*, in)` konstruiert. Im Falle von `Stella`, wird ein Defaultwert von 0 dem Konstruktor bereitgestellt.



Tipp

Defaultwerte für Argumente können in jeder Funktion angegeben werden; ihren größten Nutzen finden Sie jedoch bei der Reduzierung der Konstruktorenanzahl.

20.2 Konstruktion von Klassenelementen

C++ konstruiert Datenelementobjekte zur gleichen Zeit, wie das Objekt selber. Es gibt keinen offensichtlichen Weg, Argumente bei der Konstruktion von Datenelementen zu übergeben.



Hinweis

In den Beispielen in Sitzung 19 gab es keinen Grund, Argumente an die Konstruktoren der Datenelemente zu übergeben – diese Version von `Student` hat sich auf die Anwendung verlassen, die Funktion `init()` aufzurufen, um Objekte mit gültigen Werten zu initialisieren.

Betrachten Sie Listing 20-2, in dem die Klasse `Student` ein Objekt der Klasse `StudentId` enthält. Ich habe beide mit Ausgabeanweisungen in den Konstruktoren ausgestattet, um zu sehen, was passiert.

Listing 20-2: Konstruktor `Student` mit einem Elementobjekt aus der Klasse `StudentId`

```
// DefaultStudentId - erzeuge ein Student-Objekt mit
//                      der nächsten verfügbaren ID
#include <stdio.h>
#include <iostream.h>
#include <string.h>

// StudentId
int nNextStudentId = 0;
class StudentId
```

224 Samstagabend

```

{
    public:
        StudentId()
        {
            nId = ++nNextStudentId;
            cout << »Konstruktor StudentId » << nId << »\n«;
        }

        ~StudentId()
        {
            cout << »Destruktor StudentId »
                << nId << »\n«;
        }
    protected:
        int nId;
};

// Student
class Student
{
    public:
        // Constructor - erzeuge Student mit
        // dem gegebenen Namen
        // (»kein Name« als Default)
        Student(char *pszName = »kein Name«)
        {
            cout << »Konstruktor Student » << pszName << »\n«;

            // kopiere übergebene Zeichenkette in Element
            this->pszName = new char[strlen(pszName) + 1];
            strcpy(this->pszName, pszName);
        }

        ~Student()
        {
            cout << »Destruktor Student » << pszName << »\n«;
            delete pszName;
            pszName = 0;
        }

    protected:
        char* pszName;
        StudentId id;
};

int main(int nArgs, char* pszArg[])
{
    Student s(»Randy«);
    return 0;
}

```

Eine Studenten-ID wird jedem Studenten zugeordnet, wenn ein Student-Objekt erzeugt wird. In diesem Beispiel werden IDs fortlaufend vergeben unter Verwendung der globalen Variable `nNextStudentId`, die die nächste zu vergebende ID enthält.

Lektion 20 – Klassenkonstruktoren II 225

Die Ausgabe dieses einfachen Programms sieht so aus:

```
Konstruktor StudentId 1
Konstruktor Student Randy
Destruktor Student Randy
Destruktor StudentId 1
```

Beachten Sie, dass die Meldung vom Konstruktor `StudentId( )` vor der Ausgabe des Konstruktors `Student( )` erscheint, und die Meldung des `StudentId`-Destruktors nach der Ausgabe des `Student`-Destruktors erscheint.



Wenn alle Konstruktoren hier etwas ausgeben, könnten sie denken, dass jeder Konstruktor etwas ausgeben muss. Die meisten Konstruktoren geben nichts aus. Konstruktoren in Büchern tun es, weil die Leser normalerweise den guten Rat des Autors ignorieren, Schritt für Schritt durch die Programme zu gehen.

Wenn der Programmierer keinen Konstruktor bereitstellt, ruft der Defaultkonstruktor, der von C++ automatisch bereitgestellt wird, die Defaultkonstruktoren der Datenelemente auf. Das Gleiche gilt für den Destruktor, der automatisch die Destruktoren der Datenelemente aufruft, die Destruktoren haben. Der Defaultdestruktor von C++ tut das Gleiche.

Das ist alles großartig für den Defaultkonstruktor. Aber was ist, wenn wir einen anderen Konstruktor als den Defaultkonstruktor aufrufen wollen? Wo tun wir das Objekt hin? Um das zu verdeutlichen, lassen Sie uns annehmen, dass, statt der Berechnung der `Studenten-ID`, diese als Argument des Konstruktors bereitgestellt wird.

Lassen Sie uns ansehen, wie es nicht funktioniert. Betrachten Sie das Programm in Listing 20-3.

Listing 20-3: So kann ein Datenelementobjekt nicht erzeugt werden

```
// FalseStudentId - es wird versucht, die StudentId
//                  mit einem Konstruktor zu
//                  erzeugen, der nicht default ist
#include <stdio.h>
#include <iostream.h>
#include <string.h>

// StudentId
int nNextStudentId = 0;
class StudentId
{
public:
    StudentId(int nId = 0)
    {
        this->nId = nId;
        cout << »Konstruktor StudentId » << nId << »\n«;
    }

    ~StudentId()
    {
        cout << »Destruktor StudentId »
              << nId << »\n«;
    }
protected:
```

226 Samstagabend

```

    int nId;
};

// Student
class Student
{
public:
    // Konstruktor - erzeuge Student mit
    // dem gegebenen Namen und
    // Studenten-ID
    Student(char *pszName = »kein Name«,
            int ssId = 0)
    {
        cout << »Konstruktor Student » << pszName << »\n«;

        // kopiere übergebene Zeichenkette in Element
        this->pszName = new char[strlen(pszName) + 1];
        strcpy(this->pszName, pszName);

        // das funktioniert nicht
        StudentId id(ssId); // konstruiere Studenten-ID
    }

    ~Student()
    {
        cout << »Destruktor Student » << pszName << »\n«;
        delete pszName;
        pszName = 0;
    }

protected:
    char* pszName;
    StudentId id;
};

int main(int nArgs, char* pszArg[])
{
    Student s(»Randy«, 1234);
    cout << »Nachricht von main\n«;
    return 0;
}

```

Der Konstruktor für `StudentId` wurde so verändert, dass er einen Wert von außen entgegennimmt (der Defaultwert ist notwendig, um das Beispiel kompilieren zu können aus Gründen, die bald klar werden). Innerhalb des Konstruktors von `Student` hat der Programmierer (das bin ich) (clever) versucht, ein Objekt `StudentId` mit Namen `id` zu erzeugen.

Die Ausgabe des Programms zeigt ein Problem:

```

Konstruktor StudentId 0
Konstruktor Student Randy
Konstruktor StudentId 1234
Destruktor StudentId 1234
Nachricht von main
Destruktor Student Randy
Destruktor StudentId 0

```

Lektion 20 – Klassenkonstruktoren II 227

Zum einen scheint der Konstruktor zweimal aufgerufen zu werden, das erste Mal mit null, bevor der Konstruktor `Student` beginnt, und das zweite Mal mit dem erwarteten Wert 1234 vom Konstruktor `Student` aus. Offensichtlich wurde ein zweites Objekt `StudentId` erzeugt innerhalb des Konstruktors `Student`, das vom `StudentId`-Datenelement verschieden ist.

Die Erklärung für dieses bizarre Verhalten ist, dass das Datenelement `id` bereits existiert, wenn der Body des Konstruktors betreten wird. Anstatt ein existierendes Datenelement `id` zu erzeugen, erzeugt die Deklaration im Konstruktor ein lokales Objekt mit dem gleichen Namen. Dieses lokale Objekt wird vernichtet, wenn der Konstruktor wieder verlassen wird.

Wir brauchen einen anderen Mechanismus um anzuzeigen »konstruiere das existierende Element; erzeuge kein neues.« C++ definiert die neue Konstruktion wie folgt:

```
class Student
{
public:
    Student(char* pszName = »no name«, int ssId = 0)
        : id(ssId) // konstruiere Datenelement id mit
                  // dem angegebenen Wert
    {
        cout << »Konstruktor Student » << pszName << »\n«;
        this->pszName = new char[strlen(pszName) + 1];
        strcpy(this->pszName, pszName);
    }
    // ...
};
```

Beachten Sie insbesondere die erste Zeile des Konstruktors. Wir haben das vorher noch nicht gesehen. Der Doppelpunkt »:« bedeutet, dass Aufrufe von Konstruktoren der Datenelemente der aktuellen Klasse folgen. Für den C++-Compiler liest sich die Zeile so: »Konstruiere das Element `id` mit dem Argument `ssId` des `Student`-Konstruktors. Alle Datenelemente, die nicht so behandelt werden, werden mit ihrem Defaultkonstruktor konstruiert.«



Das gesamte Programm befindet sich auf der beiliegenden CD-ROM unter dem Namen `StudentId`.

Das neue Programm erzeugt das erwartete Ergebnis:

```
Konstruktor StudentId 1234
Konstruktor Student Randy
Nachricht von main
Destruktor Student Randy
Destruktor StudentId 1234
```

228 Samstagabend

Das Objekt `s` wird in `main()` erzeugt mit dem Studentennamen »Randy« und einer Student-ID von 1234. Dieses Objekt wird am Ende von `main()` automatisch vernichtet. Die »:«-Syntax muss auch bei der Wertzuweisung an Elemente vom Typ `const` oder mit Referenztyp verwendet werden. Betrachten Sie die folgende dumme Klasse:

```
class SillyClass
{
public:
    SillyClass(int& i) : nTen(10), refI(i)
    {
    }
protected:
    const int nTen;
    int& refI;
};
```

Wenn das Programm mit der Ausführung des Bodys des Konstruktors beginnt, sind die beiden Elemente `nTen` und `refI` bereits angelegt. Diese Datenelemente müssen initialisiert sein, bevor die Kontrolle an den Body des Konstruktors übergeht.



Jedes Datenelement kann mit der »im-Voraus«-Syntax deklariert werden.

Tip

20.3 Reihenfolge der Konstruktion

Wenn es mehrere Objekte gibt, die alle einen Konstruktor besitzen, kümmert sich der Programmierer normalerweise nicht um die Reihenfolge, in der die Dinge erzeugt werden. Wenn einer oder mehrere der Konstruktoren Seiteneffekte haben, kann die Reihenfolge einen Unterschied machen.



Hinweis

Ein Seiteneffekt ist eine Zustandsänderung, die durch eine Funktion verursacht wird, die aber nicht zu den Argumenten oder dem zurückgegebenen Objekt gehört. Wenn z.B. eine Funktion einer globalen Variablen einen Wert zuweist, wäre das ein Seiteneffekt.

Die Regeln für die Reihenfolge der Konstruktion sind:

- Lokale und statische Objekte werden in der Reihenfolge erzeugt, in der ihre Deklarationen aufgerufen werden.
- Statische Objekte werden nur einmal erzeugt.
- Alle globalen Objekte werden vor `main()` konstruiert.
- Globale Objekte werden in keiner bestimmten Reihenfolge angelegt.
- Elemente werden in der Reihenfolge erzeugt, in der sie in der Klasse deklariert sind.

20.3.1 Lokale Objekte werden in der Reihenfolge konstruiert

Lokale Objekte werden in der Reihenfolge konstruiert, in der das Programm ihre Deklaration antrifft. Normalerweise ist das die gleiche Reihenfolge, in der das Programm die Objekte antrifft, außer die Funktion springt um gewisse Deklarationen herum. (So nebenbei, Deklarationen zu überspringen ist eine üble Sache. Es verwirrt den Leser und den Compiler.)

20.3.2 Statische Objekte werden nur einmal angelegt

Statische Objekte sind lokalen Variablen ähnlich, nur dass Sie nur einmal angelegt werden. Das ist auch zu erwarten, weil sie ihren Wert von einem Funktionsaufruf zum nächsten behalten. Anders als bei C, wo statische Variablen initialisiert werden, wenn das Programm beginnt, muss C++ warten, bis die Kontrolle das erste Mal ihre Deklaration durchläuft, um dann die Konstruktion der Variablen auszuführen. Betrachten sie das folgende triviale Programm:

```
class DoNothing
{
public:
    DoNothing(int initial)
    {
    }
};

void fn(int i)
{
    static DoNothing staticDN(1);
    DoNothing localDN(2);
}
```

Die Variable `staticDN` wird beim ersten Aufruf von `fn( )` konstruiert. Beim zweiten Aufruf von `fn( )` konstruiert das Programm nur `localDN`.

20.3.3 Alle globalen Variablen werden vor `main( )` erzeugt

Alle globalen Variablen betreten ihren Gültigkeitsbereich unmittelbar bei Programmstart. Somit werden alle globalen Objekte konstruiert, bevor die Kontrolle an `main( )` übergeht.

20.3.4 Keine bestimmte Reihenfolge für globale Objekte

Die Reihenfolge der Konstruktion von lokalen Objekten herauszufinden, ist einfach. Durch den Kontrollfluss ist eine Reihenfolge gegeben. Bei globalen Objekten gibt es keinen solchen Fluss, der eine Reihenfolge vorgeben würde. Alle globalen Variablen betreten gleichzeitig ihren Geltungsbereich, erinnern Sie sich? Nun, Sie werden sicherlich anführen, dass der Compiler doch eine Datei von oben nach unten durcharbeiten könnte, um alle globalen Variablen zu finden. Das ist für eine einzelne Datei richtig (und ich nehme an, dass die meisten Compiler genau das tun).

Unglücklicherweise bestehen die meisten Programme in der realen Welt aus mehreren Dateien, die getrennt kompiliert werden, und dann gemeinsam zum Programm zusammengefügt werden. Weil der Compiler keinen Einfluss darauf hat, wie diese Teile zusammengefügt werden, hat er keinen Einfluss darauf, wie globale Objekte von Datei zu Datei erzeugt werden.

230 Samstagabend

Meistens ist die Reihenfolge gleichgültig. Hin und wieder können aber dadurch Bugs entstehen, die sehr schwierig zu finden sind. (Es passiert jedenfalls oft genug, um in einem Buch Erwähnung zu finden.)

Betrachten Sie das Beispiel:

```
class Student
{
public:
    Student (unsigned id) : studentId(id)
    // unsigned ist vorzeichenloses int
    {
    }
    const unsigned studentId;
};

class Tutor
{
public:
    // Konstruktor - weise dem Tutor einen Studenten
    // zu durch Auslesen seiner ID
    Tutor(Student &s)
    {
        tutoredId = s.studentId;
    }
protected:
    unsigned tutoredId;
};

// erzeuge global einen Student
Student randy(1234);

// weise diesem Student einen Tutor zu
Tutor jenny(randy);
```

Hier weist der Konstruktor von `Student` eine Studenten-ID zu. Der Konstruktor von `Tutor` kopiert sich diese ID. Das Programm deklariert einen Studenten `randy` und weist dann diesem Student den Tutor `jenny` zu.

Das Problem ist, dass wir implizit annehmen, dass `randy` vor `jenny` konstruiert wird. Nehmen wir an, es wäre anders herum. Dann würde `jenny` mit einem Speicherblock initialisiert, der noch nicht in ein `Student`-Objekt verwandelt wurde und daher »Müll« als Studenten-ID enthält.



Dieses Beispiel ist nicht besonders schwer zu durchschauen und mehr als nur ein wenig konstruiert. Trotzdem können Probleme mit globalen Objekten, die in keiner bestimmten Ordnung konstruiert werden, sehr subtil sein. Um diese Probleme zu vermeiden, sollten Sie es dem Konstruktor eines globalen Objektes nicht gestatten, sich auf den Inhalt eines anderen globalen Objektes zu verlassen.

Lektion 20 – Klassenkonstruktoren II**231****20.3.5 Elemente werden in der Reihenfolge ihrer Deklaration konstruiert**

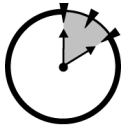
Elemente einer Klasse werden in der Reihenfolge konstruiert, in der sie in der Klasse deklariert wurden. Das ist nicht so offensichtlich wie es aussieht. Betrachten Sie das folgende Beispiel:

```
class Student
{
public:
    Student (unsigned id, unsigned age) :
        sAge(age), sId(id)
    {
    }
    const unsigned sId;
    const unsigned sAge;
};
```

In diesem Beispiel wird `sId` konstruiert, bevor `sAge` überhaupt daran denken kann, als zweites in der Initialisierungsliste des Konstruktors zu stehen. Sie würden nur dann einen Unterschied in der Reihenfolge der Konstruktion bemerken, wenn beide Elemente Objekte wären, deren Konstruktoren gegenseitige Seiteneffekte haben.

20.3.6 Destruktoren in umgekehrter Reihenfolge wie Konstruktoren

Schließlich werden die Destruktoren, egal in welcher Reihenfolge die Konstruktoren aufgerufen wurden, in der umgekehrten Reihenfolge aufgerufen. (Es ist schön zu wissen, dass es in C++ wenigstens eine Regel ohne Wenn und Aber gibt.)

**10 Min.****20.4 Der Kopierkonstruktor**

Ein Kopierkonstruktor ist ein Konstruktor, der den Namen `X::X(&X)` trägt, wobei `X` ein Klassenname ist. D.h. er ist der Konstruktor einer Klasse `X`, und nimmt als Argument eine Referenz auf ein Objekt der Klasse `X`. Nun, ich weiß, dass das wirklich nutzlos klingt, aber denken Sie einen Augenblick darüber nach, was passiert, wenn

Sie eine Funktion wie die folgende aufrufen:

```
void fn1(Student fs)
{
    // ...
}
int fn2()
{
    Student ms;
    fn1(ms);
    return 0;
}
```

Wie sie wissen, wird eine Kopie des Objektes `ms` – und nicht das Objekt selber – an die Funktion `fn1( )` übergeben. C++ könnte einfach eine exakte Kopie des Objektes machen und diese an `fn1( )` übergeben.

232 Samstagabend

Das ist in C++ nicht akzeptabel. Erstens wird zur Konstruktion eines Objektes ein Konstruktor benötigt, selbst für das Kopieren eines existierenden Objektes. Zweitens, was ist, wenn wir keine einfache Kopie des Objektes haben möchten? (Lassen wir das »Warum?« für eine Weile.) Wir müssen in der Lage sein, anzugeben, wie die Kopie konstruiert werden soll.

Das Programm CopyStudent in Listing 20-4 demonstriert diesen Punkt.

Listing 20-4: Der Kopierkonstruktor in Aktion

```
// CopyStudent - demonstriert den Kopierkonstruktor
//                bei einer Wertübergabe
#include <stdio.h>
#include <iostream.h>
#include <string.h>

class Student
{
public:
    // Initialisierungsfunktion
    void init(char* pszName, int nId, char* pszPreamble = »\0«)
    {
        int nLength = strlen(pszName)
                      + strlen(pszPreamble)
                      + 1;
        this->pszName = new char[nLength];
        strcpy(this->pszName, pszPreamble);
        strcat(this->pszName, pszName);
        this->nId = nId;
    }

    // Konventioneller Konstruktor
    Student(char* pszName = »noname«, int nId = 0)
    {
        cout << »Konstruktor Student » << pszName << »\n«;
        init(pszName, nId);
    }

    // Kopierkonstruktor
    Student(Student &s)
    {
        cout << »Kopierkonstruktor von »
              << s.pszName
              << »\n«;
        init(s.pszName, s.nId, »Kopie von »);
    }

    ~Student()
    {
        cout << »Destruktor » << pszName << »\n«;
        delete pszName;
    }

protected:
    char* pszName;
    int   nId;
};
```

Lektion 20 – Klassenkonstruktoren II**233**

```
// fn - erhält Argument als Wert
void fn(Student s)
{
    cout << »In Funktion fn()\n«;
}

int main(int nArgs, char* pszArgs[])
{
    // erzeuge ein Student-Objekt
    Student randy(»Randy«, 1234);

    // übergib es als Wert an fn()
    cout << »Aufruf von fn()\n«;
    fn(randy);
    cout << »Rückkehr von fn()\n«;

    // fertig
    return 0;
}
```

Die Ausgabe des Programms sieht so aus:

```
Konstruktor Student Randy
Aufruf von fn( )
Kopierkonstruktor Student Randy
In Funktion fn( )
Destruktor Kopie von Randy
Rückkehr von fn( )
Destruktor Randy
```

Beginnend bei `main( )` sehen wir, wie dieses Programm arbeitet. Der normale Konstruktor erzeugt die erste Nachricht. `main( )` erzeugt die Nachricht »Aufruf von ...«. C++ ruft den Kopierkonstruktor auf, um eine Kopie von `randy` an `fn( )` zu übergeben, was die nächste Zeile der Ausgabe erzeugt. Die Kopie wird am Ende von `fn( )` vernichtet. Das originale Objekt `randy` wird am Ende von `main( )` vernichtet.

(Dieser Kopierkonstruktor macht ein wenig mehr, als nur eine Kopie des Objekts; er trägt die Phrase "Kopie von" am Anfang des Namens ein. Das war zu Ihrem Vorteil. Normalerweise sollten sich Kopierkonstruktoren darauf beschränken, einfach Kopien zu konstruieren. Aber prinzipiell können sie alles tun.)

20.4.1 Flache Kopie gegen tiefe Kopie

C++ sieht den Kopierkonstruktor als so wichtig an, dass C++ selber einen Kopierkonstruktor erzeugt, wenn Sie keinen definieren. Der Default-Kopierkonstruktor, den C++ bereitstellt, führt eine Element-zu-Element-Kopie durch.

Eine Element-zu-Element-Kopie durchzuführen, ist die offensichtliche Aufgabe eines Kopierkonstruktors. Außer zum Hinzufügen so unsinniger Dinge wie "Kopie von" an den Anfang eines Studentennamen, wo sonst würden wir eine Element-zu-Element-Kopie erstellen?

Betrachten Sie, was passiert, wenn der Konstruktor einen Speicherbereich vom Heap alloziert. Wenn der Kopierkonstruktor einfach eine Kopie davon macht, ohne seinen eigenen Speicher anzulegen, kommen wir in eine schwierige Situation: zwei Objekte denken, sie hätten exklusiven Zugriff

234 Samstagabend

auf den gleichen Besitz. Das wird noch schlimmer, wenn der Destruktor für beide Objekte aufgerufen wird und sie beide versuchen, den Speicher zurückzugeben. Um das konkreter werden zu lassen, betrachten Sie die Klasse `Student` nochmals:

```
class Student
{
public:
    Person(char *pszName)
    {
        pszName = new char[strlen(pszName) + 1];
        strcpy(pName, pN);
    }
    ~Person()
    {
        delete pszName;
    }
protected:
    char* pszName;
};
```

Hier alloziert der Konstruktor Speicher vom Heap, um den Namen der Person zu speichern – der Destruktor gibt freundlicherweise den Speicher zurück, wie er soll. Wenn dieses Objekt als Wert an eine Funktion übergeben wird, würde C++ eine Kopie des `Student`-Objektes machen, den Zeiger `pszName` eingeschlossen. Das Programm hat dann zwei `Student`-Objekte, die auf den gleichen Speicherbereich zeigen, der den Namen des Studenten enthält. Wenn das erste Objekt vernichtet wird, wird der Speicherbereich zurückgegeben. Bei der Vernichtung des zweiten Objektes wird versucht, den gleichen Speicher erneut freizugeben – ein fataler Vorgang für ein Programm.



Speicher vom Heap ist nicht der einzige Besitz, der eine tiefe Kopie erforderlich macht, aber der häufigste. Offene Dateien, Ports und allozierte Hardware (wie z.B. Drucker) benötigen ebenfalls tiefe Kopien. Das sind dieselben Typen, die der Destruktor zurückgeben muss. Eine gute Daumenregel ist, dass ihre Klasse einen Kopierkonstruktor benötigt, wenn sie einen Destruktor benötigt, um Ressourcen wieder freizugeben.

20.4.2 Ein Fallback-Kopierkonstruktor

Es gibt Situationen, in denen Sie keine Kopien Ihrer Objekte erzeugt haben möchten. Das kann der Fall sein, weil Sie den Code, um eine tiefe Kopie zu erzeugen, nicht bereitstellen können oder wollen. Das ist auch der Fall, wenn das Objekt sehr groß ist und das Erzeugen einer Kopie, flach oder tief, einige Zeit benötigen würde.

Eine einfache Verteidigungsposition, um das Problem zu vermeiden, dass ohne Ihr Wissen flache Kopien von Objekten angelegt werden, ist das Erzeugen eines Kopierkonstruktors, der als `protected` deklariert ist.

Lektion 20 – Klassenkonstruktoren II

235



Wenn Sie keinen Kopierkonstruktor selber erzeugen, legt C++ einen eigenen Kopierkonstruktor an.

Im folgenden Beispiel kann C++ keine Kopie der Klasse `BankAccount` anlegen, wodurch sichergestellt ist, dass das Programm nicht versehentlich eine Überweisung auf die Kopie eines Kontos und nicht auf das Konto selber ausführt.

```
class BankAccount
{
protected:
    int nBalance;
    BankAccount(BankAccount& ba)
    {
    }

public:
    // ... Rest der Klasse ...
};
```



0 Min.

Zusammenfassung

Der Konstruktor hat die Aufgabe, Objekte zu erzeugen und mit einem gültigen Zustand zu initialisieren. Viele Objekte haben jedoch keinen gültigen Default-Zustand. Z.B. ist es nicht möglich, ein gültiges `Student`-Objekt ohne einen Namen und eine ID zu erzeugen. Ein Konstruktor mit Argumenten ermöglicht es dem Programm, initiale Werte für neu erzeugte Objekte zu übergeben. Diese Argumente können auch an andere Konstruktoren der Datenelemente der Klasse weitergegeben werden. Die Reihenfolge, in der die Datenelemente konstruiert werden, ist im C++-Standard definiert (eines der wenigen Dinge, die wohldefiniert sind).

- Argumente von Konstruktoren erlauben die Festlegung eines initialen Zustandes von Objekten durch das Programm; für den Fall, dass diese initialen Werte nicht gültig sind, muss die Klasse einen Reservezustand haben.
- Es muss vorsichtig mit Konstruktoren umgegangen werden, die Seiteneffekte haben, wie z.B. das Ändern globaler Variablen, weil einer oder mehrere Konstruktoren kollidieren können, wenn sie ein Objekt erzeugen, das Datenelemente enthält, die selber Objekte sind.
- Ein spezieller Konstruktor, der Kopierkonstruktor, hat den Prototyp `X : X(&X)`, wobei `X` der Name der Klasse ist. Dieser Konstruktor wird dazu verwendet, eine Kopie eines existierenden Objektes zu erzeugen. Kopierkonstruktoren sind extrem wichtig, wenn das Objekt Ressourcen enthält, die im Destruktor an einen Ressourcen-Pool zurückgegeben werden.

Obwohl die Prinzipien der objektorientierten Programmierung erläutert wurden, sind wir noch nicht da, wo wir hin wollen. Wir haben ein Mikrowelle gebaut, wir haben eine Box um die arbeitenden Teile gebaut, und wir haben ein Benutzerinterface definiert; aber wir haben keine Beziehung zwischen Mikrowellen und anderen Ofentypen hergestellt. Das ist das Wesentliche der objektorientierten Programmierung, und wird in Teil 5 diskutiert.

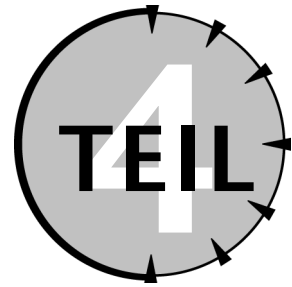
236

Samstagabend

Selbsttest

1. Warum sollte eine Klasse wie `Student` einen Konstruktor der Form `Student(char* pszName, int nId)` haben? (Siehe »Konstruktoren mit Argumenten«)
2. Wie können Sie etwas über die Reihenfolge aussagen, in der globale Objekte konstruiert werden? (Siehe »Keine bestimmte Reihenfolge für globale Objekte«)
3. Warum benötigt eine Klasse einen Kopierkonstruktor? (Siehe »Der Kopierkonstruktor«)
4. Was ist der Unterschied zwischen einer flachen Kopie und einer tiefen Kopie? (Siehe »Flache Kopie gegen tiefe Kopie«)

Samstagabend – Zusammenfassung



1. *Denken Sie über den Inhalt Ihres Kleiderschranks nach. Beschreiben Sie informell, was Sie dort finden.*
2. *Betrachten Sie Schuhe. Beschreiben Sie das Interface von Schuhen.*
3. *Benennen Sie zwei verschiedenen Typen von Schuhen. Was ist der Effekt, einen anstelle des anderen zu verwenden?*
4. *Schreiben Sie einen Konstruktor für eine Klasse, die wie folgt definiert ist:*

```
class Link
{
    static Link* pHead;
    Link* pNextLink;
};
```

5. *Schreiben Sie einen Kopierkonstruktor und einen Destruktor für die folgende Klasse*
LinkedList:

```
#include <stdio.h>
#include <iostream.h>
#include <string.h>

class LinkedList
{
protected:
    LinkedList* pNext;
    static LinkedList* pHead;
    char* pszName;

public:
    // Konstruktor - kopiere den Namen
    LinkedList(char* pszName)
    {
        int nLength = strlen(pszName) + 1;
        this->pszName = new char[nLength];
        strcpy(this->pszName, pszName);
        pNext = 0;
    }

    // diese Elementfunktionen soll es geben
    void addToList();
    void removeFromList();
};
```

```
// Destruktor -  
~LinkedList()  
{  
    // ... was kommt hier hin?  
}  
LinkedList(LinkedList& l)  
{  
    // ... und hier?  
}  
};
```

Hinweise:

- a. Bedenken Sie, was passiert, wenn das Objekt immer noch in der verketteten Liste ist, nachdem es vernichtet wurde.**
- b. Erinnern Sie sich daran, dass Speicher, der vom Heap alloziert wurde, zurückgegeben werden sollte, bevor der Zeiger darauf verloren geht.**

☐ Freitag

☐ Samstag

☒ **Sonntag**

Sonntagmorgen

Teil 5

Lektion 21

Vererbung

Lektion 22

Polymorphie

Lektion 23

Abstrakte Klassen und Faktorisieren

Lektion 24

Mehrfachvererbung

Lektion 25

Große Programme II

Lektion 26

C++-Präprozessor II

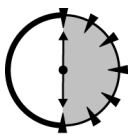


Vererbung



Checkliste

- ☒ Vererbung definieren
- ☒ Von einer Basisklasse erben
- ☒ Die Basisklasse konstruieren
- ☒ die Beziehungen IS_A und HAS_A vergleichen



30 Min.

In dieser Sitzung diskutieren wir Vererbung. *Vererbung* ist die Fähigkeit einer Klasse, auf Fähigkeiten oder Eigenschaften einer anderen Klasse zurückzugreifen. Ich z.B. bin ein Mensch. Ich erbe von der Klasse *Mensch* bestimmte Eigenschaften, wie z.B. meine Fähigkeit zu (mehr oder weniger) intelligenter Konversation, und meine Abhängigkeit von Luft, Wasser und Nahrung. Diese Eigenschaften sind nicht einzigartig für Menschen. Die Klasse *Mensch* erbt diese Abhängigkeit von Luft, Wasser und Nahrung von der Klasse *Säugetier*.

21.1 Vorteile von Vererbung

Die Fähigkeit, Eigenschaften nach unten weiterzugeben, ist ein mächtige. Sie erlaubt es uns, Dinge auf ökonomische Art und Weise zu beschreiben. Wenn z.B. mein Sohn fragt »Was ist eine Ente?« kann ich sagen »Es ist ein Vogel, der Quakquak macht.« Was immer Sie über diese Antwort denken mögen, übermittelt sie ihm einiges an Informationen. Er weiß, was ein Vogel ist, und jetzt weiß er all diese Dinge für Enten, plus die zusätzliche Quakquak-Eigenschaft.

Es gibt mehrere Gründe, weshalb Vererbung in C++ eingeführt wurde. Sicherlich ist der wichtigste Grund die Möglichkeit, Vererbungsbeziehungen auszudrücken. (Ich werde darauf gleich zurückkommen.) Ein weniger wichtiger Grund ist der, den Schreibaufwand zu reduzieren. Nehmen Sie an, Sie haben eine Klasse *Student*, und wir sollen eine neue Klasse *GraduateStudent* hinzufügen. Vererbung kann die Anzahl der Dinge, die wir in eine solche Klasse packen müssen, drastisch reduzieren. Alles, was wir in der Klasse *GraduateStudent* wirklich brauchen, sind die Dinge, die den Unterschied zwischen Studenten und graduierten Studenten beschreiben.

242 Sonntagmorgen

Wichtiger ist das verwandte Reizwort der 90-er Jahre, Wiederverwendung. Softwarewissenschaftler haben vor einer gewissen Zeit festgestellt, dass es keinen Sinn macht, in jedem Softwareprojekt bei Null zu beginnen, und die gleichen Softwarekomponenten immer wieder neu zu schreiben.

Vergleichen Sie diese Situation bei der Software mit anderen Industrien. Wie viele Autohersteller fangen bei jedem Auto ganz von vorne an? Und selbst wenn sie das täten, wie viele würden beim nächsten Modell wieder ganz von vorne beginnen? Praktiker in anderen Industrien haben es sinnvoller gefunden, bei Schrauben, Muttern und auch größeren Komponenten wie Motoren und Kompressoren zu beginnen.

Unglücklicherweise ist, mit Ausnahme der sehr kleinen Funktionen in der Standardbibliothek von C, nur sehr wenig Wiederverwendung von Softwarekomponenten zu sehen. Ein Problem ist, dass es fast unmöglich ist, eine Komponente in einem früheren Programm zu finden, die exakt das tut, was Sie brauchen. Im Allgemeinen müssen diese Komponenten angepasst werden.

Es gibt eine Daumenregel die besagt »Wenn Sie es geöffnet haben, haben Sie es zerbrochen«. Mit anderen Worten, wenn Sie eine Funktion oder Klasse modifizieren müssen, um sie an Ihre Anwendung anzupassen, müssen Sie wieder alles neu testen, nicht nur die Teile, die Sie hinzugefügt haben. Änderungen können irgendwo im Code Bugs verursachen. (»Wer den Code zuletzt angefasst hat, muss den Bug fixen«.)

Vererbung ermöglicht es, bestehende Klassen an neue Anwendungen anzupassen ohne sie verändern zu müssen. Von der bestehenden Klasse wird eine neue Unterklasse abgeleitet, die alle nötigen Zusätze und Änderungen enthält.

Das bringt einen dritten Vorteil. Nehmen Sie an, wir erben von einer existierenden Klasse. Später finden wir heraus, dass die Basisklasse einen Fehler enthält und korrigiert werden muss. Wenn wir die Klasse zur Wiederverwendung modifiziert haben, müssen wir in jeder Anwendung einzeln auf den Fehler testen und ihn korrigieren. Wenn wir von der Klasse ohne Änderungen geerbt haben, können wir die berichtigte Klasse sicher ohne Weiteres übernehmen.

21.2 Faktorieren von Klassen

Um unsere Umgebung zu verstehen, haben die Menschen umfangreiche Begrifflichkeiten eingeführt. Unser *Fido* ist ein Spezialfall von *Rüde*, was ein Spezialfall von *Hund* ist, was ein Spezialfall von *Säugetier* ist usw. Das formt unser Verständnis unserer Welt.

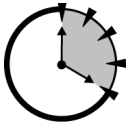
Um ein anderes Beispiel zu gebrauchen, ist ein Student ein spezieller Typ Person. Wenn ich das gesagt habe, weiß ich bereits viele Dinge über Studenten. Ich weiß, dass Sie eine Sozialversicherungsnummer haben, dass Sie zu viel fernsehen, dass sie zu schnell fahren, und nicht genug üben. Ich weiß all diese Dinge, weil es Eigenschaften aller Leute sind.

In C++ bezeichnen wir das als *Vererbung*. Wir sagen, dass die Klasse `Student` von der Klasse `Person` *erbt*. Wir sagen auch, dass die Klasse `Person` die *Basisklasse* von `Student` ist und `Student` eine *Unterklasse* von `Person` ist. Schließlich sagen wir, dass ein `Student` *IS_A* `Person` (ich verwende die Großbuchstaben als meine Art, um diese eindeutige Beziehung zu bezeichnen). C++ teilt diese Terminologie mit anderen objektorientierten Sprachen.

Beachten Sie, dass obwohl `Student` *IS_A* `Person` wahr ist, das Gegenteil nicht der Fall ist. (Eine Aussage wie diese bezieht sich immer auf den allgemeinen Fall. Es kann sein, dass eine bestimmte `Person` ein `Student` ist.) Eine Menge Leute, die zur Klasse `Person` gehören, gehören nicht zur Klasse `Student`. Das liegt daran, dass die Klasse `Student` Eigenschaften besitzt, die sie mit der Klasse `Person` nicht teilt. Z.B. hat `Student` einen mittleren Grad, aber `Person` hat das nicht.

Lektion 21 – Vererbung**243**

Die Vererbungsbeziehung ist jedoch transitiv. Wenn ich z.B. eine neue Klasse GraduateStudent als Unterklasse von Student einführe, muss GraduateStudent auch Person sein. Es muss so aussehen: wenn GraduateStudent IS_A Student und Student IS_A Person, dann GraduateStudent IS_A Person.

**20 Min.****21.3 Implementierung von Vererbung in C++**

Um zu demonstrieren, wie Vererbung in C++ ausgedrückt wird, lassen Sie uns zu dem Beispiel GraduateStudent zurückkehren und dieses mit einigen exemplarischen Elementen ausstatten:

```
// GSinherit - demonstriert, wie GraduateStudent von
//           Student die Eigenschaften eines
//           Studenten erben kann
#include <stdio.h>
#include <iostream.h>
#include <string.h>

// Advisor - nur eine Beispielklasse
class Advisor
{
};

// Student - alle Informationen über Studenten
class Student
{
public:
    Student()
    {
        // initialer Zustand
        pszName = 0;
        nSemesterHours = 0;
        dAverage = 0;
    }
    ~Student()
    {
        // wenn es einen Namen gibt ...
        if (pszName != 0)
        {
            // ... dann gib den Puffer zurück
            delete pszName;
            pszName = 0;
        }
    }

    // addCourse - fügt den Effekt eines absolvierten
    //           Kurses mit dGrade zu dAverage
    //           hinzu
    void addCourse(int nHours, double dGrade)
    {
        // aktuellen gewichteten Mittelwert
```

244 **Sonntagmorgen**

```

        int ndGradeHours = (int)(nSemesterHours * dAverage + dGrade);

        // beziehe die absolvierten Stunden ein
        nSemesterHours += nHours;

        // berechne neuen Mittelwert
        dAverage = ndGradeHours / nSemesterHours;
    }

    // die folgenden Zugriffsfunktionen geben
    // der Anwendung Zugriff auf wichtige
    // Eigenschaften
    int hours( )
    {
        return nSemesterHours;
    }
    double average( )
    {
        return dAverage;
    }

protected:
    char* pszName;
    int nSemesterHours;
    double dAverage;

    // Kopierkonstruktor - ich will nicht, dass
    // Kopien erzeugt werden
    Student(Student& s)
    {
    }
};

// GraduateStudent - diese Klasse ist auf die
// Studenten beschränkt, die ihr
// Vordiplom haben
class GraduateStudent : public Student
{
public:
    GraduateStudent()
    {
        dQualifierGrade = 2.0;
    }

    double qualifier( )
    {
        return dQualifierGrade;
    }

protected:
    // alle graduierten Studenten haben einen Advisor
    Advisor advisor;

    // das ist der Grad, unter dem ein
    // GraduateStudent den Kurs nicht
    // erfolgreich absolviert hat

```



```

double dQualifierGrade;
};

int main(int nArgc, char* pszArgs[])
{
    // erzeuge einen Studenten
    Student llu;

    // und jetzt einen graduierten Studenten
    GraduateStudent gs;

    // das Folgende ist völlig korrekt
    llu.addCourse(3, 2.5);
    gs.addCourse(3, 3.0);

    // das Folgende aber nicht
    gs.qualifier();    // das ist gültig
    llu.qualifier();   // das aber nicht
    return 0;
}

```

Die Klasse `Student` wurde in der gewohnten Weise deklariert. Die Deklaration der Klasse `GraduateStudent` unterscheidet sich davon. Der Name der Klasse, gefolgt von dem Doppelpunkt, gefolgt von `public Student` deklariert die Klasse `GraduateStudent` als Unterklasse von `Student`.



Das Schlüsselwort `public` impliziert, dass es sicherlich auch eine `protected`-Vererbung gibt. Das ist der Fall, ich möchte jedoch diesen Typ Vererbung für den Moment aus der Diskussion heraus lassen.

Die Funktion `main( )` deklariert zwei Objekte, `llu` und `gs`. Das Objekt `llu` ist ein konventionelles `Student`-Objekt, aber das Objekt `gs` ist etwas Neues. Als ein Mitglied einer Unterklasse von `Student`, kann `gs` alles tun, was `llu` tun kann. Es hat die Datenelemente `pszName`, `nSemesterHours` und `dAverage` und die Elementfunktion `addCourse( )`. Buchstäblich gilt, `gs` IS_A `Student` – `gs` ist nur ein wenig mehr als ein `Student`. (Sie werden es am Ende des Buches sicher nicht mehr ertragen können, dass ich »IS_A« so oft benutze.) In der Tat hat die Klasse `GraduateStudent` die Eigenschaft `qualifier( )`, die `Student` nicht besitzt.

Die nächsten beiden Zeilen fügen den beiden Studenten `llu` und `gs` einen Kurs hinzu. Erinnern Sie sich daran, dass `gs` auch ein `Student` ist.

Eine der letzten Zeilen in `main( )` ist nicht korrekt. Es ist in Ordnung die Methode `qualifier( )` für das Objekt `gs` aufzurufen. Es ist nicht in Ordnung, die Eigenschaft `qualifier` für das Objekt `llu` zu verwenden. Das Objekt `llu` ist nur ein `Student` und hat nicht die Eigenschaften, die für `GraduateStudent` einzigartig sind.

246 **Sonntagmorgen**

Betrachten Sie das folgende Szenario:

```
// fn - führt eine Operation auf Student aus
void fn(Student &s)
{
    // was immer fn tun möchte
}

int main(int nArgc, char* pszArgs[])
{
    // erzeuge einen graduierten Studenten ...
    GraduateStudent gs;

    // ... übergib ihn als einfachen Studenten
    fn(gs);
    return 0;
}
```

Beachten Sie, dass die Funktion `fn( )` ein Objekt vom Typ `Student` als Argument erwartet. Der Aufruf von `main( )` übergibt der Funktion ein Objekt aus der Klasse `GraduateStudent`. Das ist in Ordnung, weil (um es noch einmal zu wiederholen) »ein `GraduateStudent` IS_A `Student`.«

Im Wesentlichen entstehen die gleichen Bedingungen, wenn eine Elementfunktion von `Student` mit einem `GraduateStudent`-Objekt aufgerufen wird. Z.B.:

```
int main(int nArgc, char* pszArgs[])
{
    GraduateStudent gs;
    gs.addCourse(3, 2.5); // ruft Student::addCourse( )
    return 0;
}
```

21.4 Unterklassen konstruieren

Obwohl eine Unterklasse Zugriff hat auf die `protected`-Elemente der Basisklasse und diese in ihrem eigenen Konstruktor initialisieren kann, möchten wir gerne, dass sich die Basisklasse selber konstruiert. Das ist in der Tat, was passiert. Bevor die Kontrolle über die öffnende Klammer des Konstruktors von hinwegkommt, geht sie zuerst auf die Defaultkonstruktor von `Student` über (weil kein anderer Konstruktor angegeben wurde). Wenn `Student` auf einer weiteren Klasse basieren würde, wie z.B. `Person`, würde der Konstruktor dieser Klasse aufgerufen, bevor der Konstruktor von `Student` die Kontrolle bekommt. Wie ein Wolkenkratzer wird ein Objekt von seinem Fundament die Klassenstruktur aufwärts aufgebaut.

Wie mit Elementobjekten, ist es manchmal nötig, Argumente an den Konstruktor der Basisklasse zu übergeben. Wir tun dies in fast der gleichen Weise, wie bei den Elementobjekten, wie das folgende Beispiel zeigt:

```
// Student - diese Klasse enthält alle Typen
//          von Studenten
class Student
{
public:
    // Konstruktor - definiere Defaultargument,
```

```

//          um auch einen Defaultkonstruktor
//          zu haben
Student(char* pszName = 0)
{
    // initialer Zustand
    this->pszName = 0;
    nSemesterHours = 0;
    dAverage = 0.0;

    // wenn es einen Namen gibt ...
    if (pszName != 0)
    {
        this->pszName =
            new char[strlen(pszName) + 1];
        strcpy(this->pszName, pszName);
    }
}
~Student()
{
    // wenn es einen Namen gibt ...
    if (pszName != 0)
    {
        // ... dann gib den Puffer zurück
        delete pszName;
        pszName = 0;
    }
}
// ... Rest der Klassendefinition ...
};

// GraduateStudent - diese Klasse ist auf die
//                  Studenten beschränkt, die ihr
//                  Vordiplom haben
class GraduateStudent : public Student
{
public:
    // Konstruktor - erzeuge graduierten Studenten
    //              mit einem Advisor, einem Namen
    //              und einem Qualifizierungsgrad
    GraduateStudent(
        Advisor &adv,
        char*   pszName = 0,
        double  dQualifierGrade = 0.0)
        : Student(pName),
          advisor(adv)
    {
        // wird erst ausgeführt, nachdem die anderen
        // Konstruktoren aufgerufen wurden
        dQualifierGrade = 0;
    }
protected:
    // alle graduierten Studenten haben einen Advisor
    Advisor advisor;

    // das ist der Grad, unter dem ein
    // GraduateStudent den Kurs nicht

```

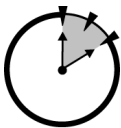
248 **Sonntagmorgen**

```
// erfolgreich absolviert hat
double dQualifierGrade;
};
void fn(Advisor &advisor)
{
    // graduierten Studenten erzeugen
    GraduateStudent gs(»Marion Haste«,
                      advisor,
                      2.0);
    //... was immer diese Funktion tut ...
}
```

Hier wird ein `GraduateStudent`-Objekt mit einem `Advisor` erzeugt, dessen Name »Marion Haste« und dessen Grad gleich 2.0 ist. Der Konstruktor von `GraduateStudent` ruft den Konstruktor `Student` auf, und übergibt den Namen des Studenten. Die Basisklasse wird konstruiert vor allen anderen Elementobjekten; somit wird der Konstruktor von `Student` vor den Konstruktor von `Advisor` aufgerufen. Nachdem die Basisklasse konstruiert wurde, wird das `Advisor`-Objekt `advisor` mittels Kopierkonstruktor konstruiert. Erst dann kommt der Konstruktor von `GraduateStudent` zum Zuge.



Die Tatsache, dass die Basisklasse zuerst erzeugt wird, hat nichts mit der Ordnung der Konstruktoranweisungen hinter dem Doppelpunkt zu tun. Die Basisklasse wäre auch dann vor den Datenelementen konstruiert worden, wenn die Anweisungen `advisor(adv), Student(pszName)` gelautet hätten. Es ist jedenfalls eine gute Idee, diese Klauseln in der Reihenfolge zu schreiben, in der sie ausgeführt werden, nur um niemanden zu verwirren.



10 Min.

Gemäß unserer Regel, dass Destruktoren in der umgekehrten Reihenfolge aufgerufen werden wie die Konstrukturen, bekommt der Destruktor von `GraduateStudent` zuerst die Kontrolle. Nachdem er seine letzten Dienste erbracht hat, geht die Kontrolle auf den Destruktor von `Advisor` und dann auf den Destruktor von `Student` über. Wenn `Student` von einer Klasse `Person` abgeleitet wäre, ginge die Kontrolle auf den Destruktor von `Person` nach `Student` über.



Der Destruktor der Basisklasse `Student` wird ausgeführt, obwohl es keinen expliziten Destruktor `~GraduateStudent` gibt.

Das ist logisch. Der wenige Speicher, der schließlich zu einem `GraduateStudent`-Objekt wird, wird erst in ein `Student`-Objekt konvertiert. Dann ist es die Aufgabe des Konstruktors `GraduateStudent`, seine Transformation in ein `GraduateStudent`-Objekt zu vervollständigen. Der Destruktor kehrt diesen Prozess einfach um.



Beachten Sie einige wenige Dinge in diesem Beispiel. Erstens wurden Defaultargumente im Konstruktor `GraduateStudent` bereitgestellt, um diese Fähigkeit an die Basisklasse `Student` weiterzugeben. Zweitens können Defaultwerte für Argumente nur von rechts nach links angegeben werden. Das Folgende ist nicht möglich:
`GraduateStudent(char* pszName = 0, Advisor& adv) ...`
Die Argumente ohne Defaultwerte müssen zuerst kommen.

Beachten Sie, dass die Klasse `GraduateStudent` ein `Advisor`-Objekt in der Klasse enthält. Es enthält keinen Zeiger auf ein `Advisor`-Objekt. Letzteres würde so geschrieben werden:

```
class GraduateStudent : public Student
{
public:
    GraduateStudent(
        Advisor& adv,
        char* pszName = 0)
        : Student(pName),
        {
            pAdvisor = new Advisor(adv);
        }
protected:
    Advisor* pAdvisor;
};
```

Hierbei wird die Basisklasse `Student` zuerst erzeugt (wie immer). Der Zeiger wird innerhalb des Body des Konstruktors `GraduateStudent` initialisiert.

21.5 Die Beziehung HAS_A

Beachten Sie, dass die Klasse `GraduateStudent` die Elemente der Klasse `Student` und `Advisor` einschließt, aber auf verschiedene Weisen. Durch die Definition eines Datenelementes aus der Klasse `Advisor` wissen wir, dass ein `GraduateStudent` alle Datenelemente von `Advisor` in sich enthält, und wir drücken das aus, indem wir sagen, `GraduateStudent HAS_A Advisor`. Was ist der Unterschied zwischen dieser Beziehung und Vererbung?

Lassen Sie uns ein Auto als Beispiel nehmen. Wir könnten logisch ein Auto als Unterklasse von *Fahrzeug* definieren und dadurch allgemeine Eigenschaften von Fahrzeugen erben. Gleichzeitig hat ein Auto auch einen Motor. Wenn Sie ein Auto kaufen, können Sie logisch davon ausgehen, dass Sie auch einen Motor kaufen.

Wenn nun einige Freunde am Wochenende eine Rallye mit dem Fahrzeug der eigenen Wahl veranstalten, wird sich niemand darüber beschweren, wenn Sie mit ihrem Auto kommen, weil Auto IS_A Fahrzeug. Wenn Sie aber zu Fuß kommen und Ihren Motor unter dem Arm tragen, haben sie allen Grund, erstaunt zu sein, weil ein Motor kein Fahrzeug ist. Ihm fehlen einige wesentliche Eigenschaften, die alle Fahrzeuge haben. Es fehlen dem Motor sogar Eigenschaften, die alle Autos haben.

250 **Sonntagmorgen**

Vom Standpunkt der Programmierung aus ist es ebenso einfach. Betrachten sie das Folgende:

```
class Vehicle
{
};
class Motor
{
};
class Car : public Vehicle
{
public:
    Motor motor;
};
void VehicleFn(Vehicle &v);
void motorFn(Motor &m);
int main(int nArgc, char* pszArgs[])
{
    Car c;
    vehicleFn(c);    // das ist erlaubt
    motorFn(c);      // das ist nicht erlaubt
    motorFn(c.motor); // das jedoch schon
    return 0;
}
```



0 Min.

Der Aufruf `vehicleFn(c)` ist erlaubt, weil `c` IS_A `Vehicle`. Der Aufruf `motorFn(c)` ist nicht erlaubt, weil `c` kein `Motor` ist, obwohl es einen `Motor` enthält. Wenn beabsichtigt ist, den Teil `Motor` von `c` an eine Funktion zu übergeben, muss dies explizit ausgedrückt werden, wie im Aufruf `motorFn(c.motor)`.



Natürlich ist der Aufruf `motorFn(c.motor)` nur dann erlaubt, wenn `c.motor` *public* ist.

Ein weiterer Unterschied: Die Klasse `Car` hat Zugriff auf die `protected`-Elemente von `Vehicle`, aber nicht auf die `protected`-Elemente von `Motor`.

Zusammenfassung

Das Verständnis von Vererbung ist wesentlich für das Gesamtverständnis der objektorientierten Programmierung. Es wird auch benötigt, um das nächste Kapitel verstehen zu können. Wenn Sie den Eindruck haben, dass sie es verstanden haben, gehen Sie weiter zu Kapitel 22. Wenn nicht, lesen Sie dieses Kapitel erneut.

Selbsttest

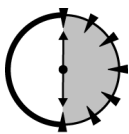
1. Was ist die Beziehung zwischen einem graduierten Studenten und einem Studenten. Ist es eine Beziehung der Form IS_A oder HAS_A? (Siehe »Die Beziehung HAS_A«)
2. Nennen Sie drei Vorteile davon, dass Vererbung in der Programmiersprache C++ vorhanden ist. (Siehe »Vorteile von Vererbung«)
3. Welcher der folgenden Begriffe passt nicht? *Erbt*, *Unterklasse*, *Datenelement* und *IS_A*? (Siehe »Faktorisieren von Klassen«)



Polymorphie

Checkliste

- ☒ Elementfunktionen in Unterklassen überschreiben
- ☒ Polymorphie anwenden (alias späte Bindung)
- ☒ Polymorphie mit früher Bindung vergleichen
- ☒ Polymorphie speziell betrachten



30 Min.

Verbung gibt uns die Möglichkeit, eine Klasse mit Hilfe einer anderen Klasse zu beschreiben. Genauso wichtig ist, dass dadurch die Beziehung zwischen den Klassen deutlich wird. Nochmals, eine Mikrowelle ist ein Typ Ofen. Es fehlt jedoch noch ein Teil im Puzzle.

Sie haben das bestimmt bereits bemerkt, aber eine Mikrowelle und ein herkömmlicher Ofen sehen sich nicht besonders ähnlich. Die beiden Ofentypen arbeiten auch nicht gleich. Trotzdem möchte ich mir keine Gedanken darüber machen, wie jeder einzelne Ofen das »Kochen« ausführt. Diese Sitzung beschreibt, wie C++ dieses Problem behandelt.

22.1 Elementfunktionen überschreiben

Es war immer möglich, eine Elementfunktion in einer Klasse mit einer Elementfunktion in der gleichen Klasse zu überschreiben, solange die Argumente verschieden sind. Es ist auch möglich, ein Element einer Klasse mit einer Elementfunktion einer anderen Klasse zu überschreiben, selbst wenn die Argumente gleich sind.



Vererbung liefert eine weitere Möglichkeit: Eine Elementfunktion in einer Unterklasse kann eine Elementfunktion der Basisklasse überladen.

Betrachten Sie z.B. das einfache Programm `EarlyBinding` in Listing 22-1.

Listing 22-1: Beispielprogramm `EarlyBinding`

```
// EarlyBinding - Aufrufe von überschriebenen
//               Elementfunktionen werden anhand
//               des Objekttyps aufgelöst
#include <stdio.h>
#include <iostream.h>

class Student
{
public:
    // berechnet das Schulgeld
    double calcTuition()
    {
        return 0;
    }
};
class GraduateStudent : public Student
{
public:
    double calcTuition()
    {
        return 1;
    }
};

int main(int nArgc, char* pszArgs[])
{
    // der folgende Ausdruck ruft
    // Student::calcTuition();
    Student s;
    cout << »Der Wert von s.calcTuition ist »
         << s.calcTuition()
         << »\n«;

    // das ruft GraduateStudent::calcTuition();
    GraduateStudent gs;
    cout << »Der Wert von gs.calcTuition ist »
         << gs.calcTuition()
         << »\n«;
    return 0;
}
```

Ausgabe

```
Der Wert von s.calcTuition ist 0
Der Wert von gs.calcTuition ist 1
```

Wie bei jedem anderen Fall von Überschreiben, muss C++ entscheiden, welche Funktion `calcTuition()` gemeint ist, wenn der Programmierer `calcTuition()` aufruft. Normalerweise reicht die Klasse aus, um den Aufruf aufzulösen, und es ist bei diesem Beispiel nicht anders. Der Aufruf `s.calcTuition()` bezieht sich auf `Student::calcTuition()`, weil `s` als `Student` deklariert ist, wobei `gs.calcTuition()` sich auf `GraduateStudent::calcTuition()` bezieht.

Die Ausgabe des Programms `EarlyBinding` zeigt, dass der Aufruf überschriebener Elementfunktionen gemäß dem Typ des Objektes aufgelöst wird.



Das Auflösen von Aufrufen von Elementfunktionen basierend auf dem Typ des Objektes wird Bindung zur Compilezeit oder auch frühe Bindung genannt.

22.2 Einstieg in Polymorphie

Überschreiben von Funktionen basierend auf der Klasse von Objekten ist schon sehr schön, aber was ist, wenn die Klasse des Objektes, das eine Methode aufruft, zur Compilezeit nicht eindeutig bestimmt werden kann? Um zu demonstrieren, wie das passieren kann, lassen Sie uns das vorangegangene Programm auf eine scheinbar triviale Weise ändern. Das Ergebnis ist das Programm `AmbiguousBinding`, das Sie in Listing 22-2 finden.

Listing 22-2: Programm `AmbiguousBinding`

```
// AmbiguousBinding - die Situation wird verwirrend
//                      wenn der Typ zur Compilezeit
//                      nicht gleich dem Typ
//                      zur Laufzeit ist
#include <stdio.h>
#include <iostream.h>

class Student
{
public:
    double calcTuition()
    {
        return 0;
    }
};
class GraduateStudent : public Student
{
public:
    double calcTuition()
    {
        return 1;
    }
};

double fn(Student& fs)
{
    // auf welche Funktion calcTuition() bezieht
    // sich der Aufruf? Welcher Wert wird
    // zurückgegeben?
    return fs.calcTuition();
}

int main(int nArgc, char* pszArgs[])
```

```

{
    // der folgende Ausdruck ruft
    // Student::calcTuition();
    Student s;
    cout << »Der Wert von s.calcTuition bei\n«
         << »Aufruf durch fn() ist »
         << fn(s)
         << »\n«;

    // das ruft GraduateStudent::calcTuition();
    GraduateStudent gs;
    cout << »Der Wert von gs.calcTuition bei\n«
         << »Aufruf durch fn() ist »
         << fn(gs)
         << »\n«;
    return 0;
}

```

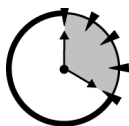
Der einzige Unterschied zwischen Listing 22-1 und 22-2 ist, dass die Aufrufe von `calcTuition()` über eine Zwischenfunktion `fn()` ausgeführt werden. Die Funktion `fn(Student& fs)` ist so deklariert, dass sie ein `Student`-Objekt übergeben bekommt, aber abhängig davon, wie `fn()` aufgerufen wird, kann `fs` ein `Student` oder ein `GraduateStudent` sein. (Erinnern Sie sich? `GraduateStudent` ist ein `Student`.) Aber diese beiden Typen von Objekten berechnen ihr Schulgeld verschieden.

Weder `main()` noch `fn()` kümmern sich eigentlich darum, wie das Schulgeld berechnet wird. Wir hätten gerne, dass `fs.calcTuition()` die Funktion `Student::calcTuition()` aufruft, wenn `fs` ein `Student` ist, aber `GraduateStudent::calcTuition()`, wenn `fs` ein `GraduateStudent` ist. Aber diese Entscheidung kann erst zur Laufzeit getroffen werden, wenn der tatsächliche Typ des übergebenen Objektes bestimmt werden kann.

Im Falle des Programms `AmbiguousBindung` sagen wir, dass der Compiletyp von `fs`, der immer `Student` ist, verschieden ist vom Laufzeittyp, der `GraduateStudent` oder `Student` sein kann.



*Die Fähigkeit zu entscheiden, welche der mehrfach überladenen Elementfunktionen aufgerufen werden soll, basierend auf dem Laufzeittyp, wird als **Polymorphie** oder **späte Bindung** bezeichnet. Polymorphie kommt vom *poly* (=viel) und *morph* (=Form).*



20 Min.

22.3 Polymorphie und objektorientierte Programmierung

Polymorphie ist der Schlüssel zur objektorientierten Programmierung. Sie ist so wichtig, dass Sprachen, die keine Polymorphie unterstützen, sich nicht objektorientiert nennen dürfen. Sprachen, die Klassen, aber keine Polymorphie unterstützen, werden als *objektbasierte Sprachen* bezeichnet. Ada ist ein Beispiel einer solchen Sprache.

Ohne Polymorphie hat Vererbung keine Bedeutung.

256 **Sonntagmorgen**

Erinnern Sie sich, wie ich Nachos im Ofen hergestellt habe? In diesem Sinne habe ich als später Binder gearbeitet. Im Rezept steht: »Nachos im Ofen erwärmen.« Da steht nicht: »Wenn der Ofen eine Mikrowelle ist, tun Sie das; wenn es ein herkömmlicher Ofen ist, tun sie das; wenn der Ofen ein Elektroofen ist, tun sie noch etwas anderes.« Das Rezept (der Code) verlässt sich auf mich (den späten Binder) zu entscheiden, welche Tätigkeit (Elementfunktion) `erwärmen` bedeutet, angewendet auf einen Ofen (die spezielle Instanz von `Oven`) oder eine ihrer Varianten (Unterklassen), wie z.B. Mikrowellen (`Microwave`). Das ist die Art und Weise, in der Leute denken, und eine Sprache in dieser Weise zu entwickeln, ermöglicht es der Sprache, besser zu beschreiben, was Leute denken.

Es gibt da noch die beiden Aspekte der Pflege und Wiederverwendbarkeit. Nehmen Sie an, ich habe dieses großartige Programm beschrieben, das die Klasse `Student` verwendet. Nach einigen Monaten des Entwurfs, der Implementierung und des Testens erstelle ich ein Release der Anwendung.

Es vergeht einige Zeit und mein Chef bittet mich, dem Programm `GraduateStudent`-Objekte hinzuzufügen, die sehr ähnlich zu Studenten sind, aber nicht identisch damit. Tief im Programm ruft die Funktion `someFunktion()` die Elementfunktion `calcTuition()` wie folgt auf:

```
void someFunktion(Student &s)
{
    //... was immer sie tut ...
    s.calcTuition();
    //... wird hier fortgesetzt ...
}
```

Wenn C++ keine späte Bindung ausführen würde, müsste ich die Funktion `someFunktion()` editieren, um auch `GraduateStudent`-Objekte verarbeiten zu können. Das könnte etwa so aussehen:

```
#define STUDENT 1
#define GRADUATESTUDENT 2
void someFunktion(Student &s)
{
    //... was immer sie tut ...
    // füge ein Typelement hinzu, das den
    // tatsächlichen Typ des Objekts angibt
    switch (s.type)
    {
        STUDENT:
            s.Student::calcTuition();
            break;
        GRADUATESTUDENT:
            s.GraduateStudent::calcTuition();
            break;
    }
    //... alles Weitere hier ...
}
```



Durch Verwendung des vollständigen Namens der Funktion zwingt der Ausdruck `s.GraduateStudent::calcTuition()` den Aufruf, die `GraduateStudent`-Version der Funktion zu verwenden, selbst wenn `s` als `Student` deklariert ist.

Ich würde dann ein `Element` type in der Klasse einführen, das ich im Konstruktor von `Student` auf `STUDENT` setzen würde, und auf `GRADUATESTUDENT` im Konstruktor von `GraduateStudent`. Der Wert von `type` würde den Laufzeittyp von `s` darstellen. Ich würde dann den Test im Codeschnipsel einfügen, um die dem Wert dieses Elements entsprechende Funktion aufzurufen.

Das hört sich nicht schlecht an, mit Ausnahme von drei Dingen. Erstens ist das hier nur eine Funktion. Nehmen Sie an, dass `calcTuition()` von vielen Stellen aus aufgerufen wird, und nehmen Sie an, dass `calcTuition()` nicht der einzige Unterschied der beiden Klassen ist. Die Chancen stehen nicht sehr gut, dass ich alle Stellen finde, an denen ich etwas ändern muss.

Zweitens muss ich Code, der bereits fertiggestellt wurde, editieren (d.h. »brechen«), wodurch die Möglichkeit für neue Fehler gegeben ist. Das Editieren kann zeitaufwendig und langweilig sein, was wiederum die Gefahr von Fehlern erhöht. Irgendeine meiner Änderungen kann falsch sein oder nicht in den existierenden Code passen. Wer weiß das schon?

Schließlich, nachdem das Editieren, das erneute Debuggen und Testen abgeschlossen sind, muss ich zwei Versionen unterstützen (wenn ich nicht die Unterstützung für die Originalversion aufgeben kann). Das bedeutet zwei Quellen, die editiert werden müssen, wenn Bugs gefunden werden, und eine Art Buchhaltung, um die beiden Systeme gleich zu halten.

Was passiert, wenn mein Chef eine weitere Klasse eingefügt haben möchte? (Mein Chef ist so.) Ich muss nicht nur diesen Prozess wiederholen, ich habe dann auch drei Versionen.

Mit Polymorphie habe ich eine gute Chance, dass ich nur die neue Klasse einfügen und neu kompilieren muss. Es kann sein, dass ich die Basisklasse selber ändern muss, das ist aber wenigstens alles an einer Stelle. Änderungen an der Anwendung sollten wenige bis keine sein.

Das ist noch ein weiterer Grund, Datenelemente `protected` zu halten, und auf sie über als `public` deklarierte Elementfunktionen zuzugreifen. Datenelemente können nicht durch Polymorphie in einer Unterklasse überschrieben werden, so wie es für Elementfunktionen möglich ist.

22.4 Wie funktioniert Polymorphie?

Nach allem, was ich bisher gesagt habe, kann es verwundern, dass in C++ die frühe Bindung die Defaultmethode ist. Die Ausgabe des Programms `AmbiguousBinding` sieht wie folgt aus:

```
Der Wert von s.calcTuition bei
Aufruf durch fn() ist 0

Der Wert von gs.calcTuition bei
Aufruf durch fn() ist 0
```

Der Grund ist einfach. Polymorphie bedeutet ein wenig Mehraufwand, sowohl beim Speicherbedarf und beim Code, der den Aufruf ausführt. Die Erfinder von C++ waren in Sorge darüber, dass ein solcher Mehraufwand ein Grund sein könnte, die Programmiersprache C++ nicht zu verwenden, und so machten sie die frühe Bindung zur Defaultmethode.

Um Polymorphie anzuzeigen, muss der Programmierer das Schlüsselwort `virtual` verwenden, wie im Programm `LateBinding` zu sehen ist, das Sie in Listing 22-3 finden.

258 **Sonntagmorgen****Listing 22-3: Programm LateBinding**

```
// LateBinding - bei später Bindung wird die
// Entscheidung, welche der
// überschriebenen Funktionen
// aufgerufen wird, zur Laufzeit
// getroffen
#include <stdio.h>
#include <iostream.h>

class Student
{
public:
    virtual double calcTuition()
    {
        return 0;
    }
};
class GraduateStudent : public Student
{
public:
    virtual double calcTuition()
    {
        return 1;
    }
};

double fn(Student& fs)
{
    // weil calcTuition() virtual deklariert ist,
    // wird der Laufzeittyp von fs verwendet, um
    // den Aufruf aufzulösen
    return fs.calcTuition();
}

int main(int nArgc, char* pszArgs[])
{
    // der folgende Ausdruck ruft
    // fn() mit einem Student-Objekt
    Student s;
    cout << »Der Wert von s.calcTuition bei\n«
         << »virtuellem Aufruf durch fn() ist »
         << fn(s)
         << »\n\n«;

    // der folgende Ausdruck ruft
    // fn() mit einem GraduateStudent-Objekt
    GraduateStudent gs;
    cout << »Der Wert von gs.calcTuition bei\n«
         << »virtuellem Aufruf durch fn() ist »
         << fn(gs)
         << »\n\n«;
    return 0;
}
```

Lektion 22 – Polymorphie 259

Das Schlüsselwort `virtual`, das der Deklaration von `calcTuition()` zugefügt wurde, erzeugt eine virtuelle Elementfunktion. Das bedeutet, dass Aufrufe von `calcTuition()` spät gebunden werden, wenn der Typ des aufrufenden Objektes nicht zur Compilezeit bestimmt werden kann.

Das Programm `LateBinding` enthält den gleichen Aufruf der Funktion `fn()` wie in den beiden früheren Versionen. In dieser Version geht der Aufruf von `calcTuition()` an `Student::calcTuition()`, wenn `fs` ein `Student` ist und an `GraduateStudent::calcTuition()`, wenn `fs` ein `GraduateStudent` ist.

Die Ausgabe von `LateBinding` sehen Sie unten. Die Funktion `calcTuition()` als virtuell zu deklarieren, lässt `fn()` Aufrufe anhand des Laufzeittyps auflösen.

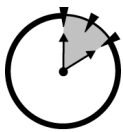
```
Der Wert von s.calcTuition bei
virtuellem Aufruf durch fn() ist 0

Der Wert von gs.calcTuition bei
virtuellem Aufruf durch fn() ist 1
```

Bei der Definition einer virtuellen Elementfunktion steht das Schlüsselwort `virtual` nur bei der Deklaration und nicht bei der Definition, wie im folgenden Beispiel zu sehen ist:

```
class Student
{
public:
    // deklariere als virtual
    virtual double calcTuition()
    {
        return 0;
    }
};

// 'virtual' kommt in der Definition nicht vor
double Student::calcTuition()
{
    return 0;
}
```



10 Min.

22.5 Was ist eine virtuelle Funktion nicht?

Nur weil Sie denken, dass ein bestimmter Funktionsaufruf spät gebunden wird, bedeutet das nicht, dass dies auch der Fall ist. C++ erzeugt beim Kompilieren keine Hinweise darauf, welche Aufrufe es früh und welche es spät bindet.

Die kritischste Sache ist die, dass alle Elementfunktionen, die in Frage kommen, identisch deklariert sind, ihren Rückgabetype eingeschlossen. Wenn sie nicht identisch deklariert sind, werden die Elementfunktionen nicht mittels Polymorphie überschrieben, ob sie nun als `virtual` deklariert sind oder nicht. Betrachten sie den folgenden Codeschnipsel:

```
#include <iostream.h>
class Base
{
public:
    virtual void fn(int x)
    {
        cout << »In Base int x = »
            << x << »\n«;
    }
};
class SubClass : public Base
{
public:
    virtual void fn(float x)
    {
        cout << »In SubClass, float x = »
            << x << »\n«;
    }
};

void test(Base &b)
{
    int i = 1;
    b.fn(i);           // nicht spät gebunden
    float f = 2.0;
    b.fn(f);           // und der Aufruf auch nicht
}
```

`fn( )` in `Base` ist als `fn(int)` deklariert, während die Version in der Unterklasse als `fn(float)` deklariert ist. Weil die Funktionen verschiedene Argumente haben, gibt es keine Polymorphie. Der erste Aufruf geht an `Base::fn(int)` – das ist nicht verwunderlich, weil `b` vom Typ `Base` und `i` ein `int` ist. Doch auch der nächste Aufruf geht an `Base::fn(int)`, nachdem `float` in `int` konvertiert wurde. Es wird kein Fehler erzeugt, weil dieses Programm legal ist (abgesehen von der Warnung, die Konvertierung von `f` betreffend). Die Ausgabe eines Aufrufs von `test( )` zeigt keine Polymorphie:

```
In Base, int x = 1
In Base, int x = 2
```

Die Argumente passen nicht exakt, es gibt keine späte Bindung – mit einer Ausnahme: Wenn die Elementfunktion in der Basisklasse einen Zeiger oder eine Referenz auf ein Objekt der Basisklasse zurückgibt, kann eine überschriebene Elementfunktion in einer Unterklasse einen Zeiger oder eine Referenz auf ein Objekt der Unterklasse zurückgeben. Mit anderen Worten, das Folgende ist erlaubt:

```
class Base
{
public:
    Base* fn();
};

class Subclass : public Base
{
public:
    Subclass* fn();
};
```


In der Praxis ist das ganz natürlich. Wenn eine Funktion mit `Subclass`-Objekten umgeht, scheint es natürlich zu sein, dass sie damit fortfährt.

22.6 Überlegungen zu virtual

Den Namen der Klasse im Aufruf anzugeben, erzwingt frühe Bindung. Der folgende Aufruf geht z.B. an `Base::fn()`, weil der Programmierer das so ausgedrückt hat, auch wenn `fn()` als virtual deklariert ist.

```
void test(Base &b)
{
    b.Base::fn(); // wird nicht spät gebunden
}
```

Eine als virtual deklarierte Funktion kann nicht inline sein. Um eine Inline-Funktion zu expandieren, muss der Compiler zur Compilezeit wissen, welche Funktion expandiert werden soll. Daher waren auch alle Elementfunktionen, die Sie in den bisherigen Beispielen gesehen haben, outline deklariert.

Konstruktoren können nicht virtual sein, weil es kein (fertiges) Objekt gibt, das zur Typbestimmung verwendet werden kann. Zum Zeitpunkt, an dem der Konstruktor aufgerufen wird, ist der Speicher, der von dem Objekt belegt wird, nur eine formlose Masse. Erst nachdem der Konstruktor fertig ist, ist das Objekt ein Element der Klasse im eigentlichen Sinne.

Im Vergleich dazu sollten Destruktoren normalerweise als virtual deklariert werden. Wenn nicht, gehen Sie das Risiko ein, dass ein Objekt nicht korrekt vernichtet wird, wie in folgender Situation:

```
class Base
{
public:
    ~Base();
};
class SubClass : public Base
{
public:
    ~SubClass();
};
void finishWithObject(Base *pHeapObject)
{
    // ... arbeite mit Objekt ...
    // jetzt gib es an den Heap zurück
    delete pHeapObject; // ruft ~Base() auf,
                        // unabhängig von Laufzeit-
                        // typ von pHeapObject
}
```

Wenn der Zeiger, der an `finishWithObject()` tatsächlich auf ein Objekt aus der Klasse `SubClass` zeigt, wird der `SubClass`-Destruktor trotzdem nicht korrekt aufgerufen. Den Destruktor virtuell zu deklarieren, löst das Problem.

262 Sonntagmorgen

Wann würden Sie also einen Destruktor nicht virtuell deklarieren? Es gibt nur eine solche Situation. Ich habe bereits erwähnt, dass virtuelle Funktionen einen gewissen »Mehraufwand« bedeuten. Lassen Sie mich ein wenig genauer sein. Wenn der Programmierer die erste virtuelle Funktion in einer Klasse definiert, fügt C++ einen zusätzlichen, versteckten Zeiger hinzu – nicht einen Zeiger pro virtueller Funktion, nur einen Zeiger, falls die Klasse mindestens eine virtuelle Funktion besitzt. Eine Klasse, die keine virtuellen Funktionen besitzt (und keine virtuellen Funktionen von einer Basisklasse erbt), besitzt keinen solchen Zeiger.

Nun, ein Zeiger klingt nicht nach sehr viel und ist es auch nicht, es sei denn die folgenden zwei Bedingungen sind erfüllt:

- Die Klasse hat nicht viele Datenelemente (so dass ein Zeiger viel ist im Vergleich zum Rest).
- Sie beabsichtigen, viele Objekte dieser Klasse zu erzeugen (ansonsten macht der zusätzliche Speicher keinen Unterschied).



0 Min.

Wenn diese beiden Bedingungen erfüllt sind und Ihre Klasse nicht bereits eine virtuelle Elementfunktion besitzt, können Sie Ihren Destruktor als nicht virtual deklarieren.

Normalerweise sollten Sie aber den Destruktor virtual deklarieren. Wenn Sie das mal nicht tun, dokumentieren Sie die Gründe dafür!

Zusammenfassung

Vererbung an sich ist schön, ist aber begrenzt in ihren Möglichkeiten. In Kombination mit Polymorphie, ist Vererbung ein mächtiges Programmierwerkzeug.

- Elementfunktionen in einer Klasse können Elementfunktionen überschreiben, die in der Basisklasse definiert sind. Aufrufe dieser Funktionen werden zur Compilezeit aufgelöst basierend auf der zur Compilezeit bekannten Klasse. Das wird frühe Bindung genannt.
- Eine Elementfunktion kann als virtual deklariert werden, wodurch Aufrufe basierend auf dem Laufzeittyp aufgelöst werden. Das wird Polymorphie oder späte Bindung genannt.
- Aufrufe, von denen bekannt ist, dass der Laufzeittyp und der Compiletyp gleich sind, werden früh gebunden, unabhängig davon, ob die Elementfunktion als virtual deklariert ist oder nicht.

Selbsttest

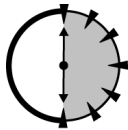
1. Was ist Polymorphie? (Siehe »Einstieg in Polymorphie«)
2. Was ist ein anderes Wort für Polymorphie? (Siehe »Einstieg in Polymorphie«)
3. Was ist die Alternative und wie wird sie genannt? (Siehe »Elementfunktionen überschreiben«)
4. Nennen Sie drei Gründe, weshalb C++ Polymorphie enthält. (Siehe »Polymorphie und objektorientierte Programmierung«)
5. Welches Schlüsselwort wird verwendet, um Elementfunktion polymorph zu deklarieren? (Siehe »Wie funktioniert Polymorphie?«)

Abstrakte Klassen und Faktorisieren



Checkliste

- ☒ Gemeinsame Eigenschaften in eine Basisklasse faktorisieren
- ☒ Abstrakte Klassen zur Speicherung faktorierter Informationen nutzen
- ☒ Abstrakte Klassen und dynamische Typen



30 Min.

Bis jetzt haben wir gesehen, wie Vererbung benutzt werden kann, um existierende Klassen für neue Anwendungen zu erweitern. Vererbung verlangt vom Programmierer die Fähigkeit, gleiche Eigenschaften verschiedener Klassen zu kombinieren; dieser Prozess wird *Faktorisieren* genannt.

23.1 Faktorisieren

Um zu sehen, wie Faktorisieren funktioniert, lassen Sie uns die beiden Klassen *Checking* (Girokonto) und *Savings* (Sparkonto) in einem hypothetischen Banksystem betrachten. Diese sind in Abbildung 23.1 grafisch dargestellt.

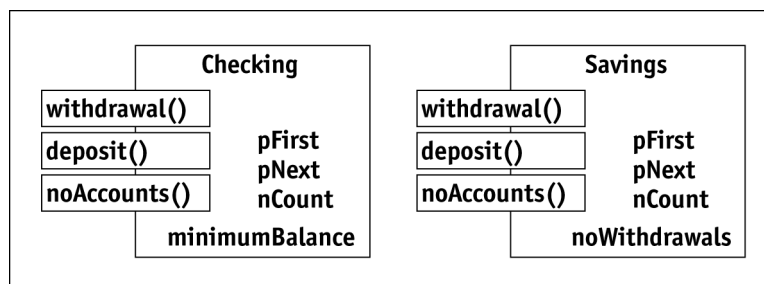


Abbildung 23.1: Unabhängige Klassen *Checking* und *Savings*.

264 Sonntagmorgen

Um diese Abbildung und die folgenden Abbildungen lesen zu können, halten Sie im Gedächtnis, dass

- die große Box eine Klasse ist, mit dem Klassennamen ganz oben,
- die Namen in den Boxen Elementfunktionen sind,
- die Namen ohne Box Datenelemente sind,
- die Namen, die teilweise außerhalb der Boxen liegen, öffentlich zugängliche Elemente sind; die anderen protected deklariert sind.
- ein dicker Pfeil die Beziehung IS_A repräsentiert und
- ein dünner Pfeil die Beziehung HAS_A repräsentiert.

Abbildung 23.1 zeigt, dass die Klassen `Checking` und `Savings` vieles gemein haben. Weil sie jedoch nicht identisch sind, müssen es zwei getrennte Klassen bleiben. Dennoch sollte es einen Weg geben, Wiederholungen zu vermeiden.

Wir könnten eine der Klassen von der anderen erben lassen. `Savings` hat ein Extraelement, es macht daher mehr Sinn, `Savings` von `Checking` abzuleiten, wie Sie in Abbildung 23.2 sehen, als umgekehrt. Die Klasse wird vervollständigt durch das Hinzufügen des Datenelementes `noWithdrawals` und dem virtuellen Überladen der Elementfunktion `withdrawal()`.

Obwohl die Lösung Arbeit einspart, ist sie nicht zufriedenstellend. Das Hauptproblem ist, dass es die Wahrheit falsch darstellt. Diese Vererbungsbeziehung impliziert, dass ein Sparkonto (`Savings`) ein spezieller Typ eines Girokontos (`Checking`) ist, was nicht der Fall ist.

»Na und?« werden Sie sagen. »Es funktioniert und spart Aufwand.« Das ist wahr, aber meine Vorbehalte sind mehr als sprachliche Trivialitäten. Solche Fehldarstellungen verwirren den Programmierer, den heutigen und den von morgen. Eines Tages wird ein Programmierer, der sich mit dem Programm nicht auskennt, das Programm lesen, und verstehen müssen, was der Code macht. Irreführende Tricks sind schwer zu durchschauen und zu verstehen.

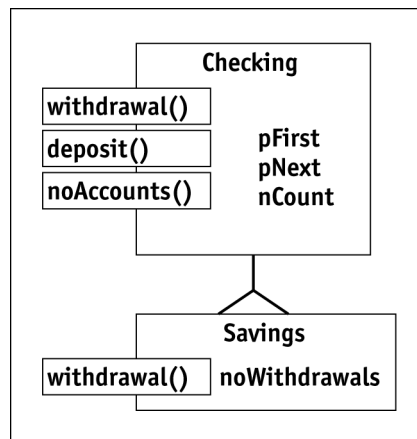


Abbildung 23.2: Savings implementiert als Unterklasse von Checking

Außerdem können solche Fehldarstellungen zu späteren Problemen führen. Nehmen Sie z.B. an, dass die Bank ihre Policen für Girokonten ändert. Sagen wir, die Bank entscheidet, dass sie eine Bearbeitungsgebühr nur dann verlangt, wenn der mittlere Kontostand im Monat unter einem gegebenen Wert liegt.

Lektion 23 – Abstrakte Klassen und Faktorisieren 265

Eine solche Änderung kann mit minimalen Änderungen an der Klasse `Checking` leicht durchgeführt werden. Wir müssen ein neues Datenelement in die Klasse `Checking` einführen, das wir `minimumBalance` nennen wollen.

Das erzeugt aber ein Problem. Weil `Savings` von `Checking` erbt, bekommt `Savings` ebenfalls ein solches Datenelement. Die Klasse hat aber für ein solches Element keine Verwendung, weil der minimale Kontostand das Sparkonto nicht beeinflusst. Ein zusätzliches Datenelement macht nicht so viel aus, aber es verwirrt.

Änderungen wie diese akkumulieren sich. Heute ist es ein zusätzliches Datenelement, morgen ist es eine geänderte Elementfunktion. Schließlich hat die Klasse `Savings` einen großen Ballast, der nur auf die Klasse `Checking` angewendet werden kann.

Wie vermeiden wir dieses Problem? Wir können beide Klassen auf einer neuen Klasse basieren lassen, die speziell für diesen Einsatz gebaut ist; lassen Sie uns diese Klasse `Account` (=Konto) nennen. Diese Klasse enthält alle Eigenschaften, die `Savings` und `Checking` enthalten, wie in Abbildung 1.3 zu sehen ist.

Wie löst das unser Problem? Erstens ist das eine viel treffendere Beschreibung der realen Welt (was immer das ist). In meinem Konzept gibt es etwas, das als Konto bezeichnet wird. Girokonto und Sparkonto sind Spezialisierungen dieses fundamentalen Konzeptes.

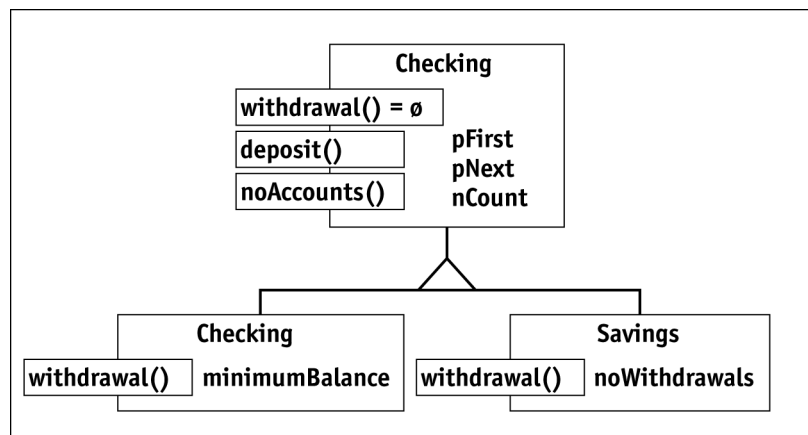


Abbildung 23.3: `Checking` und `Savings` auf Klasse `Account` basieren lassen

Zusätzlich bleibt die Klasse `Savings` von Änderungen an der Klasse `Checking` unberührt (und umgekehrt). Wenn die Bank eine grundlegende Änderung an allen Konten durchführen möchte, können wir die Klasse `Account` modifizieren und alle abgeleiteten Klassen erben diese Änderung automatisch. Aber wenn die Bank ihre Policen nur für Girokonten ändert, bleibt die Klasse `Savings` von dieser Änderung verschont.

Dieser Prozess, gleiche Eigenschaften aus ähnlichen Klassen zu extrahieren, wird als **Faktorisieren** bezeichnet. Das ist ein wichtiges Feature objektorientierter Sprachen aus den bereits genannten Gründen, plus einem neuen: Reduktion von Redundanz.

In Software ist nutzlose Masse eine üble Sache. Je mehr Code Sie generieren, desto mehr müssen Sie auch debuggen. Es lohnt nicht, Nachtschichten einzulegen, um cleveren Code zu generieren, der hier und da ein paar Zeilen Code einspart – diese Art Schlauheit entpuppt sich oft als Bumerang. Aber das Faktorisieren redundanter Information durch Vererbung kann den Programmieraufwand tatsächlich reduzieren.



Faktorisieren ist nur zulässig, wenn die Vererbungsbeziehung der Realität entspricht. Zwei Klassen `Mouse` und `Joystick` zu faktorisieren ist legitim, weil es beide Klassen sind, die Zeigerhardware beschreiben. Zwei Klassen `Mouse` und `Display` zu faktorisieren, weil sie elementare Systemfunktionen des Betriebssystems benutzen, ist nicht legitim – `Maus` und `Bildschirm` teilen keine Eigenschaft in der realen Welt.

Faktorisieren kann, und wird es auch in der Regel, zu mehreren Abstraktionsstufen führen. Ein Programm, das z.B. für eine fortschrittlichere Bank geschrieben wurde, könnte eine Klassenstruktur enthalten, wie in Abbildung 23.4 zu sehen ist.

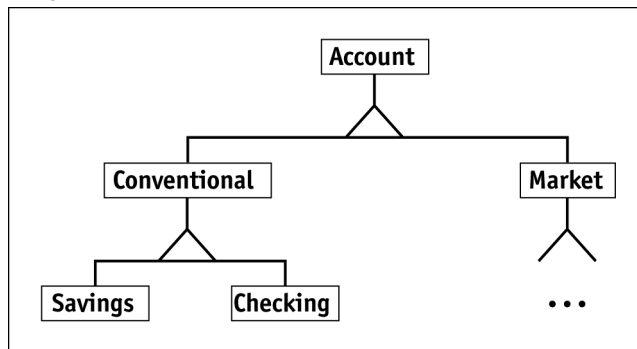


Abbildung 23.4: Eine weiter entwickelte Hierarchie für eine Bank.

Es wurde eine weitere Klasse zwischen den Klassen `Checking` und `Savings` und der allgemeinen Klasse `Account` eingeführt. Diese Klasse `Conventional` enthält die Features konventioneller Kontos. Andere Kontotypen wie z.B. Aktiondepots, sind ebenso vorgesehen.

Solche mehrarmigen Klassenstrukturen sind üblich und anzustreben, so lange ihre Beziehungen die Wirklichkeit widerspiegeln. Es gibt jedoch nicht nur eine korrekte Klassenhierarchie für eine gegebene Menge von Klassen.

Nehmen Sie an, dass unsere Bank es ihren Kunden ermöglicht, Girokonten und Aktiendepots online zu verwalten. Transaktionen für andere Kontotypen können nur bei der Bank getätigt werden. Obwohl die Klassenstruktur in Abbildung 23.4 natürlich erscheint, ist die Hierarchie in Abbildung 23.5 mit dieser Information ebenfalls gerechtfertigt. Der Programmierer muss entscheiden, welche Klassenhierarchie am besten zu den Daten passt, und zu der saubersten und natürlichsten Implementierung führen wird.

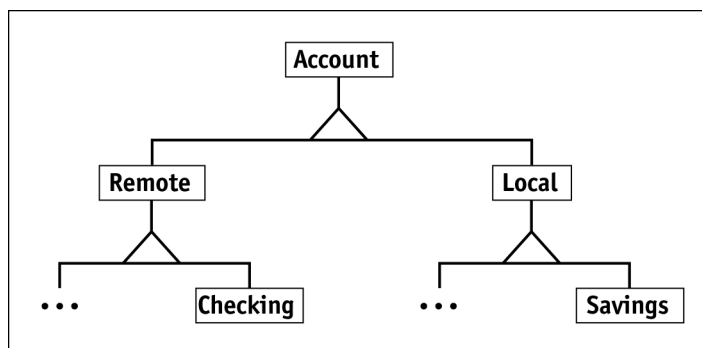
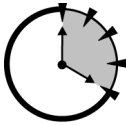


Abbildung 23.5: Eine alternative Klassenhierarchie zu Abbildung 23.4



20 Min.

23.2 Abstrakte Klassen

So sehr Faktorisieren auch den Intellekt befriedigt, bringt es ein Problem mit sich. Lassen Sie uns ein weiteres Mal auf das Kontobeispiel zurückkommen, insbesondere auf die gemeinsame Basisklasse `Account`. Lassen Sie uns eine Minute überlegen, wie wir die verschiedenen Elementfunktionen von `Account` definieren würden.

Die meisten Elementfunktionen von `Account` sind kein Problem, weil beide Kontotypen sie auf die gleiche Weise implementieren. Bei `withdrawal()` ist das anders. Die Regeln zum Abheben sind bei Girokonten und Sparkonten verschieden. Somit würden wir erwarten, dass `Savings::withdrawal()` und `Checking::withdrawal()` unterschiedlich implementiert sind. Aber die Frage ist ja, wie implementieren wir dann `Account::withdrawal()`?

»Kein Problem«, werden Sie sagen. »Gehen Sie einfach zu Ihrer Bank und fragen Sie dort »Wie sind die Regeln für das Abheben von Konten?« Die Antwort ist »Welche Art Konto?« Ein ratloser Blick.

Das Problem ist, dass die Frage keinen Sinn macht. Es gibt keine Sache »einfach Konto«. Alle Konten (in diesem Beispiel) sind entweder Girokonten oder Sparkonten. Das Konzept Konto ist ein abstraktes, das gleiche Eigenschaften der konkreten Klassen faktoriert. Es ist jedoch unvollständig, weil es die kritische Eigenschaft `withdrawal()` nicht besitzt. (Wenn wir zu den Details kommen, werden wir weitere Eigenschaften finden, die einem einfachen Konto fehlen.)

Lassen Sie mich ein Beispiel aus der Tierwelt entleihen. Wir können die verschiedenen Spezies der warmblütigen lebendgebärenden Tiere unterscheiden und daraus schließen, dass es ein Konzept Säugetiere gibt. Wir können von dieser Klasse *Säugetier* Klassen ableiten wie *Hund*, *Katze* und *Mensch*. Es ist jedoch nicht möglich, irgendwo etwas zu finden, das ein reines Säugetier ist, d.h. ein Säugetier, das nicht zu einer der Spezies gehört. Säugetier ist ein Konzept auf hohem Abstraktionsniveau – es gibt keine Instanz von Säugetier.



Das Konzept Säugetier unterscheidet sich grundlegend vom Konzept Hund.
»Hund« ist ein Name, den wir einem existierenden Objekt gegeben haben. Es gibt nichts in der realen Welt was nur Säugetier ist.

Wir möchten nicht, dass der Programmierer ein Objekt `Account` (=Konto) oder eine Klasse *Mammal* (=Säugetier) erzeugt, weil wir nicht wissen, was wir damit anfangen sollen. Um diesem Problem zu begegnen, erlaubt es C++ dem Programmierer, eine Klasse zu deklarieren, von der kein Objekt instanziiert werden kann. Der einzige Sinn einer solchen Klasse ist, dass sie vererben kann.

Eine Klasse, die nicht instanziiert werden kann, heißt *abstrakte Klasse*.

23.2.1 Deklaration einer abstrakten Klasse

Eine abstrakte Klasse ist eine Klasse mit einer oder mehreren rein virtuellen Funktionen. Eine *rein virtuelle Funktion* ist eine virtuelle Elementfunktion, die so markiert ist, dass sie keine Implementierung besitzt.

Eine rein virtuelle Funktion hat keine Implementierung, weil wir nicht wissen, wie wir sie implementieren sollen. Z.B. wissen wir nicht, wie wir `withdrawal()` in einer Klasse `Account` ausführen sollen. Das macht einfach keinen Sinn. Wir können jedoch nicht einfach die Definition von `withdra-`

268 **Sonntagmorgen**

wal() weglassen, weil C++ annehmen wird, dass wir vergessen haben, die Funktion zu definieren und uns einen Linkfehler ausgeben wird, der uns mitteilt, dass eine Funktion fehlt (wahrscheinlich vergessen).

Die Syntax zur Deklaration einer rein virtuellen Funktion – die C++ mitteilt, dass die Funktion keine Definition hat – wird in folgender Klasse `Account` demonstriert:

```
// Account - das ist eine abstrakte Basisklasse
//           für alle Kontoklassen
class Account
{
public:
    Account(unsigned nAccNo);

    // Zugriffsfunktionen
    int accountNo();
    Account* first();
    Account* next();

    // Transaktionsfunktionen
    virtual void deposit(float fAmount) = 0;
    virtual void withdrawal(float fAmount) = 0;

protected:
    // speichere Kontoobjekte in einer Liste, damit
    // es keine Beschränkung der Anzahl gibt
    static Account* pFirst;
    Account* pNext;

    // alle Konten haben eine eindeutige Nummer
    unsigned nAccountNumber;
};
```

Die `=0` hinter der Deklaration von `deposit( )` und `withdrawal( )` zeigt an, dass der Programmierer nicht beabsichtigt, diese Funktionen zu definieren. Die Deklaration ist ein Platzhalter für die Unterklassen. Von den konkreten Unterklassen von `Account` wird erwartet, dass sie diese Funktionen mit konkreten Funktionen überladen.



Eine konkrete Elementfunktion ist eine Funktion, die nicht rein virtuell ist. Alle Elementfunktion vor dieser Sitzung waren konkret.



Obwohl diese Notation, die `=0` verwendet, anzeigt, dass eine Elementfunktion abstrakt ist, bizarr ist, bleibt sie doch so. Es gibt einen obskuren Grund dafür, wenn auch nicht gerade eine Rechtfertigung, aber das geht über den Bereich dieses Buches hinaus.

Lektion 23 – Abstrakte Klassen und Faktorisieren 269

Eine abstrakte Klasse kann nicht mit einem Objekt instanziiert werden. D.h. sie können kein Objekt aus einer abstrakten Klasse anlegen. Z.B. ist das Folgende nicht möglich:

```
void fn()
{
    // deklariere ein Konto
    Account acnt(1234); // das ist nicht erlaubt
    acnt.withdrawal(50); // was soll das tun?
}
```

Wenn die Deklaration erlaubt wäre, würde das resultierende Objekt unvollständig sein, und einige Eigenschaften vermissen lassen. Was soll z.B. der obige Aufruf von `acnt.withdrawal(50)` tun? Es gibt keine Funktion `Account::withdrawal()`.

Abstrakte Klassen dienen als Basisklassen für andere Klassen. Ein `Account` enthält alle die Eigenschaften, die wir einem generischen Konto zuschreiben, die Möglichkeit des Abhebens und des Einzahlens eingeschlossen. Wir können nur nicht definieren, wie ein generisches Konto solche Dinge ausführt – es bleibt den Unterklassen, das zu definieren. Anders ausgedrückt, ein Konto ist so lange kein Konto, bis der Benutzer Einzahlungen und Abhebungen machen kann, selbst wenn solche Operationen nur in speziellen Kontotypen definiert werden können, wie z.B. Girokonten und Sparkonten.

Wir können weitere Typen vom Konten durch Ableitung von `Account` erzeugen, aber sie können nicht durch ein Objekt instanziiert werden, so lange sie abstrakt bleiben.

23.2.2 Erzeugung einer konkreten Klasse aus einer abstrakten Klasse

Die Unterklasse einer abstrakten Klasse bleibt abstrakt, bis alle virtuellen Funktionen überladen sind. Die folgende Klasse `Savings` ist nicht abstrakt, weil sie die rein virtuellen Funktionen `deposit()` und `withdrawal()` mit perfekten Definitionen überlädt.

```
// Account - das ist eine abstrakte Basisklasse
//           für alle Kontoklassen
class Account
{
public:
    Account(unsigned nAccNo);
    // Zugriffsfunktionen
    int accountNo();
    Account* first();
    Account* next();

    // Transaktionsfunktionen
    virtual void deposit(float fAmount) = 0;
    virtual void withdrawal(float fAmount) = 0;

protected:
    // speichere Kontoobjekte in einer Liste, damit
    // es keine Beschränkung der Anzahl gibt
    static Account* pFirst;
    Account* pNext;

    // alle Konten haben eine eindeutige Nummer
    unsigned nAccountNumber;
};
```

```
// Savings - implementiert das Konzept Account
class Savings : public Account
{
public:
    // Konstruktor - Sparbücher werden mit einem
    //          initialen Kontostand erzeugt
    Savings(unsigned nAccNo,
            float fInitialBalance)
        : Account(nAccNo)
    {
        fBalance = fInitialBalance;
    }

    // Sparkonten wissen, wie diese Operationen
    // ausgeführt werden
    virtual void deposit(float fAmount);
    virtual void withdrawal(float fAmount);

protected:
    float fBalance;
};

// deposit and withdrawal - definiert die Standard-
//          kontooperationen für
//          Sparkonten
void Savings::deposit(float fAmount)
{
    // ... die Funktion ...
}
void Savings::withdrawal(float fAmount)
{
    // ... die Funktion ...
}
```

Ein Objekt der Klasse `Savings` weiß, wie Einzahlungen und Abhebungen ausgeführt werden, wenn sie aufgerufen werden. Das Folgende macht also Sinn:

```
void fn()
{
    Savings s(1234);
    s.deposit(100.0);
}
```



Die Klasse `Account` hat einen Konstruktor, obwohl sie abstrakt ist. Alle Konten werden mit einer ID erzeugt. Die konkrete Kontoklasse `Savings` übergibt die ID an die Basisklasse `Account`, während Sie den initialen Kontostand selbst übernimmt. Das ist Teil unseres Objektmodells – die Klasse `Account` enthält das Element für die Kontonummer, somit wird es `Account` überlassen, dieses Feld zu initialisieren.

Lektion 23 – Abstrakte Klassen und Faktorisieren 271

**10 Min.****23.2.3 Warum ist eine Unterklasse abstrakt?**

Eine Unterklasse einer abstrakten Klasse kann abstrakt bleiben. Stellen Sie sich vor, wir hätten eine weitere Zwischenklasse in die Klassenhierarchie eingefügt. Nehmen Sie z.B. an, dass meine Bank so etwas wie ein Geldkonto eingeführt hat.

Geldkonten sind Konten, in denen Guthaben in Geld und nicht z.B. in Wertpapieren vorliegt. Girokonten und Sparkonten sind Geldkonten. Alle Einzahlungen auf Geldkonten werden bei meiner Bank gleich gehandhabt; Sparkonten berechnen jedoch nach den ersten fünf Abhebungen Gebühren, während Girokonten keine Gebühren für etwas erheben.

Mit diesen Definitionen kann die Klasse `CashAccount` die Funktion `deposit()` implementieren, weil die Operation wohldefiniert und für alle Geldkonten gleich ist; `CashAccount` kann jedoch `withdrawal()` nicht implementieren, weil unterschiedliche Geldkonten diese Operation unterschiedlich ausführen.

In C++ sehen die Klassen `CashAccount` und `Savings` wie folgt aus:

```
// CashAccount - ein Geldkonto speichert Geldwerte
//                und nicht z.B. Wertpapiere.
//                Geldkonten erfordern Geldangaben,
//                alle Einzahlungen werden gleich
//                behandelt. Abhebungen werden von
//                verschiedenen Geldkontoformen ver-
//                schieden gehandhabt.
class CashAccount : public Account
{
public:
    CashAccount(unsigned nAccNo,
                float fInitialBalance)
        : Account(nAccNo)
    {
        fBalance = fInitialBalance;
    }

    // Transaktionsfunktionen
    // deposit - alle Geldkonten erwarten
    //            Einzahlungen als Betrag
    virtual void deposit(float fAmount)
    {
        fBalance += fAmount;
    }

    // Zugriffsfunktionen
    float balance()
    {
        return fBalance;
    }

protected:
    float fBalance;
};

// Savings - ein Sparkonto ist ein Geldkonto; die
//            Operation des Abhebens ist wohldefiniert
class Savings : public CashAccount
```

272 **Sonntagmorgen**

```

{
    public:
        Savings(unsigned nAccNo,
                float fInitialBalance = 0.0F)
            : CashAccount(nAccNo, fInitialBalance)
        {
            // ... was immer Savings tun muss,
            // was ein Account noch nicht getan hat ...
        }

        // ein Sparkonto weiß, wie Abheben funktionieren
        virtual void withdrawal(float fAmount);
};

// fn - eine Testfunktion
void fn()
{
    // eröffne ein Sparkonto mit $200 darauf
    Savings savings(1234, 200);

    // Einzahlung $100
    savings.deposit(100);

    // und $50 abheben
    savings.withdrawal(50);
}

```

Die Klasse `CashAccount` bleibt abstrakt, weil sie die Funktion `deposit()`, aber nicht die Funktion `withdrawal()` überlädt. `Savings` ist konkret, weil sie die verbleibende rein virtuelle Elementfunktion überlädt.

Die Testfunktion `fn()` erzeugt ein `Savings`-Objekt, tätigt eine Einzahlung und dann eine Abhebung.



Ursprünglich musste jede rein virtuelle Funktion in einer Unterklasse überladen werden, selbst wenn die Funktion mit einer weiteren rein virtuellen Funktion überladen wurde. Schließlich haben die Leute festgestellt, dass das genauso dumm ist, wie es sich anhört, und haben diese Forderung fallen gelassen. Weder Visual C++ noch GNU C++ stellen diese Forderung, ältere Compiler tun das möglicherweise.

23.2.4 Ein abstraktes Objekt an eine Funktion übergeben

Obwohl Sie keine abstrakte Klasse instanzieren können, ist es möglich, einen Zeiger oder eine Referenz auf eine abstrakte Klasse zu deklarieren. Mit Polymorphie ist das aber gar nicht so verrückt, wie es klingt. Betrachten Sie den folgenden Codeschnipsel:

```

void fn(Account* pAccount){ // das ist legal
{
    pAccount->withdrawal(100.0);
}
}

```

Lektion 23 – Abstrakte Klassen und Faktorisieren**273**

```
void otherFn()
{
    Savings s;

    // ist legal weil Savings IS_A Account
    fn(&s);
}
```

Hier wird `pAccount` als Zeiger auf `Account` deklariert. Die Funktion `fn()` darf `pAccount->withdrawal()` aufrufen, weil alle Konten wissen, wie sie Auszahlungen vornehmen. Es ist aber auch klar, dass die Funktion beim Aufruf die Adresse eines Objektes einer nichtabstrakten Unterklasse übergeben bekommt, wie z.B. `Savings`.

Es ist wichtig, hier darauf hinzuweisen, dass jedes Objekt, das `fn()` übergeben wird, entweder aus `Savings` kommt, oder aus einer anderen nichtabstrakten Unterklasse von `Account`. Die Funktion `fn()` kann sicher sein, dass wir niemals ein Objekt aus der Klasse `Account` übergeben werden, weil wir so ein Objekt nie erzeugen können. Das Folgende kann also nie passieren, weil C++ es nicht erlauben würde:

```
void otherFn()
{
    // das Folgende ist nicht erlaubt, weil Account
    // eine abstrakte Klasse ist
    Account a;

    fn(&a);
}
```

Der Schlüssel ist, dass es `fn()` erlaubt war, `withdrawal()` mit einem abstrakten `Account`-Objekt aufzurufen, weil jede konkrete Unterklasse von `Account` weiß, wie sie die Operation `withdrawal()` ausführen muss.



Eine rein virtuelle Funktion stellt ein Versprechen dar, eine bestimmte Eigenschaft in den konkreten Unterklassen zu implementieren.

23.2.5 Warum werden rein virtuelle Funktionen benötigt?

Wenn `withdrawal()` nicht in der Basisklasse `Account` definiert werden kann, warum lässt man sie dann dort nicht weg? Warum definiert man die Funktion nicht in `Savings` und lässt sie aus `Account` heraus? In vielen objektorientierten Sprachen können Sie das nur so machen. Aber C++ möchte in der Lage sein, zu überprüfen, dass Sie wirklich wissen, was Sie tun.

C++ ist eine streng getypte Sprache. Wenn Sie eine Elementfunktion ansprechen, besteht C++ darauf, dass sie beweisen, dass die Elementfunktion in der von Ihnen angegebenen Klasse existiert. Das verhindert unglückliche Laufzeitüberraschungen, wenn eine referenzierte Elementfunktion nicht gefunden werden kann.

274 **Sonntagmorgen**

Lassen Sie uns die folgenden kleineren Änderungen an Account ausführen, um das Problem zu demonstrieren:

```
class Account
{
    // wie zuvor, doch ohne Deklaration von
    // withdrawal()
};
class Savings : public Account
{
    public:
        virtual void withdrawal(float fAmount);
};

void fn(Account* pAcc)
{
    // hebe etwas Geld ab
    pAcc->withdrawal(100.00F); // das ist nicht
                              // erlaubt, weil
                              // withdrawal() nicht
                              // Element der Klasse
                              // Account ist
};

int otherFn()
{
    Savings s; // eröffne ein Konto
    fn(&s);
    //... Fortsetzung ...
}
```

Die Funktion `otherFn()` arbeitet wie zuvor. Wie vorher auch, versucht die Funktion `fn()` die Funktion `withdrawal()` mit dem `Account`-Objekt aufzurufen, das sie erhält. Weil die Funktion `withdrawal()` kein Element von `Account` ist, erzeugt der Compiler jedoch einen Fehler.

In diesem Fall hat die Klasse `Account` kein Versprechen abgegeben, eine Elementfunktion `withdrawal()` zu implementieren. Es könnte eine konkrete Unterklasse von `Account` geben, die keine solche Operation `withdrawal()` definiert. In diesem Fall hätte der Aufruf `pAcc->withdrawal()` keinen Zielort – das ist eine Möglichkeit, die C++ nicht akzeptieren kann.



0 Min.

Zusammenfassung

Klassen von Objekten auf der Basis wachsender Gemeinsamkeiten in Hierarchien aufzuteilen, wird als Faktorisieren bezeichnet. Faktorisieren führt fast unausweichlich zu Klassen, die eher konzeptionell als konkret sind. Ein Mensch ist ein Primat, ist ein Säugetier; die Klasse `Mammal` (=Säugetier) ist jedoch konzeptionell und nicht konkret – es gibt keine Instanz von `Mammal`, die nicht zu einer bestimmten Spezies gehört.

Sie haben ein Beispiel dafür mit der Klasse `Account` gesehen. Während es Sparkonten und Girokonten gibt, gibt es kein Objekt, das einfach nur ein Konto ist. In C++ sagen wir, dass `Account` eine abstrakte Klasse ist. Eine Klasse wird abstrakt, sobald eine ihrer Elementfunktionen keine Definition besitzt. Eine Unterklasse wird konkret, wenn sie alle Eigenschaften definiert, die in der abstrakten Basisklasse offengelassen wurden.

Lektion 23 – Abstrakte Klassen und Faktorisieren**275**

- Eine Elementfunktion, die keine Implementierung hat, wird als rein virtuell bezeichnet. Rein virtuelle Funktionen werden mit »=0« am Ende ihrer Deklaration bezeichnet. Rein virtuelle Funktionen haben keine Definition.
- Eine Klasse, die eine oder mehrere rein virtuelle Funktionen enthält, wird als abstrakte Klasse bezeichnet.
- Eine abstrakte Klasse kann nicht instanziiert werden.
- Eine abstrakte Klasse kann Basisklasse anderer Klassen sein.
- Unterklassen einer abstrakten Klasse werden konkret (d.h. nicht abstrakt), wenn sie alle rein virtuellen Funktionen überschrieben haben, die sie erben.

Selbsttest

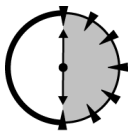
1. Was ist Faktorisieren? (Siehe »Faktorisieren«)
2. Was ist das unterscheidende Merkmal einer abstrakten Klasse in C++? (Siehe »Deklaration einer abstrakten Klasse«)
3. Wie erzeugen Sie aus einer abstrakten Klasse eine konkrete Klasse? (Siehe »Erzeugen einer konkreten Klasse aus einer abstrakten Klasse«)
4. Warum ist es möglich, eine Funktion `fn(MyClass*)` zu deklarieren, wenn `MyClass` abstrakt ist? (Siehe »Ein abstraktes Objekt an eine Funktion übergeben«)



Mehrfachvererbung

Checkliste

- ☒ Mehrfachvererbung einführen
- ☒ Uneindeutigkeiten bei Mehrfachvererbung vermeiden
- ☒ Uneindeutigkeiten bei virtueller Vererbung vermeiden
- ☒ Ordnungsregeln für mehrere Konstruktoren wiederholen



30 Min.

In den bisher diskutierten Klassenhierarchien hat jede Klasse von einer einzelnen Elternklasse geerbt. Das ist die Art, wie es auch normalerweise in der realen Welt zugeht. Eine Mikrowelle ist ein Typ Ofen. Man könnte argumentieren, dass eine Mikrowelle Gemeinsamkeiten mit einem Radar hat, der auch Mikrowellen verwendet, aber das ist wirklich ein bißchen weit hergeholt.

Einige Klassen jedoch stellen die Vereinigung zweier Klassen dar. Ein Beispiel einer solchen Klasse ist das Schlafsofa. Wie der Name bereits impliziert, ist es ein Sofa und auch ein Bett (wenn auch kein sehr komfortables). Somit sollte das Schlafsofa Eigenschaften eines Bettes und Eigenschaften eines Sofas erben. Um dieser Situation zu begegnen, erlaubt es C++, eine Klasse von mehr als einer Basisklasse abzuleiten. Das wird Mehrfachvererbung genannt.

24.1 Wie funktioniert Mehrfachvererbung?

Lassen Sie uns das Beispiel mit dem Schlafsofa ausbauen, um die Prinzipien der Mehrfachvererbung zu untersuchen. Abbildung 24.1 zeigt den Vererbungsgraph der Klasse `SleeperSofa`. Beachten Sie, wie die Klasse von der Klasse `Sofa` und der Klasse `Bed` erbt. Auf diese Weise erbt sie die Eigenschaften der beiden Klassen.

Lektion 24 – Mehrfachvererbung

277

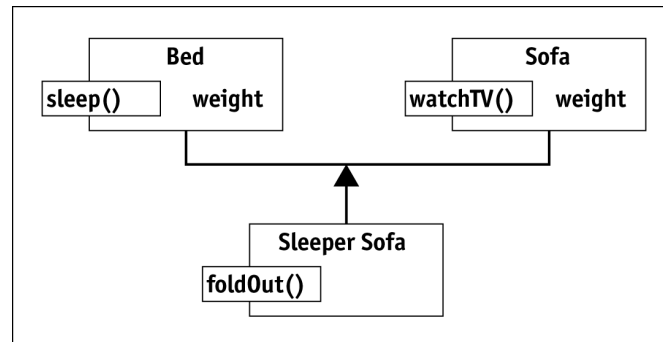


Abbildung 24.1: Klassenhierarchie eines Schlafsofas.

Der Code zur Implementierung von `SleeperSofa` sieht wie folgt aus:

```

// SleeperSofa - demonstriert, wie ein Schlafsofa
//                funktionieren könnte
#include <stdio.h>
#include <iostream.h>

class Bed
{
public:
    Bed()
    {
        cout << »Teil Bett\n«;
    }
    void sleep()
    {
        cout << »Versuche zu schlafen\n«;
    }
    int weight;
};

class Sofa
{
public:
    Sofa()
    {
        cout << »Teil Sofa\n«;
    }
    void watchTV()
    {
        cout << »Sehe fern\n«;
    }
    int weight;
};

// SleeperSofa - ist Bett und Sofa
class SleeperSofa : public Bed, public Sofa
{
public:

```

278 **Sonntagmorgen**

```

// der Konstruktor muss nichts machen
SleepersSofa()
{
    cout << »Zusammenfügen der beiden\n«;
}
void foldOut()
{
    cout << »Klappe das Bett aus\n«;
}
};

int main()
{
    SleepersSofa ss;
    // sie können auf einem Schlafsofa fernsehen ...
    ss.watchTV();          // Sofa::watchTV()
    //... und Sie können es ausklappen ...
    ss.foldOut();          // SleepersSofa::foldOut()
    //... und darauf schlafen (irgendwie)
    ss.sleep();            // Bed::sleep()
    return 0;
}

```

Die Namen der beiden Klassen – Bed und Sofa – kommen nach dem Namen SleepersSofa, was anzeigt, dass SleepersSofa die Elemente der beiden Basisklassen erbt. Somit sind beide Aufrufe `ss.sleep()` und `ss.watchTV()` gültig. Sie können ein SleepersSofa entweder als Bed oder als Sofa benutzen. Zusätzlich kann die Klasse SleepersSofa eigene Elemente haben, wie z.B. `foldOut()`.

Die Ausführung des Programms liefert die folgende Ausgabe:

```

Teil Bett
Teil Sofa
Zusammenfügen der beiden
Sieh fern
Klappe das Bett aus
Versuche zu schlafen

```

Der Teil Bett des Schlafsofas wird zuerst konstruiert, weil die Klasse Bed zuerst in der Klassenliste steht, von denen SleepersSofa erbt (es hängt nicht an der Reihenfolge, in der die Klassen definiert sind). Danach wird der Teil Sofa des Schlafsofas konstruiert. Schließlich legt SleepersSofa selber los.

Nachdem ein SleepersSofa-Objekt erzeugt wurde, greift `main()` nacheinander auf die Elementfunktionen zu – erst wird ferngesehen auf dem Sofa, dann wird das Sofa umgebaut, und dann wird auf dem Sofa geschlafen. (Offensichtlich hätten die Elementfunktionen in jeder Reihenfolge aufgerufen werden können.)

24.2 Uneindeutigkeiten bei Vererbung

Obwohl Mehrfachvererbung ein mächtiges Feature ist, bringt sie doch einige mögliche Probleme für den Programmierer mit sich. Eines ist im vorangegangenen Beispiel zu sehen. Beachten Sie, dass beide Klassen, Bed und Sofa, ein Element `weight` (Gewicht) enthalten. Das ist logisch, weil beide ein messbares Gewicht haben. Die Frage ist, welches `weight` von SleepersSofa geerbt wird.

Lektion 24 – Mehrfachvererbung 279

Die Antwort ist »beide«. `SleeperSofa` erbt ein Element `Bed::weight` und ein Element `Sofa::weight`. Weil sie den gleichen Namen haben, sind unqualifizierte Referenzen uneindeutig. Der folgende Schnipsel demonstriert das Prinzip:

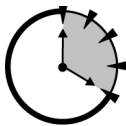
```
int main()
{
    // gib das Gewicht eines Schlafsofas aus
    SleeperSofa ss;
    cout << »Gewicht des Schlafsofas = »
        << ss.weight // das funktioniert nicht!
        << »\n«;
    return 0;
}
```

Das Programm muss eine der beiden Gewichtsangaben über den entsprechenden Namen der Basisklasse ansprechen. Der folgende Codeschnipsel ist korrekt:

```
#include <iostream.h>
void fn()
{
    SleeperSofa ss;
    cout << »Gewicht des Schlafsofas = »
        << ss.Sofa::weight // Angabe, welches weight
        << »\n«;
}
```

Obwohl diese Lösung das Problem löst, ist die Angabe einer Basisklasse in einer Anwendungsfunktion nicht wünschenswert, weil dadurch Informationen über die Klasse in den Anwendungsscode verlagert werden. In diesem Fall muss `fn()` wissen, dass `SleeperSofa` von `Sofa` erbt.

Diese Typen sogenannter Kollisionen können bei einfacher Vererbung nicht auftreten, sind aber bei Mehrfachvererbung eine ständige Gefahr.

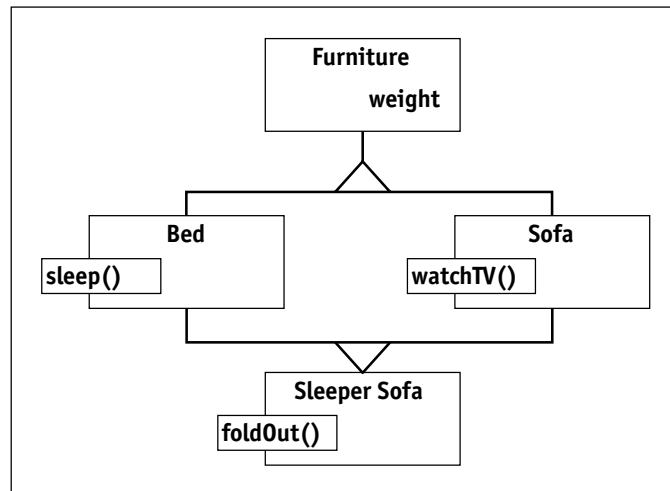


20 Min.

24.3 Virtuelle Vererbung

Im Falle von `SleeperSofa`, war die Kollision der Elemente `weight` mehr, als nur ein Unfall. Ein `SleeperSofa` hat kein Bettgewicht, unabhängig von seinem Sofagewicht – es hat nur ein Gewicht. Die Kollision entsteht, weil diese Klassenhierarchie die reale Welt nicht vollständig beschreibt. Insbesondere wurden die Klassen nicht vollständig faktoriert.

Wenn man etwas mehr nachdenkt, wird klar, dass Betten und Sofas Spezialfälle eines grundlegenden Konzeptes sind: Möbel. (Natürlich könnte ich das Konzept noch viel fundamentaler machen z.B. mit einer Klasse `ObjectWithMass` (=Objekte mit Masse), aber `Furniture` (=Möbel) ist fundamental genug.) Gewicht ist eine Eigenschaft von allen Möbelstücken. Diese Beziehung ist in Abbildung 24.2 zu sehen:

280 **Sonntagmorgen****Abbildung 24.2: Weiteres Faktorisieren von Bed und Sofa.**

Die Klasse `Furniture` zu faktorisieren sollte die Namenkollision auflösen. Sehr erleichtert, und mit großer Hoffnung auf Erfolg, realisiere ich die folgende C++-Hierarchie im Programm `AmbiguousInheritance`:

```

// AmbiguousInheritance - beide Klassen Bed und Sofa
//                        können von einer Klasse
//                        Furniture erben
#include <stdio.h>
#include <iostream.h>

class Furniture
{
public:
    Furniture()
    {
        cout << »Erzeugen des Konzeptes Furniture«;
    }
    int weight;
};

class Bed : public Furniture
{
public:
    Bed()
    {
        cout << »Teil Bett\n«;
    }
    void sleep()
    {
        cout << »Versuche zu schlafen\n«;
    }
    int weight;
};
  
```

Lektion 24 – Mehrfachvererbung 281

```

class Sofa : public Furniture
{
public:
    Sofa()
    {
        cout << »Teil Sofa\n«;
    }
    void watchTV()
    {
        cout << »Sehe fern\n«;
    }
    int weight;
};

// SleeperSofa - ist Bett und Sofa
class SleeperSofa : public Bed, public Sofa
{
public:
    // der Konstruktor muss nichts machen
    SleeperSofa()
    {
        cout << »Zusammenfügen der beiden\n«;
    }
    void foldOut()
    {
        cout << »Klappe das Bett aus\n«;
    }
};

int main()
{
    // Ausgabe des Gewichts eines Schlafsofas
    SleeperSofa ss;
    cout << »Gewicht des Schlafsofas = »
        << ss.weight // das funktioniert nicht!
        << »\n«;
    return 0;
}

```

Unglücklicherweise hilft das gar nicht – weight ist immer noch uneindeutig. »OK«, sage ich (wobei ich nicht wirklich verstehe, warum weight immer noch uneindeutig ist), »Ich werde es einfach nach Furniture casten«.

```

#include <iostream.h>
void fn()
{
    SleeperSofa ss;
    Furniture *pF;
    pF = (Furniture*)&ss; // nutze Zeiger auf
                        // Furniture...
    cout << »weight = » //... um an das Gewicht
        << pF->weight // zu kommen
        << »\n«;
};

```

282 Sonntagmorgen

Auch das funktioniert nicht. Jetzt bekomme ich eine Fehlermeldung, dass der Cast von `SleeperSofa` nach `Furniture*` uneindeutig ist. Was geht hier eigentlich vor?

Die Erklärung ist einfach. `SleeperSofa` erbt nicht direkt von `Furniture`. Beide Klassen, `Bed` und `Sofa`, erben von `Furniture`, und `SleeperSofa` erbt dann von beiden. Im Speicher sieht ein `SleeperSofa`-Objekt wie in Abbildung 24.3 aus:

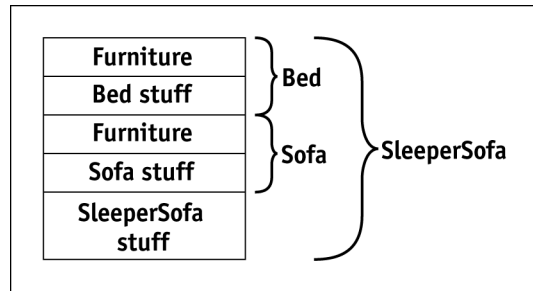


Abbildung 24.3: Speicheranordnung eines `SleeperSofa`.

Sie sehen, dass `SleeperSofa` aus einem vollständigen `Bed` besteht, gefolgt von einem vollständigen `Sofa`, gefolgt von Dingen, die für `SleeperSofa` spezifisch sind. Jedes dieser Teilobjekte in `SleeperSofa` hat seinen eigenen `Furniture`-Teil, weil jeder von `Furniture` erbt. Somit enthält ein `SleeperSofa` zwei `Furniture`-Objekte.

Ich habe nicht die Hierarchie in Abbildung 24.2 erzeugt, sondern eine Hierarchie, wie sie in Abbildung 24.4 zu sehen ist.

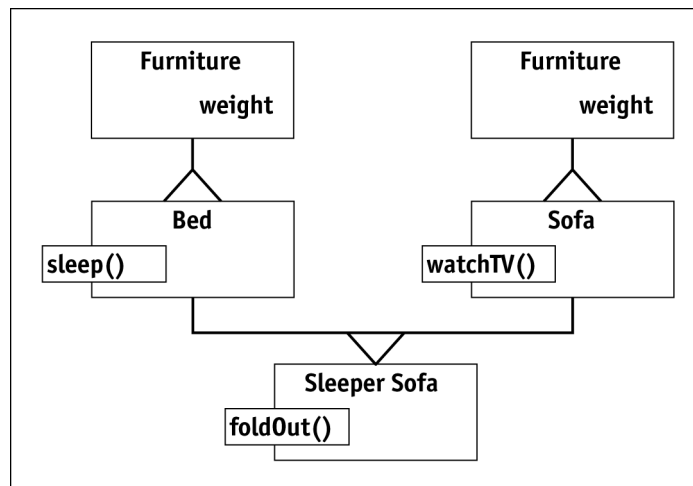


Abbildung 24.4: Tatsächliches Ergebnis des ersten Faktorisierens von `Bed` und `Sofa`.

Das ist aber Unsinn. `SleeperSofa` braucht nur eine Kopie von `Furniture`. Ich möchte, dass `SleeperSofa` nur eine Kopie von `Furniture` erbt, somit möchte ich, dass `Bed` und `Sofa` sich diese eine Kopie teilen.

Lektion 24 – Mehrfachvererbung**283**

C++ nennt das *virtuelle Vererbung*, weil sie das Schlüsselwort `virtual` verwendet.



Ich mag dieses Überladen des Begriffs `virtual` nicht, weil virtuelle Vererbung nichts mit virtuellen Funktionen zu tun hat.

Ich kehre zur Klasse `SleeperSofa` zurück, und implementiere sie wie folgt:

```
// MultipleVirtual - basiere SleeperSofa auf einer
//                      einzelnen Kopie von Furniture
// Dieses Programm kann kompiliert werden!
#include <stdio.h>
#include <iostream.h>

class Furniture
{
public:
    Furniture()
    {
        cout << »Erzeugen des Konzeptes Furniture«;
    }
    int weight;
};

class Bed : virtual public Furniture
{
public:
    Bed()
    {
        cout << »Teil Bett\n«;
    }
    void sleep()
    {
        cout << »Versuche zu schlafen\n«;
    }
    int weight;
};

class Sofa : virtual public Furniture
{
public:
    Sofa()
    {
        cout << »Teil Sofa\n«;
    }
    void watchTV()
    {
        cout << »Sehe fern\n«;
    }
    int weight;
};

// SleeperSofa - ist Bett und Sofa
```

284 **Sonntagmorgen**

```

class SleeperSofa : public Bed, public Sofa
{
public:
    // der Konstruktor muss nichts machen
    SleeperSofa()
    {
        cout << »Zusammenfügen der beiden\n«;
    }
    void foldOut()
    {
        cout << »Klappe das Bett aus\n«;
    }
};

int main()
{
    // Ausgabe des Gewichts eines Schlafsofas
    SleeperSofa ss;
    cout << »Gewicht des Schlafsofas = «
        << ss.weight    // das funktioniert!
        << »\n«;
    return 0;
}

```

Beachten Sie, dass das Schlüsselwort `virtual` bei der Vererbung von `Furniture` in `Bed` und `Sofa` eingefügt wurde. Das drückt aus »Gib mir eine Kopie von `Furniture`, wenn noch keine solche vorhanden ist, ansonsten verwende diese.« Ein `SleeperSofa` sieht im Speicher schließlich so aus:

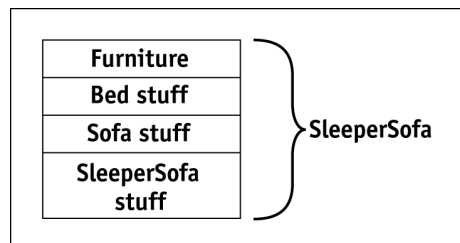


Abbildung 24.5: Speicheranordnung von `SleeperSofa` bei virtueller Vererbung.

Ein `SleeperSofa` erbt von `Furniture`, dann von `Bed` minus `Furniture`, gefolgt von `Sofa` minus `Furniture`. Dadurch werden die Elemente in `SleeperSofa` eindeutig. (Das muss nicht die Reihenfolge der Elemente im Speicher sein, das ist aber für unsere Zwecke auch nicht wichtig.)

Die Referenz in `main()` auf `weight` ist nicht länger uneindeutig, weil ein `SleeperSofa` nur eine Kopie von `Furniture` enthält. Indem von `Furniture` virtuell geerbt wird, bekommen sie die Vererbungsbeziehung wie in Abbildung 24.2.

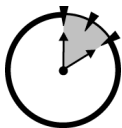
Wenn virtuelle Vererbung das Problem so schön löst, warum wird sie dann nicht immer verwendet? Dafür gibt es zwei Gründe. Erstens werden virtuell geerbte Basisklassen intern sehr verschieden von normal geerbten Klassen behandelt und diese Unterschiede schließen einen Mehraufwand ein. (Kein sehr großer Mehraufwand, aber die Erfinder von C++ waren fast paranoid, wenn es um Mehraufwand ging.) Zweitens möchten Sie manchmal zwei Kopien der Basisklasse haben (obwohl das unüblich ist).



Hinweis

Ich denke, virtuelle Vererbung sollte die Regel sein.

Als ein Beispiel für einen Fall, in dem sie keine virtuelle Vererbung haben möchten, betrachten Sie das Beispiel `TeacherAssistant`, der gleichzeitig `Student` und `Teacher` ist; beide Klassen sind Unterklassen von `Academician`. Wenn die Universität jedem `TeachingAssistant` zwei IDs gibt – eine `Student-ID` und eine `Teacher-ID` – muss die Klasse `TeacherAssistant` zwei Kopien der Klasse `Academician` enthalten.



10 Min.

24.4 Konstruktion von Objekten bei Mehrfachnennung

Die Regeln für die Konstruktion von Objekten müssen auf die Behandlung von Mehrfachvererbung ausgedehnt werden. Die Konstruktoren werden in dieser Reihenfolge aufgerufen:

- Zuerst wird der Konstruktor jeder virtuellen Basisklasse aufgerufen, in der Reihenfolge, in der von den Klassen geerbt wird.
- Dann wird der Konstruktor jeder nicht virtuellen Basisklasse aufgerufen, in der Reihenfolge, in der von den Klassen geerbt wird.
- Dann wird der Konstruktor aller Elementobjekte aufgerufen, in der Reihenfolge, in der die Elementobjekte in der Klasse erscheinen.
- Schließlich wird der Konstruktor der Klasse selber aufgerufen.
- Basisklassen werden in der Reihenfolge konstruiert, in der von ihnen geerbt wird, und nicht in der Reihenfolge in der Konstruktorzeile.

24.5 Eine Meinung dagegen

Nicht alle objektorientierten Praktiker sind der Meinung, dass Mehrfachvererbung eine gute Idee ist. Außerdem unterstützen nicht alle objektorientierten Sprachen Mehrfachvererbung. Java z.B. unterstützt keine Mehrfachvererbung – diese wird als zu gefährlich eingeschätzt und ist den damit verbundenen Ärger nicht wert.

Mehrfachvererbung ist für die Sprache nicht leicht zu implementieren.

Das ist hauptsächlich ein Compilerproblem (oder ein Problem des Compilerschreibers) und nicht das Problem des Programmierers. Mehrfachvererbung öffnet jedoch die Tür für neue Fehler. Erstens gibt es Uneindeutigkeiten, die im Abschnitt »Uneindeutigkeit bei Vererbung« erwähnt wurden. Zweitens schließt in Anwesenheit von Mehrfachvererbung das Casten eines Zeigers auf eine Unterklasse in einen Zeiger auf eine Basisklasse mysteriöse Konvertierungen ein, die unerwartete Resultate liefern können. Hier ein Beispiel:

```
#include <iostream.h>
class Base1 {int mem;};
class Base2 {int mem;};
class SubClass : public Base1, public Base2 {};

void fn(SubClass *pSC)
{
    Base1 *pB1 = (Base1*)pSC;
    Base2 *pB2 = (Base2*)pSC;
    if ((void*)pB1 == (void*)pB2)
    {
        cout << »Elemente numerisch gleich\n«;
    }
}

int main()
{
    SubClass sc;
    fn(&sc);
    return 0;
}
```

pB1 und pB2 sind numerisch nicht gleich, obwohl sie vom selben Originalwert pSC kommen. Aber die Meldung »Elemente numerisch gleich« kommt nicht. (Tatsächlich wird fn() eine Null übergeben, weil C++ diese Konvertierungen auf null nicht ausführt. Sehen Sie, wie merkwürdig das werden kann?)



0 Min.

Zusammenfassung

Ich empfehle Ihnen, Mehrfachvererbung nicht zu benutzen, bis sie C++ gut beherrschen. Einfache Vererbung stellt bereits genügend Ausdrucksstärke bereit, die genutzt werden kann. Später können sie die Handbücher studieren, bis Sie ganz sicher sind, dass Sie verstehen, was passiert, wenn Sie Mehrfachvererbung verwenden. Eine Ausnahme ist die Verwendung der Microsoft Foundation Classes (MFC), die Mehrfachvererbung nutzen. Diese Klassen wurden getestet und sind sicher. (Sie werden im Allgemeinen nicht einmal merken, dass Bibliotheken wie MFC Mehrfachvererbung nutzen.)

- Eine Klasse kann von mehr als einer Klasse erben, indem deren Klassennamen durch Kommata getrennt hinter dem »:« stehen. Obwohl in den Beispielen nur zwei Basisklassen verwendet wurden, gibt es keine Beschränkung für die Anzahl der Basisklassen. Vererbung von mehr als zwei Basisklassen ist jedoch sehr ungewöhnlich.
- Elemente, die in den Basisklassen gleich sind, sind in der Unterklasse uneindeutig. D.h. wenn beide, BaseClass1 und BaseClass2 eine Elementfunktion f() enthalten, dann ist f() uneindeutig in SubClass.
- Uneindeutigkeiten in den Basisklassen können über einen Klassenanzeiger aufgelöst werden, d.h. eine Unterklasse kann sich auf BaseClass1::f() und BaseClass2::f() beziehen.
- Wenn beide Basisklassen von einer gemeinsamen Basisklasse abgeleitet sind, in der gemeinsame Eigenschaften faktoriert sind, kann das Problem mit virtueller Vererbung gelöst werden.

Selbsttest

1. Was könnten wir als Basisklassen für eine Klasse wie `CombinationPrinterCopier` benutzen? (Ein Druck-Kopierer ist ein Laserdrucker, der auch als Kopierer verwendet werden kann.) (Siehe Einleitungsabschnitt)
2. Vervollständigen Sie die folgende Klassenbeschreibung, indem Sie die Fragezeichen ersetzen:

```
class Printer
{
public:
    int nVoltage;
    // ... weitere Elemente ...
}
class Copier
{
public:
    int nVolatage;
    // ... weitere Elemente ...
}
class CombinationPinterCopier ?????
{
    // .... Weiteres ...
}
```

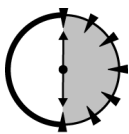
3. Was ist das Hauptproblem beim Zugriff auf `voltage` eines `CombinationPrinterCopier`-Objektes? (Siehe »Uneindeutigkeiten bei Vererbung«)
4. Gegeben, dass beide, `Printer` und `Copier`, `ElectronicEquipment` sind, was kann getan werden, um das `voltage`-Problem zu lösen? (Siehe »Virtuelle Vererbung«)
5. Nennen Sie einige Gründe, warum Mehrfachvererbung keine gute Sache sein kann. (Siehe »Eine Meinung dagegen«)



Große Programme

Checkliste

- ☒ Programme in mehrere Module teilen
- ☒ Die `#include`-Direktive verwenden
- ☒ Dateien einem Projekt hinzufügen
- ☒ Andere Kommandos des Präprozessors



30 Min.

Alle bisherigen Programme waren klein genug, um sie in eine einzige .cpp-Datei zu schreiben. Das ist für die Beispiele in einem Buch wie C++-Wochenend-Crashkurs in Ordnung, wäre aber für reale Anwendung eine ernsthafte Beschränkung. Diese Sitzung untersucht, wie ein Programm in mehrere Teile aufgeteilt werden kann durch die clevere Verwendung von Projekt- und Include-Dateien.

25.1 Warum Programme aufteilen?

Der Programmierer kann ein Programm in mehrere Dateien aufteilen, die manchmal auch *Module* genannt werden. Diese einzelnen Quelldateien werden separat kompiliert und dann im Erzeugungsprozess zu einem einzigen Programm zusammengefügt.



*Der Prozess, separat kompilierte Module zu einer ausführbaren Datei zusammenzufügen, wird als **Linken** bezeichnet.*

Es gibt mehrere Gründe dafür, ein Programm in handlichere Teile zu teilen. Erstens ermöglicht das Teilen in Module eine höhere Kapselung. Klassen mauern ihre Elemente ein, um einen gewissen Grad von Sicherheit zu erreichen. Programme können dasselbe mit Funktionen tun.



Erinnern Sie sich daran, dass Kapselung einer der Vorteile von objektorientierter Programmierung ist.

Zweitens ist ein Programm, das aus einer Anzahl gut durchdachter Module besteht, leichter zu verstehen, und damit leichter zu schreiben und zu debuggen, als ein Programm, das nur eine Quelldatei besitzt, in der alle Klassen und Funktionen enthalten sind.

Dann kommt die Wiederverwendung. Ich habe das Argument der Wiederverwendbarkeit gebraucht, um Ihnen die objektorientierte Programmierung zu verkaufen. Es ist extrem schwierig, eine einzelne Klasse zu pflegen, die in mehreren Programmen verwendet wird, wenn jedes Programm seine eigene Kopie der Klasse enthält. Es ist viel besser, wenn ein einziges Klassenmodul automatisch von den Programmen geteilt wird.

Schließlich gibt es noch ein Zeitargument. Compiler wie Visual C++ oder GNU C++ brauchen nicht sehr lange für das Kompilieren der Beispielprogramme in diesem Buch auf einem so schnellen Rechner wie dem Ihren. Kommerzielle Programme bestehen manchmal aus einigen Millionen Zeilen Quelltext. Ein solches Programm zu erzeugen kann mehr als 24 Stunden in Anspruch nehmen! (Fast so lange wie sie benötigen, dieses Buch zu lesen!) Kein Programmierer würde es hinnehmen, ein solches Programm wegen jeder kleinen Änderung neu kompilieren zu müssen. Die Zeit zum Kompilieren ist wesentlich länger als die Zeit zum Linken.

25.2 Trennung von Klassendefinition und Anwendungsprogramm

Dieser Abschnitt beginnt mit dem Beispiel EarlyBinding aus Sitzung 22, und trennt die Definition der Klasse `Student` vom Rest des Programms. Um Verwechslungen zu vermeiden, lassen Sie uns das Ergebnis `SeparatedClass` nennen.

25.2.1 Aufteilen des Programms

Wir beginnen mit der Aufteilung von `SeparatedClass` in logische Einheiten. Natürlich können die Anwendungsfunktionen `fn()` und `main()` von der Klassendefinition getrennt werden. Diese Funktionen sind weder wiederverwendbar, noch haben sie etwas mit der Definition von `Student` zu tun. In gleicher Weise hat die Klasse `Student` keine Beziehung zu den Funktionen `fn()` oder `main()`.

Ich speichere den Anwendungsteil des Programms in einer Datei `SeparatedClass.cpp`. Bis jetzt sieht das Programm so aus:

```
// SeparatedClass - demonstriert eine Anwendung
//                      getrennt von der
//                      Klassendefinition
#include <stdio.h>
#include <iostream.h>

double fn(Student& fs)
```

```

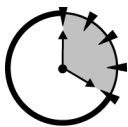
{
    // weil calcTuition() virtual deklariert ist,
    // wird der Laufzeittyp von fs verwendet, um
    // den Aufruf aufzulösen
    return fs.calcTuition();
}

int main(int nArgc, char* pszArgs[])
{
    // der folgende Ausdruck ruft
    // fn() mit einem Student-Objekt
    Student s;
    cout << »Der Wert von s.calcTuition bei\n«
         << »virtuellem Aufruf durch fn() ist »
         << fn(s)
         << »\n\n«;

    // der folgende Ausdruck ruft
    // fn() mit einem GraduateStudent-Objekt
    GraduateStudent gs;
    cout << »Der Wert von gs.calcTuition bei\n«
         << »virtuellem Aufruf durch fn() ist »
         << fn(gs)
         << »\n\n«;
    return 0;
}

```

Unglücklicherweise kann das Modul nicht erfolgreich kompiliert werden, weil nichts in `SeparatedClass.cpp` die Klasse `Student` definiert. Wir könnten natürlich die Definition von `Student` wieder in die Datei `SeparatedClass.cpp` einfügen, aber das ist nicht, was wir wollen. Wir würden damit dort hin zurückkehren, wo wir hergekommen sind.



20 Min.

25.2.2 Die `#include`-Direktive

Was benötigt wird, ist eine Methode, um die Deklaration von `Student` programmtechnisch in `SeparatedClass.cpp` einzubinden. Die `#include`-Direktive tut genau das. Die `#include`-Direktive fügt den Inhalt einer Datei, die im Quelltext benannt ist, an Stelle der `#include`-Direktive ein. Das ist schwerer zu erklären, als es in der Praxis ist.

Zuerst erzeuge ich eine Datei `student.h`, die die Definition der Klassen `Student` und `GraduateStudent` enthält:

```

// Student - definiere Eigenschaften von Student
class Student
{
public:
    virtual double calcTuition()
    {
        return 0;
    }

protected:

```

```

    int nID;
};
class GraduateStudent : public Student
{
public:
    virtual double calcTuition()
    {
        return 1;
    }

protected:
    int nGradId;
};

```



Die Zielfeile der `#include`-Direktive wird auch als Include-Datei bezeichnet. Nach Konvention tragen die Include-Dateien den Namen der Basisklasse, die sie enthalten, in Kleinbuchstaben und mit einer .h-Endung. Sie werden auch C++-Include-Dateien finden mit den Endungen .hh, .hpp und .hxx. Theoretisch kümmert sich der Compiler nicht darum.

Die neue Version der Quelldatei SeparatedClass.cpp unserer Anwendung sieht wie folgt aus:

```

// SeparatedClass - demonstriert eine Anwendung
//                  getrennt von der
//                  Klassendefinition
#include <stdio.h>
#include <iostream.h>

#include »student.h«

double fn(Student& fs)
{
    // ... von hier ab identisch mit
    // voriger Version ...
}

```

Die `#include`-Direktive wurde hinzugefügt.



Die `#include`-Direktive muss in der ersten Spalte beginnen und darf nur eine Zeile umfassen.

Wenn Sie den Inhalt von `student.h` physikalisch in die Datei `SeparatedClass.cpp` einfügen, kommen Sie zu der gleichen Datei `LateBinding.cpp`, mit der wir gestartet sind. Das ist genau das, was während des Erzeugungsprozesses passiert – C++ fügt `student.h` in `SeparatedClass.cpp` ein und kompiliert das Ergebnis.



Die `#include`-Direktive hat nicht die gleiche Syntax wie die anderen C++-Kommandos. Das liegt daran, dass es überhaupt keine C++-Direktive ist. Ein Präprozessor geht zuerst über das C++-Programm, bevor der C++-Compiler mit der Ausführung beginnt. Es ist der Präprozessor, der die `#include`-Direktive interpretiert.

25.2.3 Anwendungscode aufteilen

Das Programm `SeparatedClass` trennt die Klassendefinition erfolgreich vom Anwendungscode, aber nehmen Sie einmal an, das reicht nicht – nehmen Sie an, wir wollten die Funktion `fn( )` von `main( )` trennen. Ich könnte natürlich die gleiche Vorgehensweise anwenden und eine Datei `fn.h` erzeugen, die vom Hauptprogramm eingebunden wird.

Diese Lösung der Include-Datei löst nicht das Problem mit den Programmen, die ewig für ihre Erzeugung brauchen. Außerdem bringt die Lösung alle Probleme mit sich, die sich darum drehen, welche Funktion welche andere Funktion aufrufen kann, abhängig von den Include-Anweisungen. Eine bessere Lösung ist die Aufteilung des Quellcodes in separate Kompilierungseinheiten.

Während der Kompilierungsphase des Erzeugungsprozesses konvertiert C++ den Quelltext in den `.cpp`-Dateien in äquivalenten Maschinencode. Dieser Maschinencode wird in einer Datei gespeichert, die als Objektdatei bezeichnet wird, und die Endung `.obj` (Visual C++) oder `.o` (GNU C++) trägt. In einer anschließenden Phase, die als Linkphase bezeichnet wird, werden die Objektdateien mit der C++-Standardbibliothek zusammengefügt, um das ausführbare Programm zu bilden.

Lassen Sie uns diese Fähigkeiten zu unserem Vorteil nutzen. Wir können die Datei `SeparatedClass.cpp` in eine Datei `SeparatedFn.cpp` und eine Datei `SeparatedMain.cpp` aufteilen. Wir beginnen mit dem Anlegen dieser beiden Dateien.

Die Datei `SeparatedFn.cpp` sieht wie folgt aus:

```
// SeparatedFn - demonstriert eine Anwendung, die in
//                zwei Teile geteilt ist - der Teil
//                von fn()
#include <stdio.h>
#include <iostream.h>

#include >>student.h<<
double fn(Student& fs)
{
    // weil calcTuition() virtual deklariert ist,
    // wird der Laufzeittyp von fs verwendet, um
    // den Aufruf aufzulösen
    return fs.calcTuition();
}
```

Der Rest des Programms, die Datei `SeparatedMain.cpp`, sieht so aus:

```
// SeparatedMain - demonstriert eine Anwendung, die
//                in zwei Teile geteilt ist -
//                der Teil von main( )
#include <stdio.h>
#include <iostream.h>

#include >>student.h<<
```



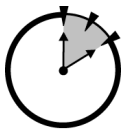
```

int main(int nArgc, char* pszArgs[])
{
    // der folgende Ausdruck ruft
    // fn() mit einem Student-Objekt
    Student s;
    cout << »Der Wert von s.calcTuition bei\n«
         << »virtuellem Aufruf durch fn() ist »
         << fn(s)
         << »\n\n«;

    // der folgende Ausdruck ruft
    // fn() mit einem GraduateStudent-Objekt
    GraduateStudent gs;
    cout << »Der Wert von gs.calcTuition bei\n«
         << »virtuellem Aufruf durch fn() ist »
         << fn(gs)
         << »\n\n«;
    return 0;
}

```

Beide Quelldateien binden die gleichen .h-Dateien ein, weil beide Zugriff auf die Definition der Klasse `Student` und der Funktionen in der C++-Standardbibliothek benötigen.



10 Min.

25.2.4 Projektdatei

Voller Erwartung öffne ich die Datei `SeparatedMain.cpp` und wähle »Build« aus.



Wenn Sie das zu Hause versuchen, stellen Sie sicher, dass sie die Projektdatei `SeparatedClass` geschlossen haben.

Eine Fehlermeldung »undeclared identifier« erscheint. C++ weiß nicht, was `fn( )` ist, wenn `SeparatedMain.cpp` kompiliert wird. Das macht Sinn, weil die Definition von `fn( )` in einer anderen Datei steht.

Natürlich muss ich eine Prototypdeklaration von `fn( )` in die Quelldatei `SeparatedMain.cpp` einfügen:

```
double fn(Student& fs);
```

Die resultierende Quelldatei lässt sich kompilieren, erzeugt aber während des Linkens einen Fehler, dass die Funktion `fn(Student)` in den .o-Dateien nicht gefunden werden kann.



Ich könnte (und sollte wahrscheinlich auch) einen Prototyp von `main( )` in die Datei `SeparatedFn.cpp` einfügen; das ist jedoch nicht nötig, weil `fn( )` die Funktion `main( )` nicht aufruft.

Was benötigt wird, ist eine Möglichkeit, C++ mitzuteilen, beide Quelldateien im gleichen Programm zusammenzubinden. Solch eine Datei wird Projektdatei genannt. Es gibt mehrere Wege, wie eine Projektdatei angelegt werden kann. Die Techniken unterscheiden sich in den zwei Compilern.

Erzeugen einer Projektdatei unter Visual C++

Für Visual C++ führen Sie die folgenden Schritte aus:

1. Stellen Sie sicher, dass Sie andere Projektdateien, die von früheren Versuchen stammen, geschlossen haben, indem Sie auf »Arbeitsbereich schließen« im Datei-Menü klicken. (Ein Arbeitsbereich ist der Name, den Microsoft für eine Sammlung von Projektdateien verwendet.)
2. Öffnen Sie die Quelldatei `SeparatedMain.cpp`. Klicken Sie auf »Compile«.
3. Wenn Visual C++ sie fragt, ob sie eine Projektdatei erzeugen möchten, antworten Sie mit Ja. Sie haben nun eine Projektdatei, die die einzelne Datei `SeparatedMain.cpp` enthält.
4. Wenn noch nicht geöffnet, öffnen sie das Workspace-Fenster (Workspace unterhalb von View klicken).
5. Schalten Sie auf FileView um. Klicken Sie auf `SeparatedMain files`, wie in Abbildung 25.1 zu sehen ist. Wählen Sie Add Files to Projekt aus. Vom Menü aus öffnen Sie die Quelldatei `SeparatedFn.cpp`. Beide Dateien `SeparatedMain.cpp` und `SeparatedFn.cpp` erscheinen nun in der Liste der Dateien, die zu dem Projekt gehören.
6. Klicken Sie Build, um das Programm erfolgreich zu erzeugen. (Das erste Mal werden beide Quelldateien kompiliert, wenn Sie Build All ausführen.)



Ich habe nicht behauptet, dies sei der eleganteste Weg. Aber es ist der einfachste.



Abbildung 25.1: Klicken Sie auf den rechten Mausbutton, um Dateien in das Projekt einzufügen.



Die Projektdatei `SeparatedMain` auf der beiliegenden CD-ROM enthält bereits beide Quelldateien.

Erzeugen einer Projektdatei unter GNU C++

Verwenden Sie die folgenden Schritte, um unter *rhide*, der GNU C++-Umgebung, eine Projektdatei zu erzeugen.

1. Ohne eine Datei offen zu haben, klicken Sie auf »Projekt öffnen« in Projekt-Menü.
2. Schreiben Sie den Namen *SeparatedMainGNU.pr* (der Name ist nicht wichtig – sie können ihn wählen wie sie möchten). Ein Projektfenster mit einem einzigen Eintrag, *<empty>*, wird am unteren Rand des Fensters geöffnet.
3. Klicken Sie auf »Add Item« unter Projekt. Öffnen Sie die Datei *SeparatedMain.cpp*.
4. Verfahren Sie mit *SeparatedFn.cpp* ebenso.
5. Wählen Sie »Make« im Compile-Menü aus, um das Programm *SeparatedMainGNU.exe* erfolgreich zu erzeugen. (Make erzeugt nur diese Dateien neu, die geändert wurden; »Neu erzeugen« erzeugt alle Dateien neu, ob sie geändert wurden oder nicht.)

Abbildung 25.2 zeigt den Inhalt des Projektfensters zusammen mit dem Meldungsfenster, das während des Erzeugungsprozesses gezeigt wird.

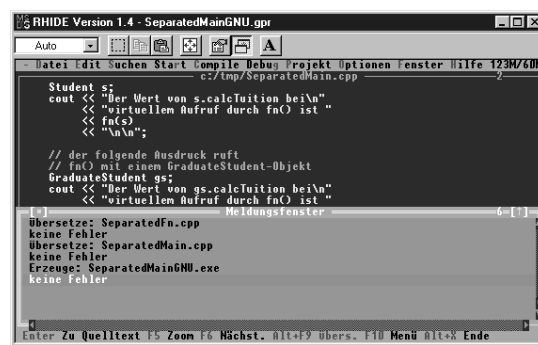


Abbildung 25.2: Die *rhide*-Umgebung zeigt die kompilierten Dateien und das erzeugte Programm an.

25.2.5 Erneute Betrachtung des Standard-Programm-Templates

Jetzt können Sie sehen, warum wir die Direktiven `#include <stdio.h>` und `#include <iostream.h>` in unseren Programmen verwendet haben. Diese Include-Dateien enthalten die Definitionen der Funktionen und Klassen, die verwendet wurden, wie z.B. `strcat( )` und `cin`.

Die von Standard-C++ definierten .h-Dateien werden durch die Klammern `<>` eingebunden, während lokal definierte .h-Dateien durch Hochkommata definiert werden. Der einzige Unterschied zwischen den beiden Schreibweisen ist, dass C++ die in Hochkommata eingeschlossenen Dateien zuerst im aktuellen Verzeichnis sucht (das Verzeichnis, das die Projektdatei enthält) und C++ die Suche für Dateinamen in Klammern im Verzeichnis der C++-Include-Dateien beginnt. Für beide Wege kann der Programmierer die zu durchsuchenden Verzeichnisse durch Einstellungen in der Projektdatei beeinflussen.

In der Tat ist es das Konzept des separaten Kompilierens, das Include-Dateien kritisch macht. Beide, *SeparatedFn* und *SeparatedMain* kennen *Student*, weil *student.h* eingebunden wurde. Wir hätten auch diese Definition in beide Quelldateien hineinschreiben können, das wäre aber sehr gefährlich gewesen. Die gleiche Definition an zwei verschiedenen Stellen schafft die Möglichkeit, dass die

beiden nicht synchronisiert werden – eine Definition könnte geändert werden und die andere nicht.

Die Definition von `Student` in eine einzige Datei zu schreiben und diese Datei in zwei Module einzubinden, macht die Entstehung verschiedener Definitionen unmöglich.

25.2.6 Handhabung von Outline-Elementfunktionen

Die Beispielsklassen `Student` und `GraduateStudent` definieren ihre Funktionen innerhalb der Klasse; die Elementfunktionen sollten aber außerhalb der Klasse deklariert sein (nur die Klasse `Student` wird gezeigt – die Klasse `GraduateStudent` ist identisch).

```
// Student - definierte Eigenschaften von Student
class Student
{
    public:
        // deklariere Elementfunktion
        virtual double calcTuition();

    protected:
        int nID;
};

// definiere den Code separat von der Klasse
double Student::calcTuition();
{
    return 0;
}
```

Ein Problem tritt auf, wenn der Programmierer beide Dateien, die Klasse und die Elementfunktionen, in die gleiche .h-Datei einzubinden versucht. Die Funktion `Student::calcTuition()` wird ein Teil von `SeparatedMain.o` und `SeparatedFn.o`. Wenn diese Dateien gelinkt werden, wird sich der C++-Linker darüber beschweren, dass `calcTuition()` zweimal definiert ist.



Wenn eine Elementfunktion innerhalb einer Klasse definiert ist, unternimmt C++ gewisse Anstrengungen, Doppeldefinitionen von Funktionen zu vermeiden. C++ kann dieses Problem nicht verhindern, wenn die Elementfunktion außerhalb der Klasse definiert ist.

Externe Elementfunktionen müssen in ihrer eigenen .cpp-Datei definiert werden, wie in der folgenden Datei `Student.cpp`:

```
#include >>student.h<<
// definiere den Code separat von der Klasse
double Student::calcTuition();
{
    return 0;
}
```

**0 Min.**

Zusammenfassung

Diese Sitzung hat Ihnen gezeigt, wie der Programmierer Programme in mehrere Quelldateien aufspalten kann. Kleinere Quelldateien sparen Zeit zur Programmgenerierung, weil der Programmierer nur die Sourcmodule kompilieren muss, die tatsächlich geändert wurden.

- Separat kompilierte Module steigern die Kapselung von Paketen ähnlicher Funktionen. Wie Sie bereits gesehen haben, sind separate, gekapselte Pakete, einfacher zu schreiben und zu debuggen. Die C++-Standardbibliothek ist ein solches gekapseltes Paket.
- Der Generierungsprozess besteht aus zwei Phasen. In der ersten, der Kompilierungsphase, werden die C++-Anweisungen in maschinenlesbare, aber unvollständige Objektdateien übersetzt. In der zweiten Phase, der Linkphase, werden diese Objektdateien zu einer einzigen ausführbaren Datei zusammengefügt.
- Deklarationen, Klassendefinitionen eingeschlossen, müssen zusammen mit jeder C++-Quelldatei kompiliert werden, die diese Funktion oder Klasse deklariert. Der einfachste Weg, dies zu bewerkstelligen, ist der verwandte Deklarationen in eine .h-Datei zu schreiben, die dann in den .cpp-Quelldateien mittels einer `#include`-Direktive eingebunden wird.
- Die Projektdatei listet die Module auf, die das Programm bilden. Die Projektdatei enthält weitere programmspezifische Einstellungen, die Einfluss darauf haben, wie die C++-Umgebung das Programm erzeugt.

Selbsttest

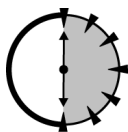
1. Wie wird eine C++-Quelldatei in eine maschinenlesbare Objektdatei überführt? Wie nennt man diesen Vorgang? (Siehe »Warum Programme aufteilen?«)
2. Wie nennt man den Prozess, der diese Objektdateien zu einer einzigen ausführbaren Datei zusammenfügt? (Siehe »Warum Programme aufteilen?«)
3. Welche Aufgaben hat die Projektdatei? (Siehe »Projektdatei«)
4. Was ist die Hauptaufgabe der `#include`-Direktive? (Siehe »Die `#include`-Direktive«)



C++-Präprozessor

Checkliste

- ☒ Häufig benutzte Konstanten mit Namen versehen
- ☒ Compilezeitmakros definieren
- ☒ Den Kompilierungsprozess kontrollieren



30 Min.

Die Programme in Sitzung 25 nutzten die `#include`-Direktive des Präprozessors, um die Definition von Klassen in mehrere Quelldateien einzubinden, die gemeinsam das Programm bildeten. In der Tat haben alle bisher gesehenen Programme `stdio.h` und `iostream.h` eingebunden, in denen Funktionen der Standardbibliothek definiert sind. In dieser Sitzung untersuchen wir die `#include`-Direktive in Verbindung mit anderen Präprozessorkommandos.

26.1 Der C++-Präprozessor

Als C++-Programmierer klicken Sie und ich auf das Build-Kommando, um den C++-Compiler anzuweisen, unseren Quellcode in ein ausführbares Programm zu übersetzen. Wir kümmern uns in der Regel nicht um die Details, wie das Kompilieren abläuft. In Sitzung 25 haben Sie gelernt, dass der Erzeugungsprozess aus zwei Teilen besteht, einer Kompilierungsphase, die jede `.cpp`-Datei in Maschinencode übersetzt, und einer Linkphase, die diese Objektdateien zusammen mit den Bibliotheksdateien von C++ zu einer ausführbaren `.exe`-Datei zusammenfasst. Was noch nicht klar wurde, ist, dass die Kompilierungsphase selber aus verschiedenen Phasen besteht.

Der Compiler operiert auf Ihren C++-Quelldateien in mehreren Schritten. Der erste Schritt findet und identifiziert alle Variablen und Klassendeklarationen, während ein weiterer Schritt den Objektcode generiert. Jeder Compiler macht so viele oder wenige Schritte wie er braucht – es gibt dafür keinen C++-Standard.

Noch vor dem ersten Compilerschritt bekommt jedoch der C++-Präprozessor eine Chance. Der C++-Präprozessor geht durch die `.cpp`-Dateien, und sucht nach Zeilen, die mit einem Doppelkreuz (`#`) in der ersten Spalte beginnen. Die Ausgabe des Präprozessors, wiederum ein C++-Programm, wird zur weiteren Bearbeitung an den Compiler übergeben.



Die Programmiersprache C nutzt den gleichen Präprozessor, so dass alles, was wir hier über den C++-Präprozessor sagen, auch für C gilt.

26.2 Die #include-Direktive

Die #include-Direktive bindet den Inhalt einer benannten Datei an ihrer Stelle ein. Der Präprozessor versucht nicht, den Inhalt der .h-Datei zu verarbeiten.

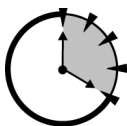


Die Include-Datei muss nicht mit .h enden, aber es kann den Programmierer und den Präprozessor verwirren, wenn sie das nicht tut.



Der Name nach dem #include-Kommando muss entweder in Hochkommata (" ") oder in Klammern (< >) stehen. Der Präprozessor nimmt an, dass Dateien, die in Hochkommata angegeben werden, benutzerdefiniert sind, und daher im aktuellen Verzeichnis stehen. Nach Dateien in Klammern sucht der Präprozessor in den Verzeichnissen des C++-Compilers.

Die Include-Datei sollte keine C++-Funktionen enthalten, weil sie separat durch das Modul expandiert und kompiliert werden, das die Datei einbindet. Der Inhalt der Include-Datei sollte auf die Klassendefinition, Definition von globalen Variablen und andere Präprozessor-Direktiven beschränkt sein.



20 Min.

26.3 Die Direktive #define

Die Direktive #define definiert eine Konstante oder ein Makro. Das folgende Beispiel zeigt, wie #define zur Definition einer Konstanten gebraucht wird:

```
#define MAX_NAME_LENGTH 256

void fn(char* pszSourceName)
{
    char szLastName[MAX_NAME_LENGTH];

    if (strlen(pszSourceName) >= MAX_NAME_LENGTH)
    {
        // ... Zeichenkette zu lang -
        // Fehlerbehandlung ...
    }

    // ... hier geht es weiter ...
}
```

Die Präprozessor-Direktive definiert einen Parameter `MAX_NAME_LENGTH`, der zur Compilezeit durch den konstanten Wert 256 ersetzt wird. Der Präprozessor ersetzt den Namen `MAX_NAME_LENGTH` durch die Konstante 256 überall, wo sie benutzt wird. Überall dort, wo wir `MAX_NAME_LENGTH` sehen, sieht der Compiler 256.



Das Beispiel demonstriert die Namenskonvention für `#define`-Konstanten. Namen werden in Großbuchstaben mit Unterstrichen zur Trennung geschrieben.

Wenn sie auf diese Weise verwendet wird, ermöglicht es die `#define`-Direktive dem Programmierer, konstante Werte mit aussagekräftigen Namen zu versehen; `MAX_NAME_LENGTH` sagt dem Programmierer mehr als 256. Konstanten auf diese Weise zu definieren, macht Programme leichter modifizierbar. Z.B. kann die maximale Anzahl Zeichen in einem Namen programmweit limitiert sein. Diesen Wert von 256 auf 128 zu ändern ist einfach, indem nur das `#define`-Kommando geändert werden muss, unabhängig davon, an wie vielen Stellen die Konstante verwendet wird.

26.3.1 Definition von Makros

Die `#define`-Direktive erlaubt auch Definitionsmakros – eine Compilezeit-Direktive, die Argumente enthält. Das Folgende demonstriert die Definition und die Verwendung des Makros `square( )`, das den Code generiert, um das Quadrat ihres Argumentes zu berechnen.

```
#define square(x) x * x
void fn()
{
    int nSquareOfTwo = square(2);
    // ... usw. ...
}
Der Präprozessor macht hieraus:
void fn()
{
    int nSquareOfTwo = 2 * 2;
    // ... usw. ...
}
```

26.3.2 Häufige Fehler bei der Verwendung von Makros

Der Programmierer muss sehr vorsichtig sein bei der Verwendung von `#define`-Makros. Z.B. erzeugt das Folgende nicht das erwartete Ergebnis:

```
#define square(x) x * x
void fn()
{
    int nSquareOfTwo = square(1 + 1);
}
```

Der Präprozessor generiert hieraus :

```
void fn()
```


Lektion 26 – C++-Präprozessor II 301

```
{
    int nSquareOfTwo = 1 + 1 * 1 + 1;
}
```

Weil Multiplikation Vorrang vor Addition hat, wird der Ausdruck interpretiert, als wenn er so geschrieben wäre:

```
void fn()
{
    int nSquareOfTwo = 1 + (1 * 1) + 1;
}
```

Der Ergebniswert von `nSquareOf` ist 3 und nicht 4.

Eine vollständige Qualifizierung des Makros durch großzügige Verwendung von Klammern schafft Abhilfe, weil Klammern die Reihenfolge der Auswertung kontrollieren. Mit einer Definition von `square` in der folgenden Weise gibt es keine Probleme:

```
#define square(x) ((x) * (x))
```

Doch auch das löst das Problem nicht in jedem Falle. Z.B. kann das Folgende nicht zum Laufen gebracht werden:

```
#define square(x) ((x) * (x))
void fn()
{
    int nV1 = 2;
    int nV2;
    nV2 = square(nV1++);
}
```

Sie können erwarten, dass `nV2` den Ergebniswert 4 statt 6 und `nV1` den Wert 3 statt 4 erhält wegen der folgenden Expansion des Makros:

```
void fn()
{
    int nV1 = 2;
    int nV2;
    nV2 = nV1++ * nV1++;
}
```

Makros sind nicht typsicher. Das kann in Ausdrücken mit unterschiedlichen Typen zu Verwirrungen führen:

```
#define square(x) ((x) * (x))
void fn()
{
    int nSquareOfTwo = square(2.5);
}
```

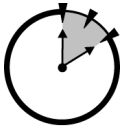
Weil `nSquareOfTwo` ein `int` ist, könnten Sie erwarten, dass der Ergebniswert 4 statt dem tatsächlichen Wert 6 ($2.5 * 2.5 = 6.25$) ist.

C++-Inline-Funktionen vermeiden das Problem mit den Makros.

302 **Sonntagmorgen**

```
inline int square(int x) {return x * x;}
void fn()
{
    int nV1 = square(1 + 1); // nV1 ist 4
    int nV2;
    nV2 = square(nV1++)      // nV2 ist 4, nV1 is 3
    int nV3 = square(2.5)    // nV3 ist 4
}
```

Die Inline-Version von `square( )` erzeugt nicht mehr Code als ein Makro, hat aber nicht die Nachteile der Präprozessor-Variante.



10 Min.

26.4 Compilerkontrolle

Der Präprozessor stellt auch Möglichkeiten bereit, den Compilevorgang zu steuern.

26.4.1 Die `#if`-Direktive

Die Präprozessor-Direktive, die C++ am ähnlichsten ist, ist die `#if`-Anweisung. Wenn der konstante Ausdruck nach dem `#if` nicht gleich null ist, werden alle Anweisungen bis zu `#else` an den Compiler übergeben. Wenn der Ausdruck null ist, werden die Anweisungen zwischen `#else` und `#endif` übergeben. Der `#else`-Zweig ist optional. Z.B.:

```
#define SOME_VALUE 1
#if SOME_VALUE
int n = 1;
#else
int n = 2;
#endif
```

wird konvertiert in

```
int n = 1;
```

Ein paar Operatoren sind für den Präprozessor definiert, z.B.

```
#define SOME_VALUE 1
#if SOME_VALUE - 1
int n = 1;
#else
int n = 2;
#endif
wird konvertiert in:
int n = 2;
```



Tipp

Denken Sie daran, dass dies Entscheidungen zur Compilezeit sind und keine Laufzeitentscheidungen. Die Ausdrücke nach `#if` enthalten Konstanten und `#define`-Direktiven – Variablen und Funktionsaufrufe sind nicht erlaubt.

26.4.2 Die #ifdef-Direktive

Eine andere Kontrolldirektive des Präprozessors ist `#ifdef`. Das `#ifdef` ist wahr, wenn die Konstante danach definiert ist. Somit wird das Folgende

```
#define SOME_VALUE 1
#ifdef SOME_VALUE
    int n = 1;
#else
    int n = 2;
#endif
```

konvertiert in:

```
int n = 1;
```

Die Direktive

```
#ifdef SOME_VALUE
    int n = 1;
#else
    int n = 2;
#endif
```

jedoch wird konvertiert in

```
int n = 2;
```

Das `#ifndef` ist auch definiert mit der genau umgekehrten Definition.

Verwendung von #ifdef/#ifndef zur Einschlusskontrolle

Der häufigste Einsatz von `#ifdef` ist die Kontrolle über die Einbeziehung von Code. Ein Symbol kann nicht mehr als einmal definiert werden. Das Folgende ist ungültig:

```
class MyClass
{
    int n;
};
class MyClass
{
    int n;
};
```

Wenn `MyClass` in der Include-Datei `myclass.h` definiert ist, wäre es ein Fehler, diese Datei zweimal in die `.cpp`-Quelldatei einzubinden. Sie könnten denken, dass dieses Problem leicht vermeidbar ist. Es ist aber üblich, dass Include-Dateien andere Include-Dateien einbinden, wie in folgendem Beispiel:

```
#include >>myclass.h<<
class mySpecialClass : public MyClass
{
    int m;
}
```

304 Sonntagmorgen

Ein nichtsahnender Programmierer könnte leicht beide, `ass.h` und `myspecialclass.h`, in dieselbe Quelldatei einbinden, was durch die doppelte Definition zu einem Compilerfehler führt.

```
// das kann nicht kompiliert werden
#include >myclass.h<<
#include >myspecialclass.h<<
void fn(MyClass& mc)
{
    // ... kann ein Objekt aus der Klasse MyClass
    // oder MySpecialClass sein
}
```

Dieses spezielle Beispiel lässt sich leicht korrigieren. In einer großen Anwendung können die Beziehungen zwischen den Include-Dateien viel komplexer sein.

Der wohlüberlegte Einsatz der `#ifndef`-Direktive verhindert dieses Problem, indem `myclass.h` wie folgt geschrieben wird:

```
#ifndef MYCLASS_H
#define MYCLASS_H
class MyClass
{
    int n;
};
#endif
```

Wenn `myclass.h` zum ersten Mal eingebunden wird, ist `MYCLASS_H` nicht definiert und `#ifndef` ist wahr. Die Konstante `MYCLASS_H` wird jedoch innerhalb von `myclass.h` definiert. Das nächste Mal, wenn `myclass.h` während des Kompilierens angefasst wird, ist `MYCLASS_H` definiert, und die Klassendefinition wird weggelassen.

Debug-Code und #ifndef

Ein anderer häufiger Einsatz von `#ifndef` ist die Integration von Debug-Code zur Compilezeit. Betrachten Sie z.B. die folgende Debug-Funktion:

```
void dumpState(MySpecialClass& msc)
{
    cout <<>>MySpecialClass:<<
        <<>>m = >> <<msc.m
        <<>>n = >> <<msc.n;
}
```

Jedes Mal, wenn diese Funktion aufgerufen wird, druckt `dumpState()` den Inhalt des `MySpecialClass`-Objektes in die Standardausgabe. Ich kann überall in meinem Programm Aufrufe dieser Funktion einbauen, um den Zustand von `MySpecialClass`-Objekten zu kontrollieren. Wenn das Programm fertig ist, muss ich alle diese Aufrufe wieder entfernen. Das ist nicht nur ermüdend, sondern birgt das Risiko in sich, dass hierbei neue Fehler in das System gelangen. Außerdem kann es sein, dass ich die Anweisungen zum erneuten Debuggen des Systems wieder benötige.

Ich könnte eine Art Flag definieren, das steuert, ob das Programm den Status der `MySpecialClass`-Objekte ausgibt. Aber die Aufrufe selber stellen einen Mehraufwand dar, der die Funktionen verlangsamt. Ein besserer Ansatz ist der folgende:

Lektion 26 – C++-Präprozessor II 305

```
#ifdef DEBUG
void dumpState(MySpecialClass& msc)
{
    cout <<>>MySpecialClass:<<
        <<>>m = >> <<msc.m
        <<>>n = >> <<msc.n;
}
#else
inline dumpState(MySpecialClass& mc)
{
}
#endif
```



0 Min.

Wenn der Parameter `DEBUG` definiert ist, wird die Funktion `dumpState( )` kompiliert. Wenn `DEBUG` nicht definiert ist, wird eine Inline-Version von `dumpState( )` kompiliert, die nichts tut. Der C++-Compiler konvertiert jeden Aufruf dieser Funktion zu nichts.



Tipp

Wenn die inline-Version der Funktion nicht funktioniert, liegt das vielleicht daran, dass der Compiler keine Inline-Funktionen unterstützt. Dann verwenden Sie die folgende Makrodefinition:

```
#define dumpState(x)
```

Visual C++ und GNU C++ unterstützen beide diesen Ansatz. Konstanten können in den Projekteinstellungen ohne Hinzufügen der `#define`-Direktive in den Sourcecode definiert werden. In der Tat ist die Konstante `_DEBUG` automatisch definiert, wenn im Debug-Modus kompiliert wird.

Zusammenfassung

Der häufigste Einsatz des Präprozessors ist das Einbinden der gleichen Klassendefinition oder der gleichen Funktionsprototypen in mehrere .cpp-Quelldateien mit der `#include`-Direktive. Die Präprozessor-Direktiven `#if` und `#ifdef` erlauben die Kontrolle darüber, welche Zeilen des Codes kompiliert werden und welche nicht.

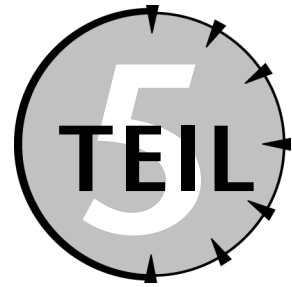
- Der Name der Datei, die mittels `#include` eingebunden wird, sollte mit `.h` enden. Das nicht zu tun, verwirrt andere Programmierer und vielleicht sogar den Compiler. Dateinamen, die in Hochkommata ("") eingeschlossen sind, werden im aktuellen (oder in einem anderen benutzerdefinierten) Verzeichnis gesucht, wohingegen Klammern (<>) zur Referenzierung von Include-Dateien von C++ verwendet wird.
- Wenn der konstante Ausdruck nach der `#if`-Direktive nicht null ist, dann werden die folgenden C++-Anweisungen an den Compiler übergeben; andernfalls werden sie nicht übergeben. Die Direktive `#ifdef x` ist wahr, wenn die `#define`-Konstante `x` definiert ist.
- Alle Präprozessor-Direktiven kontrollieren, welche C++-Anweisungen der Compiler »sieht«. Alle werden zur Compilezeit ausgewertet und nicht zur Laufzeit.

306 **Sonntagmorgen**

Selbsttest

1. Was ist der Unterschied zwischen `#include "file.h"` und `#include <file.h>`? (Siehe »Die `#include`-Direktive«)
2. Was sind die beiden Typen der `#define`-Direktive? (Siehe »Die `#define`-Direktive«)
3. Gegeben die Makrodefinition `#define square(x) x * x`, was ist der Wert von `square(2 + 3)`? (Siehe »Die `#define`-Direktive«)
4. Nennen Sie einen Vorteil von Inline-Funktionen gegenüber einer äquivalenten Makrodefinition. (Siehe »Häufige Fehler bei der Verwendung von Makros«)
5. Was ist ein häufiger Einsatz der `#ifndef`-Direktive? (Siehe »Verwendung von `#ifdef`/`#ifndef` zur Einschlusskontrolle«)

Sonntagmorgen – Zusammenfassung



1. Wie sieht die Ausgabe des folgenden Programms aus?

```
// ConstructionTest - zeigt die Reihenfolge, in der
//                     Objekte konstruiert werden
#include <stdio.h>
#include <iostream.h>
#include <string.h>

// Advisor - eine leere Klasse
class Advisor
{
public:
    Advisor(char* pszName)
    {
        cout << »Advisor:<<
            << pszName
            << »\n<<;
    }
};

class Student
{
public:
    Student() : adv(»Student Datenelement<<)
    {
        cout << »Student\n<<;
        new Advisor(»Student lokal<<);
    }
    Advisor adv;
};

class GraduateStudent : public Student
{
public:
    GraduateStudent() :
        adv(»GraduateStudent Datenelement<<)
    {
        cout << » GraduateStudent\n<<;
        new Advisor(»GraduateStudent lokal<<);
    }

protected:
    Advisor adv;
};

int main(int nArgc, char* pszArgs[])
```

```

{
    GraduateStudent gs;
    return 0;
}

```

2. **Gegeben sei, dass ein GraduateStudent einen Grad von 2.5 oder besser erreichen muß, um zu bestehen, und 1.5 für reguläre Studenten ausreicht. Schreiben Sie eine Funktion `pass( )` unter Verwendung der Klassen, die wir in dieser Sitzung geschrieben haben, die einen Grad akzeptiert, und 0 zurückgibt für einen Studenten, der durchgefallen ist, und 1, wenn er bestanden hat.**
3. **Schreiben Sie eine Klasse `Checking` (Girokonto), die von `Account` und `CashAccount` wie oben gezeigt erbt. Ein Girokonto ist einem Sparkonto sehr ähnlich, außer dass eine Gebühr für jedes Abheben anfällt. Machen Sie sich keine Gedanken zu Überziehungen.**



Wenn Sie Zeit sparen möchten, können Sie die Klassen `Account`, `CashAccount` und `Savings` aus dem Verzeichnis `ExerciseClasses` der beiliegenden CD-ROM kopieren. Sie können diese als Startpunkt verwenden.

Testen Sie Ihre Klasse mit dem Folgenden:

```

void fn(Account* pAccount)
{
    pAccount->deposit(100);
    pAccount->withdrawal(50);
}

int main(int nArgc, char* pszArgs[])
{
    // eröffne ein Sparkonto
    Savings savings(1234, 0);
    fn(&savings);

    // und nun ein Girokonto
    Checking checking(5678, 0);
    fn(&checking);
    // Ausgabe des Ergebnisses
    cout << »Kontostand Sparkonto ist »
         << savings.balance()
         << »\n«;
    cout << »Kontostand Girokonto ist »
         << checking.balance()
         << »\n«;

    return 0;
}

```


Hinweise: Die Ausgabe des Programms sollte so aussehen:

```
Kontostand Sparkonto ist 50  
Kontostand Girokonto ist 49
```

4. **Schreiben Sie ein Programm, das mit Mehrfachvererbung ein Objekt `pc` aus der Klasse `CombinationPrinterCopier` erzeugt. Dieses Programm sollte unter Verwendung von `pc` drucken. Zusätzlich sollte das Programm die elektrische Spannung (voltage) von `pc` ausgeben.**

Hinweise:

- a. Suchen Sie zur Hilfe nach Worten in Anführungszeichen.
- b. Sie sind nicht in der Lage, die Spannung durch die Konstruktoren nach oben weiterzugeben. Setzen Sie stattdessen die voltage im Konstruktor `CombinationPrinterCopier`.

Sonntag- nachmittag

Teil 6

Lektion 27

Überladen von Operatoren

Lektion 28

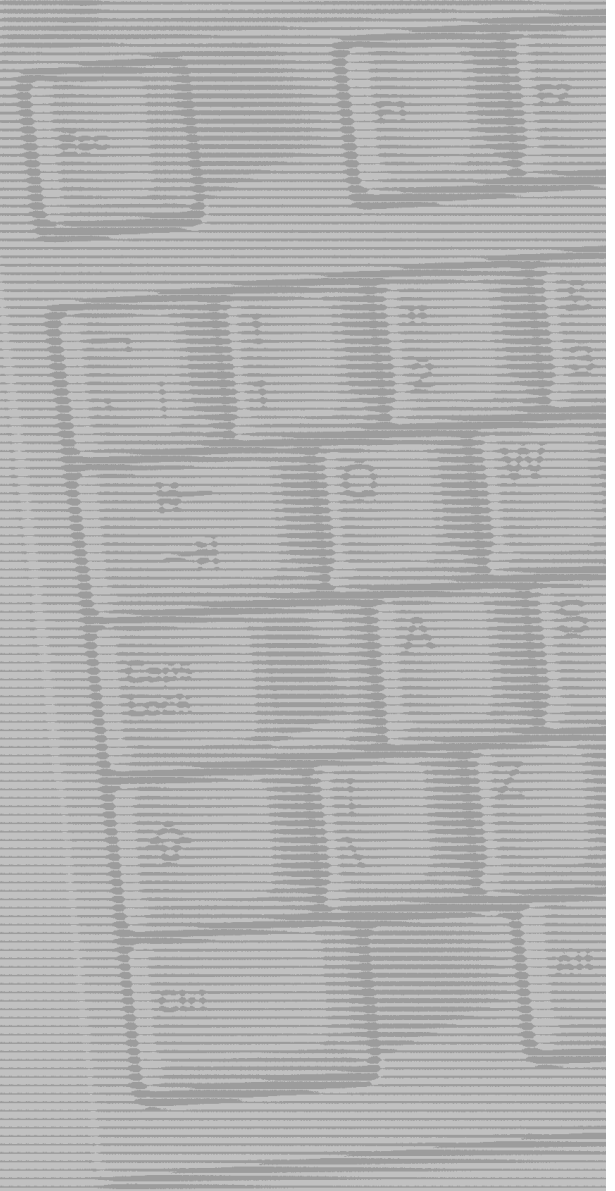
Der Zuweisungsoperator

Lektion 29

Stream-I/O

Lektion 30

Ausnahmen

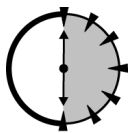


Überladen von Operatoren



Checkliste

- ☒ Überladen von C++-Operatoren im Überblick
- ☒ Diskussion Operatoren und Funktionen
- ☒ Implementierung von Operatoren als Elementfunktion und Nichtelementfunktion
- ☒ Der Rückgabewert eines überladenen Operators
- ☒ Ein Spezialfall: Der Cast-Operator



30 Min.

Die Sitzungen 6 und 7 diskutierten die mathematischen und logischen Operatoren, die C++ für die elementaren Datentypen definiert.

Die *elementaren* Datentypen sind die, die in die Sprache eingebaut sind, wie `int`, `float`, `double` usw. und die Zeigertypen.

Zusätzlich zu den elementaren Operatoren erlaubt es C++ dem Programmierer, Operatoren für Klassen zu definieren, die der Programmierer geschrieben hat. Das wird *Überladen* von Operatoren genannt.

Normalerweise ist das Überladen von Operatoren optional und sollte nicht von C++-Anfängern probiert werden. Eine Menge erfahrener C++-Programmierer denken, dass das Überladen von Operatoren keine so tolle Sache ist. Es gibt jedoch drei Operatoren, deren Überladen Sie erlernen müssen: Zuweisung (`=`), Linksshift (`<<`) und Rechtsshift (`>>`). Sie sind wichtig genug, um ein eigenes Kapitel zu bekommen, das diesem Kapitel unmittelbar folgt.



Tipp

Das Überladen von Operatoren kann Fehler verursachen, die schwer zu finden sind. Seien Sie ganz sicher, wie das Überladen von Operatoren funktioniert, bevor Sie es einsetzen.

312 Sonntagnachmittag**27.1 Warum sollte ich Operatoren überladen?**

C++ betrachtet benutzerdefinierte Typen, wie `Student` und `Sofa`, als genauso gültig wie die elementaren Typen, z.B. `int` und `char`. Wenn Operatoren für elementare Datentypen definiert sind, warum sollten sie nicht auch für benutzerdefinierte Typen definiert werden können?

Das ist ein schwaches Argument, aber ich gebe zu, dass das Überladen von Operatoren seinen Nutzen hat. Betrachten Sie die Klasse `USDollar`. Einige Operatoren machen keinen Sinn, wenn sie auf `Dollars` angewendet werden. Was soll z.B. das Invertieren eines `Dollars` bedeuten? Auf der anderen Seite sind einige Operatoren definitiv anwendbar. So macht es z.B. Sinn, einen `USDollar` zu einem `USDollar` zu addieren oder davon zu subtrahieren. Es macht Sinn, `USDollar` mit `double` zu multiplizieren. Es macht jedoch keinen Sinn, `USDollar` mit `USDollar` zu multiplizieren.

Das Überladen von Operatoren kann die Lesbarkeit verbessern. Betrachten sie das Folgende, zunächst ohne überladene Operatoren:

```
//expense - berechne bezahlten Betrag
//          (Hauptsumme und Einzelrate)
USDollar expense(USDollar principal, double rate)
{
    // berechne den Ratenbetrag
    USDollar interest = principal.interest(rate);

    // addiere zur Hauptsumme und gib sie zurück
    return principal.add(interest);
}
```

Wenn der entsprechende Operator überladen wird, sieht die gleiche Funktion wie folgt aus:

```
//expense - berechne bezahlten Betrag
//          (Hauptsumme und Einzelrate)
USDollar expense(USDollar principal, double rate)
{
    USDollar interest = principal * rate;
    return principal + interest;
}
```

Bevor wir untersuchen, wie Operatoren überladen werden, müssen wir die Beziehung zwischen Operatoren und Funktionen verstehen.

27.2 Was ist die Beziehung zwischen Operatoren und Funktionen?

Ein Operator ist nichts anderes, als eine eingebaute Funktion mit einer eigenen Syntax. Z.B. hätte der Operator `+` genauso gut als `add( )` geschrieben werden können.

C++ gibt jedem Operator einen funktionsähnlichen Namen. Der funktionale Name des Operators ist das Operatorsymbol, vor dem das Schlüsselwort `operator` steht, gefolgt von den entsprechenden Argumenttypen. Der Operator `+` z.B., der ein `int` zu einem anderen `int` addiert und daraus ein `int` erzeugt, heißt `int operator+(int, int)`.

Lektion 27 – Überladen von Operatoren 313

Der Programmierer kann alle Operatoren überladen, außer `.`, `::`, `*` (Dereferenzierung) und `&`, durch Überladen ihres funktionalen Namens mit den folgenden Einschränkungen:

- Der Programmierer kann keine neuen Operatoren einführen. Sie können keinen Operator `x $ y` einführen.
- Das Format der Operatoren kann nicht geändert werden. Somit können Sie keinen Operator `%i` definieren, weil `%` ein binärer Operator ist.
- Der Vorrang der Operatoren kann nicht geändert werden. Ein Programm kann nicht erzwingen, dass `operator+` vor dem `operator*` ausgeführt wird.
- Schließlich können Operatoren nicht neu definiert werden, wenn sie auf elementare Typen angewendet werden. Existierende Operatoren können nur für neue Typen überladen werden.

27.3 Wie funktioniert das Überladen von Operatoren?

Lassen Sie uns das Überladen von Operatoren in Aktion sehen. Listing 27-1 zeigt eine Klasse `USDollar`, die einen Additionsoperator und einen Inkrementoperator definiert.

Listing 27-1: USDollar mit überladenen Operatoren für Addition und Inkrementierung

```
// USDollarAdd - demonstriert die Definition und die
//               Verwendung des Additionsoperators
//               für die Klasse USDollar
#include <stdio.h>
#include <iostream.h>

// USDollar - repräsentiert den US Dollar
class USDollar
{
    // stelle sicher, dass die benutzerdefinierten
    // Operatoren Zugriff auf die protected-Elemente
    // der Klasse haben
    friend USDollar operator+(USDollar&, USDollar&);
    friend USDollar& operator++(USDollar&);

public:
    // konstruiere ein Dollarobjekt mit initialen
    // Werten für Dollar und Cents
    USDollar(int d = 0, int c = 0);

    // rationalize - normalisiere den Centbetrag
    //               durch Addition eines Dollars pro
    //               100 Cents
    void rationalize()
    {
        dollars += (cents / 100);
        cents   %= 100;
    }

    // output - schreibe den Wert des Objektes
    //           in die Standardausgabe
    void output()
```

314 *Sonntagnachmittag*

```

    {
        cout << »$«
            << dollars
            << ».«
            << cents;
    }

protected:
    int dollars;
    int cents;
};

USDollar::USDollar(int d, int c)
{
    // speichere die initialen Werte
    dollars = d;
    cents = c;

    rationalize();
}

//operator+ - addiere s1 zu s2 und gib das Ergebnis
//              in einem neuen Objekt zurück
USDollar operator+(USDollar& s1, USDollar& s2)
{
    int cents  = s1.cents  + s2.cents;
    int dollars = s1.dollars + s2.dollars;
    return USDollar(dollars, cents);
}

//operator++ - inkrementiere das Argument; ändere
//              den Wert dieses Objektes
USDollar& operator++(USDollar& s)
{
    s.cents++;
    s.rationalize();
    return s;
}

int main(int nArgc, char* pszArgs[])
{
    USDollar d1(1, 60);
    USDollar d2(2, 50);
    USDollar d3(0, 0);

    // zuerst ein binärer Operator
    d3 = d1 + d2;
    d1.output();
    cout << » + »;
    d2.output();
    cout << » = »;
    d3.output();
    cout << »\n«;

    // jetzt ein unärer Operator

```

Lektion 27 – Überladen von Operatoren 315

```

++d3;
cout << »Nach Inkrementierung gleich »;
d3.output();
cout << »\n«;
return 0;
}

```

Die Klasse `USDollar` ist so definiert, dass sie einen ganzzahligen Dollarbetrag und einen ganzzahligen Centbetrag kleiner 100 speichert. Beim Durcharbeiten der Klasse von vorne nach hinten, sehen wir die Operatoren `operator+( )` und `operator++( )`, die als Freunde der Klasse deklariert sind.



Erinnern Sie sich daran, dass ein Klassenfreund eine Funktion ist, die Zugriff auf die protected-Elemente der Klasse hat. Weil `operator+( )` und `operator++( )` als herkömmliche Nichtelementfunktionen implementiert sind, müssen sie als Freunde der Klasse deklariert sein, um Zugriff auf die protected-Elemente der Klasse zu erhalten.

Der Konstruktor von `USDollar` erzeugt ein Objekt aus den ganzzahligen Angaben von Dollars und Cents, für die es beide Defaultwerte gibt. Einmal gespeichert ruft der Konstruktor die Funktion `rationalize( )` auf, die den Betrag normalisiert, indem Cent-Anzahlen größer als 100 dem Dollarbetrag zugeschlagen werden. Die Funktion `output( )` schreibt das `USDollar`-Objekt nach `cout`.

Der `operator+( )` wurde mit zwei Argumenten definiert, weil Addition ein binärer Operator ist (d.h. der zwei Argumente hat). Der `operator+( )` beginnt damit, die Beträge von Dollar und Cent ihrer beiden `USDollar`-Argumente zu addieren. Sie erzeugt dann ein neues `USDollar`-Objekt mit diesen Werten und gibt es an den Aufrufenden zurück.



Jede Operation auf einem Wert eines `USDollar`-Objekts sollte `rationalize( )` aufrufen, um sicherzustellen, dass der Centbetrag nicht größer oder gleich 100 ist. Die Funktion `operator+( )` ruft `rationalize( )` über den Konstruktor `USDollar` auf.

Der Inkrementoperator `operator++( )` hat nur ein Argument. Diese Funktion inkrementiert die Anzahl Cents im Objekt `s` um eins. Die Funktion gibt dann eine Referenz auf das Objekt zurück, das gerade inkrementiert wurde.

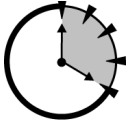
Die Funktion `main( )` zeigt die Summe der beiden Dollarbeträge an. Sie inkrementiert dann das Ergebnis der Addition, und zeigt dieses an. Die Ausführung von `USDollarAdd` sieht wie folgt aus:

```

$1.60 + $2.50 = $4.10
Nach Inkrementierung gleich $4.11

```

Im Gebrauch sehen die Operatoren sehr natürlich aus. Was könnte einfacher sein, als `d3 = d1 + d2` oder `++d3`?



20 Min.

27.3.1 Spezielle Überlegungen

Es gibt ein paar spezielle Überlegungen, die Sie beim Überladen von Operatoren anstellen sollten. Erstens folgert C++ nichts für die Beziehung zwischen Operatoren. Somit hat der Operator `operator+=( )` nichts zu tun mit `operator+( )` oder `operator=( )`.

Zusätzlich führt `operator+(USDollar&, USDollar&)` nicht zwingend eine Addition durch. Sie könnten `operator+( )` auch etwas anderes ausführen lassen; es ist aber eine WIRLICH SCHLECHTE IDEE, das zu tun. Die Leute sind daran gewöhnt, wie sich die Operatoren verhalten. Sie wollen nicht, dass die Operatoren andere Dinge tun.

Ursprünglich war es nicht möglich, den Präfixoperator `++x` unabhängig von Postfixoperator `x++` zu überladen. Genügend Programmierer haben sich darüber beklagt, so dass die Regel aufgestellt wurde, dass `operator++(ClassName)` den Präfixoperator meint und `operator++(ClassName, int)` den Postfixoperator meint; das zweite Argument wird immer mit null belegt. Die gleiche Regel gilt für `operator-( )`.

Wenn Sie nur einen der beiden für `operator++( )` oder `operator-( )` angeben, wird er für beide, Präfix und Postfix, verwendet. Der C++-Standard besagt, dass ein Compiler das nicht tun muss, aber die meisten Compiler tun es.

27.4 Ein detaillierterer Blick

Warum erfolgt bei `operator+( )` eine Wertrückgabe, aber bei `operator++( )` kommt eine Referenz zurück? Das ist kein Zufall, sondern ein sehr wichtiger Unterschied.

Die Addition zweier Objekte verändert keines der Objekte. D.h. `a + b` verändert weder `a` noch `b`. Der `operator+( )` muss daher ein temporäres Objekt erzeugen, in dem er das Ergebnis der Addition speichern kann. Das ist der Grund, weshalb `operator+( )` ein Objekt konstruiert und dieses Objekt als Wert zurückgibt.

Insbesondere würde das Folgende nicht funktionieren:

```
// das funktioniert nicht
USDollar& operator+(USDollar& s1, USDollar& s2)
{
    s1.cents += s2.cents;
    s1.dollars += s2.dollars;
    return s1;
}
```

weil hierbei `s1` verändert wird. Nach einer Addition `s1 + s2` wäre der Wert von `s1` verändert. Das Folgende funktioniert auch nicht:

```
// das funktioniert auch nicht
USDollar& operator+(USDollar& s1, USDollar& s2)
{
    int cents = s1.cents + s2.cents;
    int dollars = s1.dollars + s2.dollars;
    USDollar result(dollars, cents);
    return result;
}
```


Lektion 27 – Überladen von Operatoren 317

Obwohl das ohne Probleme kompiliert werden kann, erzeugt es ein falsches Ergebnis. Das Problem ist, dass die zurückgegebene Referenz `result` auf ein Objekt verweist, deren Gültigkeitsbereich lokal in der Funktion ist. Somit hat `result` seinen Gültigkeitsbereich bereits verlassen, wenn es von der aufrufenden Funktion verwendet werden kann.

Warum dann nicht einfach einen Speicherbereich vom Heap allozieren?

```
// das funktioniert
USDollar& operator+(USDollar& s1, USDollar& s2)
{
    int cents = s1.cents + s2.cents;
    int dollars = s1.dollars + s2.dollars;
    return *new USDollar(dollars, cents);
}
```

Das wäre gut, außer, dass es keinen Mechanismus gibt, den allozierten Speicher wieder an den Heap zurückzugeben. Solche Speicherlöcher sind schwer zu finden. Ganz langsam geht bei jeder Addition dem Heap ein wenig Speicher verloren.

Die Wertrückgabe zwingt den Compiler, selber ein eigenes temporäres Objekt anzulegen, und es auf den Stack des Aufrufenden zu packen. Das Objekt, das in der Funktion erzeugt wurde, wird dann in das Objekt kopiert, als Teil von `operator+( )`. Aber wie lange existiert das temporäre Objekt von `operator+( )`? Ein temporäres Objekt muss so lange gültig bleiben, bis der »erweiterte Ausdruck«, in dem es vorkommt, fertig ist. Der erweiterte Ausdruck ist alles bis zum Semikolon.

Betrachten Sie z.B. den folgenden Schnipsel:

```
SomeClass f();
LotsAClass g();
void fn()
{
    int i;
    i = f() + (2 * g());

    // ... die temporären Objekte, die f() und g()
    // zurückgeben, sind hier bereits ungültig ...
}
```

Das temporäre Objekt, das von `f( )` zurückgegeben wird, existiert weiter, während `g( )` aufgerufen wird und die Multiplikation durchgeführt wird. Dieses Objekt verliert beim Semikolon seine Gültigkeit.

Um zu unserem `USDollar`-Beispiel zurückzukehren, in dem das temporäre Objekt nicht gesichert wird, funktioniert das Folgende nicht:

```
d1 = d2 + d3 + ++d4;
```

Das temporäre Ergebnis aus der Addition von `d2` und `d3` muss gültig bleiben, während `d4` inkrementiert wird und umgekehrt.



C++ spezifiziert nicht die Reihenfolge, in der Operatoren ausgeführt werden. Somit wissen wir nicht, ob `d2 + d3` oder `++d4` zuerst ausgeführt wird. Sie müssen Ihre Funktionen so schreiben, dass es darauf nicht ankommt.

318 Sonntagnachmittag

Anders als `operator+( )` modifiziert `operator++( )` sein Argument. Es gibt daher keinen Grund, ein temporäres Objekt zu erzeugen und als Wert zurückzugeben. Das übergebene Argument wird als Referenz an den Aufrufenden zurückgegeben. Die folgende Funktion, die als Wert zurückgibt, enthält einen subtilen Bug:

```
// das ist nicht zu 100% verlässlich
USDollar operator++(USDollar& s)
{
    s.cents++;
    s.rationalize();
    return s;
}
```

Indem `s` als Wert zurückgegeben wird, zwingt die Funktion den Compiler, eine Kopie des Objekts zu machen. In den meisten Fällen ist das in Ordnung. Aber was passiert in einem zugegebenermaßen ungewöhnlichen aber zulässigen Ausdruck wie `++(++a)`? Wir würden erwarten, dass `a` um 2 erhöht wird. Mit der vorangegangenen Definition wird `a` jedoch um 1 erhöht und dann wird eine Kopie von `a` – und nicht `a` selber – um 1 erhöht.

Die allgemeine Regel sieht so aus: Wenn der Operator den Wert des Arguments ändert, übergeben Sie das Argument als Referenz, so dass das Original modifiziert werden und das Argument als Referenz zurückgegeben werden kann, für den Fall, dass das gleiche Objekt in nachfolgenden Operationen verwendet wird. Wenn der Operator nicht den Wert seiner Argumentes verändert, erzeugen Sie ein neues Objekt zur Speicherung des Ergebnisses und geben Sie dieses Objekt als Wert zurück. Die Eingabeargumente können bei Operatoren mit zwei Argumenten immer als Referenzen übergeben werden, um Zeit zu sparen, aber keines der Argumente sollte dann verändert werden.



Es gibt binäre Operatoren, die den Wert ihrer Argumente verändern, wie die speziellen Operatoren `+=`, `*=`, usw.

27.5 Operatoren als Elementfunktionen

Ein Operator kann zusätzlich zu seiner Implementierung als Nichtelement eine Elementfunktion sein. Auf diese Art implementiert sieht unser Beispiel `USDollar` wie in Listing 27-2 aus. (Nur die betreffenden Stellen werden gezeigt.)



Die vollständige Version des Programms in Listing 27-2 finden Sie in der Datei `USDollarMemberAdd` auf der beiliegenden CD-ROM.

Lektion 27 – Überladen von Operatoren 319**Listing 27-2: Implementierung eines Operators als Elementfunktion**

```

// USDollar - repräsentiert den US Dollar
class USDollar
{
public:
    // konstruiere ein Dollarobjekt mit initialen
    // Werten für Dollar und Cents
    USDollar(int d = 0, int c = 0) {
        dollars = d;
        cents = c;

        rationalize();
    }

    // rationalize - normalisiere den Centbetrag
    //                durch Addition eines Dollars pro
    //                100 Cents
    void rationalize()
    {
        dollars += (cents / 100);
        cents  %= 100;
    }

    // output - schreibe den Wert des Objektes
    //                in die Standardausgabe
    void output()
    {
        cout << >>$<<
            << dollars
            << >>.<<
            << cents;
    }

    //operator+ - addiere das aktuelle Objekt
    //                zu s2 und gib das Ergebnis
    //                in einem neuen Objekt zurück
    USDollar operator+(USDollar& s2)
    {
        int cents  = this->cents  + s2.cents;
        int dollars = this->dollars + s2.dollars;
        return USDollar(dollars, cents);
    }

    //operator++ - inkrementiere das aktuelle
    //                Objekt
    USDollar& operator++()
    {
        cents++;
        rationalize();
        return *this;
    }

protected:
    int dollars;
    int cents;
};

```

320 Sonntagnachmittag

Das Nichtelement `operator+(USDollar, USDollar)` wurde zur Elementfunktion `USDollar::operator+(USDollar)` umgeschrieben. Auf den ersten Blick scheint diese Elementversion ein Argument weniger zu haben als die Nichtelementversion. Wenn Sie zurückdenken, werden Sie sich jedoch daran erinnern, dass `this` das erste, versteckte Argument aller Elementfunktionen ist.

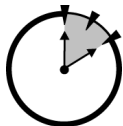
Dieser Unterschied wird am klarsten bei `USDollar::operator+( )` selber. Hier sehen Sie die Nichtelementversion und die Elementversion hintereinander.

```
// operator+ - Nichtelementversion
USDollar operator+(USDollar& s1, USDollar& s2)
{
    int cents    = s1.cents    + s2.cents;
    int dollars  = s1.dollars  + s2.dollars;
    USDollar t(dollars, cents);
    return t;
}

//operator+ - Elementversion
USDollar USDollar::operator+(USDollar& s2)
{
    int cents    = this->cents    + s2.cents;
    int dollars  = this->dollars  + s2.dollars;
    USDollar t(dollars, cents);
    return t;
}
```

Wir können sehen, dass die Funktionen fast identisch sind. Jedoch dort, wo die Nichtelementversion `s1` und `s2` addiert, addiert die Elementversion das aktuelle Objekt – auf das `this` zeigt – und `s2`.

Die Elementversion eines Operators hat immer ein Argument weniger als die Nichtelementversion – das Argument auf der linken Seite ist implizit.



10 Min.

27.6 Eine weitere Irritation durch Überladen

Nur weil Sie `operator*(double, USDollar&)` überladen haben, heißt das nicht, dass Sie `operator*(USDollar&, double)` überladen haben. Weil diese Operatoren verschiedene Argumente haben, müssen sie separat überschrieben werden. Das muss nicht so aufwendig sein, wie es vielleicht zuerst aussieht.

Erstens, kann ein Operator sich auf einen anderen Operator beziehen. Im Falle von `operator+( )`, würden wir wahrscheinlich etwas in der folgenden Art tun:

```
USDollar operator*(USDollar& s, double f)
{
    // ... Implementierung der Funktion ...
}

inline USDollar operator*(double f, USDollar& s)
{
    // verwende die obige Definition
    return s * f;
}
```

Die zweite Version ruft einfach die erste Version auf mit der entsprechenden Reihenfolge der Operanden. Die Deklaration als `inline` spart jeden Zusatzaufwand.

Lektion 27 – Überladen von Operatoren 321**27.7 Wann sollte ein Operator ein Element sein?**

Es gibt keinen großen Unterschied zwischen den Implementierungen eines Operators als Element und als Nichtelement mit diesen Ausnahmen:

1. Die folgenden Operatoren müssen als Elementfunktionen implementiert werden:
 - = Zuweisung
 - () Funktionsaufruf
 - [] Indizierung
 - > Klassenzugehörigkeit
2. Ein Operator wie der Folgende kann nicht als Elementfunktion implementiert werden:

```
// operator*(double, USDollar&) - definiert auf
//      Basis von operator*(USDollar&, double)
USDollar operator*(double factor, USDollar& s)
{
    return s * factor;
}
```

Um eine Elementfunktion sein zu können, muss `operator(float, USDollar&)` ein Element der Klasse `double` sein. Wie bereits früher erwähnt, können wir keine Operatoren zu elementaren Klassen hinzufügen. Somit muss ein Operator, für den nur das zweite Argument aus der Klasse ist, als Nichtelement implementiert werden.

Operatoren, die das Objekt, auf dem sie arbeiten, verändern, wie z.B. `operator++()`, sollten Element der Klasse sein.

27.8 Cast-Operator

Auch der Cast-Operator kann überschrieben werden. Das Programm `USDollarCast` in Listing 27-3 zeigt die Definition und den Gebrauch eines Cast-Operators, der ein `USDollar`-Objekt in ein `double` konvertiert, und wieder zurück.

Listing 27-3: Überladen des Cast-Operators

```
// USDollarCast - demonstriert das Schreiben eines
//      Cast-Operators; dieser konvertiert
//      USDollar nach double, der
//      Konstruktor konvertiert zurück
#include <stdio.h>
#include <iostream.h>

class USDollar
{
public:
    // Konstruktor, der USDollar aus double erzeugt
    USDollar(double value = 0.0);

    // Cast-Operator
    operator double()
    {
        return dollars + cents / 100.0;
    }
}
```

322 **Sonntagnachmittag**

```

// display - einfache Debug-Elementfunktion
void display(char* pszExp, double dV)
{
    cout << pszExp
        << » = $« << dollars << ».« << cents
        << » (» << dV << »)\n«;
}

protected:
    int dollars;
    int cents;
};

// Konstruktor - splitte den double-Wert in
// ganzzahligen und gebrochenen Teil
USDollar::USDollar(double value)
{
    dollars = (int)value;
    cents = (int)((value - dollars) * 100 + 0.5);
}

int main()
{
    USDollar d1(2.0), d2(1.5), d3, d4;

    // rufe Cast-Operator explizit auf ...
    double dVal1 = (double)d1;
    d1.display(>d1«, dVal1);

    double dVal2 = (double)d2;
    d2.display(>d2«, dVal2);

    d3 = USDollar((double)d1 + (double)d2);
    double dVal3 = (double)d3;
    d3.display(>d3 (Summe d1+d2 mit Casts)«, dVal3);

    //... oder implizit
    d4 = d1 + d2;
    double dVal4 = (double)d3;
    d4.display(>d4 (Summe d1+d2 ohne Casts)«, dVal4);

    return 0;
}

```

Ein Cast-Operator ist das Schlüsselwort `operator`, gefolgt von dem entsprechenden Typ. Die Elementfunktion `USDollar::operator double( )` stellt einen Mechanismus bereit, um ein Objekt der Klasse `USDollar` in ein `double` zu konvertieren. (Aus einem mir unbekannten Grund, haben Cast-Operatoren keinen Rückgabetyt.) Der Konstruktor `USDollar(double)` stellt den Konvertierungspfad von `double` zu `USDollar` her.

Wie das vorangegangene Beispiel zeigt, kann die Konvertierung mittels Cast-Operators entweder explizit oder implizit aufgerufen werden. Lassen Sie uns den impliziten Fall genauer betrachten.

Um den Ausdruck `d4 = d1 + d2` im Programm `USDollarCast` mit Sinn zu versehen, durchläuft C++ die folgenden Schritte:

Lektion 27 – Überladen von Operatoren 323

1. Zuerst sucht C++ nach einer Elementfunktion `USDollar::operator+(USDollar)`.
2. Wenn diese nicht gefunden werden konnte, sucht C++ nach der Nichtelementversion der gleichen Sache, d.h. `operator+(USDollar, USDollar)`.
3. Weil auch diese Version fehlt, sucht C++ nach einem `operator+( )`, den es unter Konvertierung des einen oder anderen Arguments in einen anderen Typ verwenden könnte. Schließlich findet es etwas Passendes: Wenn beide, `d1` und `d2`, nach `double` konvertiert werden, kann der eingebaute `operator+(double, double)` verwendet werden. Selbstverständlich muss das Ergebnis mittels des Konstruktors von `double` nach `USDollar` konvertiert werden.

Die Ausgabe von `USDollarCast` finden Sie unten. Die Funktion `USDollar::cast( )` erlaubt es dem Programmierer, frei zwischen `USDollar`-Objekten und `double`-Werten hin und her zu konvertieren.

```
d1 = $2.0 (2)
d2 = $1.50 (1.5)
d3 (Summe d1+d2 mit Casts) = $3.50 (3.5)
d4 (Summe d1+d2 ohne Casts) = $3.50 (3.5)
```

Das zeigt sowohl den Vorteil als auch den Nachteil davon, Cast-Operatoren bereitzustellen. Die Bereitstellung eines Konvertierungspfades von `USDollar` nach `double` befreit den Programmierer davon, einen vollständigen Satz von Operatoren bereitstellen zu müssen. `USDollar` kann einfach auf die für `double` definierten Operatoren zurückgreifen.

Auf der anderen Seite nimmt das dem Programmierer die Kontrolle darüber, welche Operatoren definiert werden. Durch den Konvertierungspfad nach `double`, bekommt `USDollar` alle Operatoren von `double`, ob sie nun Sinn machen oder nicht. Ich hätte genauso gut `d4 = d1 * d2` schreiben können. Außerdem kann es sein, dass diese zusätzliche Konvertierung nicht besonders schnell ist. Diese einfache Addition z.B. enthält drei Typkonvertierungen mit all den verbundenen Funktionsaufrufen, Multiplikationen, Divisionen usw.

Passen Sie auf, dass Sie nicht zwei Konvertierungspfade zum gleichen Typ bereitstellen. Das Folgende muss Probleme erzeugen:

```
class A
{
public:
    A(B& b);
};
class B
{
public:
    operator A();
};
```



0 Min.

Wenn ein Objekt der Klasse `B` in ein Objekt der Klasse `A` konvertiert werden soll, weiß der Compiler nicht, ob er den Cast-Operator `B::operator A( )` von `B` oder den Konstruktor `a::A(B&)` von `A` verwenden soll, die beide von einem `B` ausgehen und bei einem `A` ankommen.

Vielleicht ist das Ergebnis der beiden Pfade das Gleiche, aber der Compiler weiß das nicht. C++ muss wissen, welchen Konvertierungspfad Sie meinen. Der Compiler gibt eine Meldung aus, wenn er den Pfad nicht unzweideutig bestimmen kann.

324 **Sonntagnachmittag****Zusammenfassung**

Eine neue Klasse mit den entsprechenden Operatoren zu überladen, kann zu einfachem und elegantem Anwendungscode führen. In den meisten Fällen ist das Überladen von Operatoren jedoch nicht nötig. Die folgenden Sitzungen untersuchen zwei Fälle, in denen das Überladen von Operatoren kritisch ist.

- Das Überladen von Operatoren ermöglicht es dem Programmierer, existierende Operatoren für seine eigenen Klassen neu zu definieren. Der Programmierer kann jedoch keine neuen Operatoren hinzufügen, noch die Syntax bestehender Operatoren ändern.
- Es ist ein entscheidender Unterschied zwischen Übergabe und Rückgabe eines Objekts als Wert oder als Referenz. Abhängig vom Operator kann dieser Unterschied kritisch sein.
- Operatoren, die das Objekt verändern, sollten als Element implementiert werden. Einige Operatoren müssen als Element implementiert werden. Operatoren, bei denen auf der linken Seite ein elementarer Datentyp und keine benutzerdefinierte Klasse steht, können nicht als Elementfunktionen implementiert werden. Ansonsten macht es keinen großen Unterschied.
- Der Cast-Operator erlaubt es dem Programmierer, C++ mitzuteilen, wie ein benutzerdefiniertes Klassenobjekt in einen elementaren Typ konvertiert werden kann. Z.B. könnte die Konvertierung von `Student` nach `int` die ID des Studenten zurückgeben (ich habe nicht gesagt, dass diese Konvertierung eine gute Idee ist, sondern nur, dass sie möglich ist.) Dann können eigene Klassen mit den elementaren Datentypen in Ausdrücken gemischt werden.
- Benutzerdefinierte Operatoren erlauben es dem Programmierer, Programme zu schreiben, die leichter zu lesen und zu pflegen sind. Eigene Operatoren können jedoch trickreich sein und sollten mit Vorsicht verwendet werden.

Selbsttest

1. Es ist wichtig, dass Sie drei Operatoren für jede Klasse überschreiben können. Welche Operatoren sind das? (Siehe Einleitung)
2. Wie könnte der folgende Code Sinn machen? (Siehe »Warum soll ich Operatoren überladen?«)

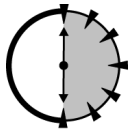
```
USDollar dollar(100, 0);  
DM& mark = !dollar;
```
3. Gibt es einen anderen Weg, das Obige nur durch »normale« Funktionsaufrufe zu schreiben, ohne Operatoren zu verwenden, die vom Programmierer geschrieben wurden? (Siehe »Warum soll ich Operatoren überladen?«)

Der Zuweisungsoperator



Checkliste

- ☒ Einführung in den Zuweisungsoperator
- ☒ Warum und wann der Zuweisungsoperator nötig ist
- ☒ Ähnlichkeiten von Zuweisungsoperator und Kopierkonstruktor



30 Min.

Ob Sie nun anfangen, Operatoren zu überladen oder nicht, Sie müssen schon früh lernen, den Zuweisungsoperator zu überladen. Der Zuweisungsoperator kann für jede benutzerdefinierte Klasse überladen werden. Wenn Sie sich an das hier vorgestellte Muster halten, werden Sie sehr bald Ihre eigene Version von `operator=( )` schreiben.

28.1 Warum ist das Überladen des Zuweisungsoperators kritisch?

C++ stellt eine Defaultdefinition von `operator=( )` für alle benutzerdefinierten Klassen bereit. Diese Defaultdefinition erstellt eine Element-zu-Element-Kopie, so ähnlich wie der Kopierkonstruktor. In folgendem Beispiel werden alle Elemente von `source` über die entsprechenden Elemente von `destination` kopiert.

```
void fn()
{
    MyStruct source, destination;
    destination = source;
}
```

Diese Defaultimplementierung ist jedoch nicht korrekt für Klassen, die Ressourcen allozieren wie z.B. Speicher vom Heap. Der Programmierer muss `operator=( )` überladen, um den Transfer von Ressourcen zu realisieren.

326 **Sonntagnachmittag****28.1.1 Vergleich mit Kopierkonstruktor**

Der Zuweisungsoperator ist dem Kopierkonstruktor sehr ähnlich. Eingesetzt sehen die beiden fast identisch aus.

```
void fn(MyClass &mc)
{
    MyClass newMC(mc);    // klar, das verwendet den
                        // Kopierkonstruktor
    MyClass newerMC = mc; // weniger klar, das ruft
                        // auch Kopierkonstruktor
    MyClass newestMC;      // das erzeugt
                        // Default-Objekt
    newestMC = mc;         // und überschreibt es mit
                        // dem Argument
}
```

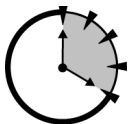
Die Erzeugung von `newMC` folgt dem Standardmuster, ein neues Objekt unter Verwendung des Kopierkonstruktors `MyClass(MyClass&)` als ein Spiegelbild des Originals zu erzeugen. Nicht so offensichtlich ist, dass C++ das zweite Format erlaubt, bei dem `newerMC` mittels Kopierkonstruktor erzeugt wird.

`newestMC` wird mittels des Default-Konstruktors erzeugt und dann durch den Zuweisungsoperator mit `mc` überschrieben. Der Unterschied ist, dass bei Aufruf des Kopierkonstruktors für `newerMC` dieses Objekt noch nicht existierte. Bei Aufruf des Zuweisungsoperators für `newestMC` war es bereits ein `MyClass`-Objekt im besten Sinne.

**Tipp**

Die Regel sieht so aus: Der Kopierkonstruktor wird benutzt, wenn ein neues Objekt erzeugt wird. Der Zuweisungsoperator wird verwendet, wenn das Objekt auf der linken Seite bereits existiert.

Wie der Kopierkonstruktor sollte ein Zuweisungsoperator immer dann bereitgestellt werden, wenn eine flache Kopie nicht angebracht ist. (Sitzung 20 enthält eine umfangreiche Diskussion von flachen und tiefen Konstruktoren.) Es reicht aus zu sagen, dass ein Kopierkonstruktor und ein Zuweisungsoperator dann definiert werden sollten, wenn die Klasse Ressourcen alloziert, damit es nicht dazu kommt, dass zwei Objekte auf die gleichen Ressourcen zeigen.

**20 Min.****28.2 Wie den Zuweisungsoperator überladen?**

Den Zuweisungsoperator zu überladen funktioniert so, wie bei den anderen Operatoren. Das Beispielprogramm `DemoAssign`, das Sie in Listing 28-1 finden, enthält sowohl einen Kopierkonstruktor als auch einen Zuweisungsoperator.

**Hinweis**

Denken Sie daran, dass der Zuweisungsoperator ein Element der Klasse sein muss.

Lektion 28 – Der Zuweisungsoperator**327****Listing 28-1: Überladen des Zuweisungsoperators**

```
// DemoAssign - demonstrate the assignment operator
#include <stdio.h>
#include <string.h>
#include <iostream.h>

// Name - eine generische Klasse, die den
//       Zuweisungsoperator und den
//       Kopierkonstruktor demonstriert
class Name
{
public:
    Name(char *pszN = 0)
    {
        copyName(pszN);
    }
    Name(Name& s)
    {
        copyName(s.pszName);
    }
    ~Name()
    {
        deleteName();
    }
    // Zuweisungsoperator
    Name& operator=(Name& s)
    {
        // gib Altes frei ...
        deleteName();
        //... bevor es durch Neues ersetzt wird
        copyName(s.pszName);
        // gib Referenz auf Objekt zurück
        return *this;
    }

    // display - gibt das Objekt in die
    //           Standardausgabe aus
    void display()
    {
        cout << pszName;
    }

protected:
    void copyName(char *pszN);
    void deleteName();
    char *pszName;
};

// copyName() - alloziere Heapspeicher zum Speichern
void Name::copyName(char *pszName)
{
    this->pszName = 0;
    if (pszName)
    {
```

328 **Sonntagnachmittag**

```
        this->pszName = new char[strlen(pszName) + 1];
        strcpy(this->pszName, pszName);
    }
}

// deleteName() - gib Heapspeicher zurück
void Name::deleteName()
{
    if (pszName)
    {
        delete pszName;
        pszName = 0;
    }
}

// displayNames - Ausgabefunktion, um die Zeilen in
//                  main() zu reduzieren
void displayNames(Name& pszN1, char* pszMiddle,
                  Name& pszN2, char* pszEnd)
{
    pszN1.display();
    cout << pszMiddle;
    pszN2.display();
    cout << pszEnd;
}

int main(int nArg, char* pszArgs[])
{
    // erzeuge zwei Objekte
    Name n1(>>Claudette<<);
    Name n2(>>Greg<<);
    displayNames(n1, >> und >>,
                 n2, >> sind neu erzeugte Objekte\n<<);

    // mache eine Kopie eines Objektes
    Name n3(n1);
    displayNames(n3, >> ist eine Kopie von >>,
                 n1, >>\n<<);

    // mache eine Kopie des Objektes von der
    // Adresse aus
    Name* pN = &n2;
    Name n4(*pN);
    displayNames(n4, >> ist eine Kopie (Adresse) von<<,
                 n2, >>\n<<);

    // überschreibe n2 mit n1
    n2 = n1;
    displayNames(n1, >> wurde zugewiesen an >>,
                 n2, >>\n<<);
    return 0;
}
```

Lektion 28 – Der Zuweisungsoperator 329**Ausgabe:**

```
Claudette und Greg sind neu erzeugte Objekte
Claudette ist eine Kopie von Claudette
Greg ist eine Kopie (Adresse) von Greg
Claudette wurde zugewiesen an Claudette
```

Die Klasse `Name` hält den Namen einer Person im Speicher, der vom Heap alloziert wurde. Die Konstruktoren und der Destruktor der Klasse `Name` sind denen sehr ähnlich, die in den Sitzungen 19 und 20 vorgestellt wurden. Der Konstruktor `Name(char*)` kopiert den gegebenen Namen in das Datenelement `pszName`. Dieser Konstruktor ist auch der Defaultkonstruktor. Der Kopierkonstruktor `Name(&Name)` kopiert den Namen des übergebenen Objektes in den Namen des aktuellen Objektes durch einen Aufruf der Funktion `copyName()`. Der Destruktor gibt die `pszName`-Zeichenkette durch einen Aufruf von `deleteName()` an den Heap zurück.

Die Funktion `main()` demonstriert jede dieser Elementfunktionen. Die Ausgabe von `DemoAssign` finden Sie oben am Ende von Listing 28-1.

Schauen Sie sich den Zuweisungsoperator genau an. Die Funktion `operator=( )` sieht doch wirklich aus wie ein Destruktor, unmittelbar gefolgt von einem Kopierkonstruktor. Das ist typisch. Betrachten Sie die Zuweisung im Beispiel `n2 = n1`. Das Objekt `n2` hat bereits einen Namen (»Greg«). In der Zuweisung muss der Speicher, den der ursprüngliche Name belegt, an den Heap zurückgegeben werden durch einen Aufruf von `deleteName()`, bevor neuer Speicher mittels `copyName()` alloziert und zugewiesen wird, in dem der neue Name (»Claudette«) gespeichert wird.

Der Kopierkonstruktor musste `deleteName()` nicht aufrufen, weil das Objekt noch nicht existierte. Es war daher noch kein Speicher belegt, als der Konstruktor aufgerufen wurde.

Im Allgemeinen hat ein Zuweisungsoperator zwei Teile. Der erste Teil baut den Destruktor in dem Sinne nach, dass er die belegten Ressourcen des Objektes freigibt. Der zweite Teil baut den Kopierkonstruktor nach in dem Sinne, dass er neue Ressourcen alloziert.

28.2.1 Zwei weitere Details des Zuweisungsoperators

Es gibt zwei weitere Details des Zuweisungsoperators, die Sie kennen sollten. Erstens ist der Rückgabetypp von `operator=( )` gleich `Name&`. Ich bin darauf nicht im Detail eingegangen, aber der Zuweisungsoperator ist ein Operator wie jeder andere. Ausdrücke, die einen Zuweisungsoperator enthalten, haben einen Wert und einen Typ, wobei diese beiden vom endgültigen Wert auf der linken Seite stammen. Im folgenden Beispiel ist der Wert von `operator=( )` gleich 2.0 und der Typ ist `double`:

```
double d1, d2;
void fn(double );
d1 = 2.0;
```

Dadurch wird es möglich, dass der Programmierer schreiben kann:

```
d2 = d1 = 2.0
fn(d2 = 3.0); // führt die Zuweisung aus, und
              // übergibt den Ergebniswert an fn()
```

Der Wert 2.0 der Zuweisung `d1 = 2.0` und ihr Typ `double` werden an den nächsten Zuweisungsoperator übergeben. Im zweiten Beispiel wird der Wert der Zuweisung `d2 = 3.0` an die Funktion `fn()` übergeben.

330 *Sonntagnachmittag*

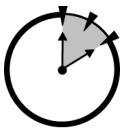
Ich hätte auch `void` zum Rückgabotyp von `Name::operator=( )` machen können. Wenn ich das jedoch tue, funktioniert das obige Beispiel nicht mehr:

```
void otherFn(Name&);
void fn()
{
    Name n1, n2, n3;

    // das Folgende ist nur möglich, wenn der
    // Zuweisungsoperator eine Referenz auf
    // das aktuelle Objekt zurückgibt
    n1 = n2 = n3;
    otherFn(n1 = n2);
}
```

Das Ergebnis der Zuweisung `n1 = n2` ist `void` – der Rückgabotyp von `operator=( )` – was nicht mit dem Prototyp von `otherFn( )` übereinstimmt. Die Deklaration von `operator=( )` mit einer Referenz auf das aktuelle Objekt und die Rückgabe von `*this` bleiben die Semantik für den Zuweisungsoperator bei elementaren Typen.

Das zweite Detail ist, dass `operator=( )` als Elementfunktion geschrieben wurde. Anders als andere Operatoren kann der Zuweisungsoperator nicht mit einer Nichtelementfunktion überladen werden. Die speziellen Zuweisungsoperatoren, wie `+=` und `*=`, haben keine besonderen Einschränkungen und können Nichtelemente sein.



10 Min.

28.3 Ein Schlupfloch

Ihre Klasse mit einem Zuweisungsoperator auszustatten, kann ihren Anwendungscode sehr flexibel machen. Wenn Ihnen das jedoch zu viel ist, oder wenn Sie keine Kopien Ihrer Objekte machen können, verhindert das Überladen des Zuweisungsoperators mit einer `protected`-Funktion, dass jemand versehentlich eine flache Kopie eines Objektes erstellt. Z.B.:

```
class Name
{
    //... wie zuvor ...
protected:
    // Zuweisungsoperator
    Name& operator=(Name& s)
    {
        return *this;
    }
};
```

Mit dieser Definition, werden Zuweisungen wie die folgende verhindert:

```
void fn(Name &n)
{
    Name newN;
    newN = n; // erzeugt einen Compilerfehler -
              // Funktion hat keinen Zugriff
              // auf operator=( )
}
```

Lektion 28 – Der Zuweisungsoperator**331****0 Min.**

Dieser Kopierschutz für Klassen erspart Ihnen den Ärger mit dem Überladen des Zuweisungsoperators, aber reduziert die Flexibilität Ihrer Klasse.

**Hinweis**

*Wenn Ihre Klasse Ressourcen alloziert, wie z.B. Speicher vom Heap, müssen Sie entweder einen entsprechenden Zuweisungsoperator und Kopierkonstruktor schreiben oder beide **protected** machen und verhindern, dass die von C++ bereitgestellte Defaultmethode verwendet wird.*

Zusammenfassung

Der Zuweisungsoperator ist der einzige Operator, den Sie überschreiben müssen, aber nur unter bestimmten Bedingungen. Glücklicherweise ist es nicht schwer, einen Zuweisungsoperator für Ihre Klasse zu definieren, wenn Sie dem Muster folgen, das in dieser Sitzung beschrieben wurde.

- C++ stellt einen Default-Zuweisungsoperator bereit, der eine Element-zu-Element-Kopie durchführt. Diese Version der Zuweisung ist für viele Klassentypen in Ordnung; Klassen jedoch, die Ressourcen allozieren, müssen einen Kopierkonstruktor und einen überladenen Zuweisungsoperator enthalten.
- Die Semantik des Zuweisungsoperators entspricht im Wesentlichen einem Destruktor, gefolgt von einem Kopierkonstruktor. Der Destruktor entfernt alle Ressourcen, die möglicherweise bereits existieren, während der Kopierkonstruktor eine tiefe Kopie der zugewiesenen Ressourcen erstellt.
- Den Zuweisungsoperator **protected** zu deklarieren, reduziert die Gefahr, aber beschränkt Ihre Klasse, indem mit Ihrer Klasse keine Zuweisungen ausgeführt werden können.

Selbsttest

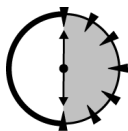
1. Wann muss Ihre Klasse einen Zuweisungsoperator enthalten? (Siehe »Warum ist das Überladen des Zuweisungsoperators kritisch?«)
2. Der Rückgabotyp des Zuweisungsoperators sollte immer mit dem Klassentyp übereinstimmen. Warum? (Siehe »Zwei weitere Details des Zuweisungsoperators«)
3. Wie können Sie verhindern, einen Zuweisungsoperator schreiben zu müssen? (Siehe »Ein Schlupfloch«)



Stream-I/O

Checkliste

- ☒ Stream-I/O als überladenen Operator wiederentdecken
- ☒ Streamdatei-I/O verwenden
- ☒ Streampuffer-I/O verwenden
- ☒ Eigene Insertter und Extraktor schreiben
- ☒ Hinter den Kulissen von Manipulatoren



30 Min.

Bis jetzt haben alle Programme ihre Eingaben über das `cin`-Eingabeobjekt und ihre Ausgaben über das `cout`-Ausgabeobjekt erledigt. Vielleicht haben Sie nicht viel darüber nachgedacht, aber diese Technik der Eingabe/Ausgabe ist eine Teilmenge dessen, was als Stream-I/O bezeichnet wird.

Diese Sitzung erklärt Stream-I/O im Detail. Ich muss Sie warnen: Stream-I/O ist ein zu großes Thema, um in einer einzigen Sitzung behandelt werden zu können – ganze Bücher sind diesem Thema gewidmet. Ich kann Ihnen jedoch zu einem Anfang verhelfen, so dass Sie die Hauptoperationen durchführen können.

29.1 Wie funktioniert Stream-I/O?

Stream-I/O basiert auf überladenen Versionen von `operator>>( )` und `operator<<( )`. Die Deklaration dieser überladenen Operatoren finden sich in der Include-Datei `iostream.h`, die wir in unsere Programme seit Sitzung 2 eingebunden haben. Der Code für diese Funktionen ist in der Standardbibliothek von C++ enthalten, mit der Ihr C++-Programm gelinkt wird. Das Folgende zeigt ein paar Prototypen aus `iostream.h`:

```
// für die Eingabe haben wir:
istream& operator>>(istream& source, char *pDest);
istream& operator>>(istream& source, int &dest);
istream& operator>>(istream& source, char &dest);
//... usw. ...

// für die Ausgabe haben wir:
```



```
ostream& operator<<(ostream& dest, char *pSource);
ostream& operator<<(ostream& dest, int source);
ostream& operator<<(ostream& dest, char source);
//... usw. ...
```

Wenn `operator>>( )` für I/O überladen wird, wird er *Extraktor* genannt; `operator<<( )` wird *Insertor* genannt.

Lassen Sie uns im Detail ansehen, was passiert, wenn ich Folgendes schreibe:

```
#include <iostream.h>
void fn()
{
    cout << »Ich heiÙe Randy\n«;
}
```

Das Objekt `cout` ist ein Objekt der Klasse `ostream` (mehr dazu später). Somit bestimmt C++, dass die Funktion `operator<<(ostream&, char*)` am besten übereinstimmt. C++ erzeugt einen Aufruf dieser Funktion, dem so genannten `char*`-Insertor und übergibt der Funktion das `ostream`-Objekt `cout` und die Zeichenkette `»Ich heiÙe Randy\n«` als Argument. D.h. es wird aufgerufen `operator<<(cout, »Ich heiÙe Randy\n«)`. Der `char*`-Insertor, der Teil der C++-Standardbibliothek ist, führt die angeforderte Ausgabe durch.

Die Klassen `ostream` und `istream` sind die Basis für eine Menge von Klassen, die den Anwendungscode mit der Außenwelt verbinden, Eingabe vom und Ausgabe ins Dateisystem eingeschlossen. Woher wusste der Compiler, dass `cout` aus der Klasse `ostream` ist? Diese und einige andere globale Objekte sind in `iostream.h` deklariert. Eine Liste dieser Objekte finden Sie in Tabelle 29-1. Diese Objekte werden bei Programmstart automatisch erzeugt, bevor `main( )` die Kontrolle erhält.

Tabelle 29-1: Objekte der Standard-Stream-I/O

Objekt	Klasse	Aufgabe
<code>cin</code>	<code>istream</code>	Standardeingabe
<code>cout</code>	<code>ostream</code>	Standardausgabe
<code>cerr</code>	<code>ostream</code>	Standardfehlerausgabe
<code>clog</code>	<code>ostream</code>	Standarddruckerausgabe

Unterklassen von `ostream` und `istream` werden für die Eingabe von und die Ausgabe in Dateien und interne Puffer verwendet.

29.2 Die Unterklassen *fstream*

Die Unterklassen `ofstream`, `ifstream` und `fstream` sind in der Include-Datei `fstream.h` definiert, um Streameingabe und Streamausgabe für Dateien zu realisieren. Diese drei Klassen bieten eine Vielzahl von Elementfunktionen. Eine vollständige Liste finden Sie in der Dokumentation Ihres Compilers, aber lassen Sie mich eine kurze Einführung geben.

334 Sonntagnachmittag

Die Klasse `ofstream`, die für die Dateiausgabe verwendet wird, hat mehrere Konstruktoren, von denen der folgende der nützlichste ist:

```
ofstream::ofstream(char *pszFileName,
                  int mode = ios::out,
                  int prot = filebuf::openprot);
```

Das erste Argument ist ein Zeiger auf den Namen der Datei, die geöffnet werden soll. Das zweite und dritte Argument geben an, wie die Datei geöffnet werden soll. Die gültigen Werte für `mode` finden Sie in Tabelle 29-2 und die für `prot` in Tabelle 29-3. Diese Werte sind Bitfelder, die durch OR verbunden sind (die Klassen `ios` und `filebuf` sind beide Elternklasse von `ostream`).



Der Ausdruck `ios::out` bezieht sich auf ein statisches Element der Klasse `ios`.

Tabelle 29-2: Konstanten zur Kontrolle, wie Dateien geöffnet werden

Flag	Bedeutung
<code>ios::ate</code>	Anhängen ans Ende der Datei, falls sie existiert
<code>ios::in</code>	Datei zur Eingabe öffnen (implizit für <code>istream</code>)
<code>ios::out</code>	Datei zur Ausgabe öffnen (implizit für <code>ostream</code>)
<code>ios::trunc</code>	Schneide Datei ab, falls sie existiert (default)
<code>ios::nocreate</code>	Wenn Datei nicht bereits existiert, gibt Fehler zurück
<code>ios::noreplace</code>	Wenn Datei bereits existiert, gibt Fehler zurück
<code>ios::binary</code>	Öffne Datei im Binärmodus (Alternative ist Textmodus)

Tabelle 29-3: Werte für `prot` im Konstruktor `ofstream`

Flag	Bedeutung
<code>filebuf::openprot</code>	Kompatibilitäts-Sharing-Modus
<code>filebuf::sh_none</code>	Exklusiv – kein Sharing
<code>filebuf::sh_read</code>	Lese-Sharing erlaubt
<code>filebuf::sh_write</code>	Schreib-Sharing erlaubt

Lektion 29 – Stream I/O 335

Das folgende Programm z.B. öffnet eine Datei MYNAME und schreibt ein paar wichtige und absolut der Wahrheit entsprechende Informationen hinein:

```
#include <fstream.h>
void fn()
{
    // öffne die Textdatei MYNAME zum Schreiben -
    // überschreibe, was in der Datei steht
    ofstream myn(>>MYNAME<<);
    myn << >>Randy Davis ist höflich und hübsch\n<<;
}
```

Der Konstruktor `ofstream::ofstream(char*)` erwartet nur einen Dateinamen und stellt Default-Werte für die anderen Dateimodi bereit. Wenn die Datei MYNAME bereits existiert, wird sie geleert; anderenfalls wird MYNAME erzeugt. Zusätzlich wird die Datei im Kompatibilitäts-Sharing-Modus geöffnet.

Wenn ich eine Datei im Binärmodus öffnen und an das Ende der Datei anfügen möchte, wenn die Datei bereits existiert, würde ich wie folgt ein `ostream`-Objekt erzeugen (siehe Tabelle 29-2). (Im Binärmodus werden Zeilenumbrüche bei der Ausgabe nicht in Carriage Return und Line Feed verwandelt, und die umgekehrte Konvertierung findet bei der Eingabe ebenfalls nicht statt.)

```
void fn()
{
    // öffne die Binärdatei BINFILE zum Schreiben;
    // wenn sie bereits existiert, füge ans Ende an
    ofstream bfile(>>BINFILE<<, ios::binary | ios::ate);
    //... Fortsetzung wie eben ...
}
```

Die Streamobjekte enthalten Zustandsinformationen über ihren I/O-Prozess. Die Elementfunktion `bad()` gibt ein Fehlerflag zurück, das innerhalb der Klassen geführt wird. Das Flag ist nicht null, wenn das Dateiojekt einen Fehler enthält.



Streamausgabe geht der Ausnahmen-basierten Technik der Fehlerbehandlung voraus, die in Sitzung 30 erklärt wird.

Um zu überprüfen, ob die Dateien MYNAME und BINFILE in dem früheren Beispiel korrekt geöffnet wurden, könnte ich schreiben:

```
#include <fstream.h>
void fn()
{
    ofstream myn(>>MYNAME<<);
    if (myn.bad()) // wenn das Öffnen fehlschlägt ...
    {
        cerr << >>Fehler beim Öffnen von MYNAME\n<<;
        return;    //... Fehler ausgeben und fertig
    }
    myn << >>Randy Davis ist höflich und hübsch\n<<;
}
```

336 Sonntagnachmittag

Alle Versuche, Ausgaben mit einem `ofstream`-Objekt durchzuführen, das einen Fehler enthält, haben keinen Effekt, bis der Fehler durch einen Aufruf der Elementfunktion `clear( )` gelöscht wird.



Dieser letzte Paragraph ist wörtlich gemeint – es ist keine Ausgabe möglich, so lange das Fehlerflag nicht null ist.

Der Destruktor der Klasse `ofstream` schließt die Datei automatisch. Im vorangegangenen Beispiel wurde die Datei bei Verlassen der Funktion geschlossen.

Die Klasse `ifstream` arbeitet auf die gleiche Weise bei der Eingabe, wie das folgende Beispiel zeigt:

```
#include <fstream.h>
void fn()
{
    // öffnet Datei zum Lesen; erzeuge die
    // Datei nicht, wenn sie nicht existiert
    ifstream bankStatement(»STATEMNT«, ios::nocreate);
    if (bankStatement.bad())
    {
        cerr << »Datei STATEMNT nicht gefunden\n«;
        return;
    }
    while (!bankStatement.eof())
    {
        bankStatement >> nAccountNumber >> amount;
        // ... verarbeite Abhebung
    }
}
```

Die Funktion öffnet die Datei STATEMNT durch die Erzeugung des Objektes `bankStatement`. Wenn die Datei nicht existiert, wird sie erzeugt. (Wir nehmen an, dass die Datei Informationen für uns hat, es würde daher keinen Sinn machen, eine neue, leere Datei zu erzeugen.) Wenn das Objekt fehlerhaft ist (z.B. wenn das Objekt nicht erzeugt wurde), gibt die Funktion eine Fehlermeldung aus und beendet die Ausführung. Andernfalls durchläuft die Funktion eine Schleife und liest dabei `nAccountNumber` und den Abhebungsbetrag `amount`, bis die Datei leer ist (end-of-file ist wahr).

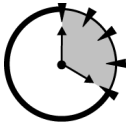
Der Versuch, aus einem `ifstream`-Objekt zu lesen, das einen Fehler enthält, kehrt sofort zurück, ohne etwas gelesen zu haben.



Lassen Sie mich erneut warnen. Es wird nicht nur nichts zurückgegeben, wenn aus einem Eingabestream gelesen wird, der einen Fehler enthält, sondern der Puffer kommt unverändert zurück. Das Programm kann leicht den falschen Schluss daraus ziehen, dass die gleiche Eingabe wie zuvor gelesen wurde. Schließlich wird `eof( )` nie true liefern auf einem Stream, der sich im Fehlerzustand befindet.

Die Klasse `fstream` ist wie eine Klasse, die `ifstream` und `ofstream` kombiniert (in der Tat erbt sie von beiden). Ein Objekt der Klasse `fstream` kann zur Eingabe oder zur Ausgabe geöffnet werden oder für beides.

29.3 Die Unterklassen `strstream`



20 Min.

Die Klassen `istrstream`, `ostrstream` und `strstream` sind in der Include-Datei `strstream.h` definiert. (Der Dateiname erscheint unter MS-DOS abgeschnitten, weil dort nicht mehr als 8 Zeichen pro Dateiname erlaubt sind; GNU C++ verwendet den vollständigen Namen `strstream.h`.)

Diese Klassen erlauben die Operationen, die in den `fstream`-Klassen für Dateien definiert sind, für Puffer, die sich im Speicher befinden.

Der Codeschnipsel parst die Daten einer Zeichenkette unter Verwendung von Streameingabe:

```
#include <strstream.h>
// <strstream.h> für GNU C++
char* parseString(char *pszString)
{
    // assoziiere ein istrstream-Objekt mit der
    // Eingabezeichenkette
    istrstream inp(pszString, 0);

    // Eingabe von diesem Objekt
    int nAccountNumber;
    float dBalance;
    inp >> nAccountNumber >> dBalance;

    // alloziere einen Puffer und verbinde ihn
    // mit einem ostrstream-Objekt
    char* pszBuffer = new char[128];
    ostrstream out(pszBuffer, 128);

    // Ausgabe an dieses Objekt
    out << »Kontonummer = » << nAccountNumber
        << », Kontostand = $» << dBalance
        << ends;

    return pszBuffer;
}
```

Die Funktion scheint komplizierter zu sein, als sie sein müsste, `parseString()` ist jedoch einfach zu schreiben aber sehr robust. Die Funktion `parseString()` kann jeden Typ Input behandeln, den der C++-Extraktor behandeln kann, und sie hat alle Formatierungsfähigkeiten des C++-Inserters. Außerdem ist die Funktion tatsächlich sehr einfach, wenn Sie verstanden haben, was sie tut.

Lassen Sie uns z.B. annehmen, dass `pszString` auf die folgende Zeichenkette zeigt:

```
»1234 100.0«
```

Die Funktion `parseString()` assoziiert das Objekt `inp` mit der Eingabezeichenkette, indem dieser Wert an den Konstruktor von `istrstream` übergeben wird. Das zweite Argument des Konstruktors ist die Länge der Zeichenkette. In diesem Beispiel ist das Argument gleich 0, was bedeutet »lies bis zum terminierenden Nullzeichen«.

338 *Sonntagnachmittag*

Die Extraktor-Anweisung `inp >>` liest erst die Kontonummer, 1234, in die `int`-Variable `nAccountNumber`, genauso, als wenn sie von der Tastatur oder aus einer Datei gelesen würde. Der zweite Teil liest den Wert 100.0 in die Variable `dBalance`.

Bei der Ausgabe wird das Objekt `out` assoziiert mit dem 128 Zeichen umfassenden Puffer, auf den `pszBuffer` zeigt. Auch hier gibt das zweite Argument die Länge des Puffers an – für diesen Wert kann es keinen Default-Wert geben, weil `ofstream` keine Möglichkeit hat, die Länge des Puffers selber festzustellen (es gibt hier kein abschließendes Nullzeichen). Ein drittes Argument, das dem Modus entspricht, hat `ios::out` als Default-Wert. Sie können dieses Argument jedoch auf `ios::ate` setzen, wenn Sie die Ausgabe an das hängen möchten, was sich bereits im Puffer befindet, anstatt den Puffer zu überschreiben.

Die Funktion gibt dann das `out`-Objekt aus – das erzeugt die formatierte Ausgabe in den Puffer der 128 Zeichen. Schließlich gibt die Funktion `parseString( )` den Puffer zurück. Die lokal definierten Objekte `inp` und `out` werden bei Rückkehr der Funktion vernichtet.



Die Konstante `ends`, die am Ende des Inserter-Kommandos steht, ist nötig, um den Null-Terminator an das Ende der Pufferzeichenkette anzufügen.

Der Puffer, der durch den vorangegangenen Codeschnipsel zurückgegeben wurde, enthält die folgende Zeichenkette:

```
»Kontonummer = 1234, Kontostand = $100.00«
```

29.3.1 Vergleich von Techniken der Zeichenkettenverarbeitung

Die Streamklassen für Zeichenketten stellen ein äußerst mächtiges Konzept dar. Das wird selbst in einem einfachen Beispiel klar. Nehmen Sie an, ich habe eine Funktion, die eine beschreibende Zeichenkette für ein `USDollar`-Objekt erstellen soll.

Meine Lösung ohne Verwendung von `ostrstream` sieht wie in Listing 29-1 aus.

Listing 29-1: Konvertierung von `USDollar` in eine Zeichenkette zur Ausgabe

```
// ToStringWOutputStream - konvertiert USDollar in eine
//                               Zeichenkette
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <string.h>

// USDollar - repräsentiert den US Dollar
class USDollar
{
public:
    // konstruiere ein USDollar-Objekt mit einem
    // initialen Wert
    USDollar(int d = 0, int c = 0);
```

```
// rationalize - normalisiere nCents durch
//          Addition eines Dollars pro
//          100 Cents
void rationalize()
{
    nDollars += (nCents / 100);
    nCents  %= 100;
}

// output - gib eine Beschreibung des aktuellen
//          Objektes zurück
char* output();

protected:
    int nDollars;
    int nCents;
};

USDollar::USDollar(int d, int c)
{
    // speichere die initialen Werte
    nDollars = d;
    nCents = c;

    rationalize();
}

// output - gib eine Beschreibung des aktuellen
//          Objektes zurück
char* USDollar::output()
{
    // alloziere einen Puffer
    char* pszBuffer = new char[128];

    // konvertiere den Wert von nDollar und nCents
    // in Zeichenketten
    char cDollarBuffer[128];
    char cCentsBuffer[128];
    ltoa((long)nDollars, cDollarBuffer, 10);
    ltoa((long)nCents, cCentsBuffer, 10);

    // Cents sollen 2 Ziffern benutzen
    if (strlen(cCentsBuffer) != 2)
    {
        char c = cCentsBuffer[0];
        cCentsBuffer[0] = '0';
        cCentsBuffer[1] = c;
        cCentsBuffer[2] = '\\0';
    }

    // füge die Zeichenketten zusammen

    strcpy(pszBuffer, »$«);
    strcat(pszBuffer, cDollarBuffer);
    strcat(pszBuffer, ».«);
}
```

340 Sonntagnachmittag

```

    strcat(pszBuffer, cCentsBuffer);
    return pszBuffer;
}

int main(int nArgc, char* pszArgs[])
{
    USDollar d1(1, 60);
    char* pszD1 = d1.output();
    cout << »Dollar d1 = » << pszD1 << »\n«;
    delete pszD1;

    USDollar d2(1, 5);
    char* pszD2 = d2.output();
    cout << »Dollar d2 = » << pszD2 << »\n«;
    delete pszD2;

    return 0;
}

```

Ausgabe

```

Dollar d1 = $1.60
Dollar d2 = $1.05

```

Das Programm ToStringWOStream stützt sich nicht auf die Streamroutinen, um den Text für das USDollar-Objekt zu erzeugen. Die Funktion USDollar::output() macht intensiven Gebrauch von der Funktion ltoa(), die long in eine Zeichenkette verwandelt, und von den Funktionen strcpy() und strcat(), die direkte Manipulationen auf Zeichenketten ausführen. Die Funktionen müssen selber mit dem Fall zurecht kommen, dass die Anzahl Cents kleiner als 10 ist und daher nur eine Stelle belegt. Die Ausgabe des Programms finden Sie am Ende des Listings.

Das Folgende zeigt eine Version von USDollar::output(), die die Klasse ostream verwendet.

```

char* USDollar::output()
{
    // alloziere einen Puffer
    char* pszBuffer = new char[128];

    // verbinde einen ostream mit dem Puffer
    ostream out(pszBuffer, 128);

    // konvertiere in Zeichenketten (Setzen von
    // width stellt sicher, dass die Breite der
    // Cents nicht kleiner als 2 ist)
    out << »$« << nDollars << ».«;
    out.fill('0');
    out.width(2);
    out << nCents << ends;

    return pszBuffer;
}

```




Diese Version ist im Programm ToStringWStreams auf der beiliegenden CD-ROM enthalten.

Diese Version assoziiert den Ausgabestream `out` mit einem lokal definierten Puffer. Sie schreibt dann die nötigen Werte unter Verwendung der üblichen Stream-Inserters und gibt den Puffer zurück. Das Setzen der Breite auf 2 stellt sicher, dass die Anzahl der verwendeten Stellen auch dann zwei ist, wenn der Wert kleiner als 10 ist. Die Ausgabe dieser Version ist identisch mit der Ausgabe von Listing 29-1. Das `out`-Objekt wird vernichtet, wenn die Kontrolle die Funktion `output()` verlässt.

Ich finde, dass die Stream-Version von `output()` viel besser verfolgt werden kann und weniger langweilig ist als die frühere Version, die keine Streams nutzte.

29.4 Manipulatoren

Bis jetzt haben wir gesehen, wie Stream-I/O verwendet werden kann, um Zahlen und Zeichenkette auszugeben unter Verwendung von Default-Formaten. Normalerweise sind die Defaults in Ordnung, aber manchmal treffen sie es einfach nicht. Weil dies so ist, stellt C++ zwei Wege bereit, um die Formatierung der Ausgabe zu kontrollieren.

Erstens kann der Aufruf einer Reihe von Elementfunktionen des Stream-Objektes das Format steuern. Sie haben das in einer früheren Elementfunktion `display()` gesehen, in der `fill('0')` und `width(2)` die minimale Breite und das Linksfüllzeichen eines `ostream`-Objektes gesetzt haben.



Das Argument `out` stellt ein `ostream`-Objekt dar. Weil `ostream` Basisklasse für `ofstream` und `ostream` ist, funktioniert die Funktion gleich gut für die Ausgabe in eine Datei und einen im Programm bereitgestellten Puffer.

Ein zweiter Zugang ist der über *Manipulatoren*. Manipulatoren sind Objekte, die in der Include-Datei `iomanip.h` definiert sind, die den gleichen Effekt haben wie Aufrufe von Elementfunktionen.

Der einzige Vorteil von Manipulatoren ist, dass das Programm sie direkt in den Stream einfügen kann und keinen separaten Funktionsaufruf ausführen muss.

Die Funktion `display()` kann mit Manipulatoren wie folgt umgeschrieben werden:

```
char* USDollar::output()
{
    // alloziere einen Puffer
    char* pszBuffer = new char[128];

    // verbinde einen ostream mit dem Puffer
    ostream out(pszBuffer, 128);

    // konvertiere in Zeichenketten; diese Version
    // verwendet Manipulatoren zum Setzen des
```

342 **Sonntagnachmittag**

```
// Füllzeichens und der Breite
out << »$« << nDollars << ».«
    << setfill('0') << setw(2)
    << nCents << ends;

return pszBuffer;
}
```

Die geläufigsten Manipulatoren und ihre Bedeutung finden Sie in Tabelle 29-4.

Tabelle 29-4: Manipulatoren und Elementfunktionen zur Formatkontrolle

Manipulator	Elementfunktion	Beschreibung
dec	flags(10)	Setze Radix auf 10
hex	flags(16)	Setze Radix auf 16
Oct	flags(8)	Setze Radix auf 8
setfill(c)	fill(c)	Setze Füllzeichen auf c
setprecision(c)	precision(c)	Setze Genauigkeit auf c
setw(n)	width(n)	Setze Feldbreite auf n Zeichen

Sehen Sie nach dem Breitenparameter (Funktion `width( )` und Manipulator `setw( )`). Die meisten Parameter behalten ihren Wert, bis sie durch einen weiteren Aufruf neu gesetzt werden, aber der Breitenparameter verhält sich so nicht. Der Breitenparameter wird auf seinen Default-Wert gesetzt, sobald die nächste Ausgabe erfolgt. Sie könnten z.B. von dem Folgenden erwarten, dass Integerzahlen mit acht Zeichen erzeugt werden:

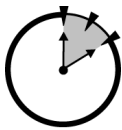
```
#include <iostream.h>
#include <iomanip.h>
void fn()
{
    cout << setw(8)      // Breite ist 8...
        << 10           // ... für die 10, aber...
        << 20           // ... default für die 20
        << »\n«;
}
```

Was Sie jedoch erhalten, ist eine Integerzahl mit acht Zeichen, gefolgt von einer Integerzahl mit zwei Zeichen. Um auch für die zweite Zahl acht Zeichen zu bekommen, ist das Folgende notwendig:

```
#include <iostream.h>
#include <iomanip.h>
void fn()
{
    cout << setw(8)      // setze die Breite ...
        << 10
        << setw(8)      // ... und setze sie wieder
        << 20
}
```

```
<< >>\n<<;
}
```

Was ist besser, Manipulatoren oder Aufrufe von Elementfunktionen? Elementfunktionen erlauben etwas mehr Kontrolle, weil es mehr davon gibt. Außerdem geben die Elementfunktionen die vorherigen Einstellungen zurück, was Sie nutzen können, um die Werte wieder zurückzusetzen, wenn Sie das möchten. Schließlich hat jede Funktion eine Version ohne Argumente, um den aktuellen Wert zurückzugeben, falls Sie die Einstellungen später wieder zurücksetzen möchten.



10 Min.

29.5 Benutzerdefinierte Inserters

Die Tatsache, dass C++ den Linksshiftoperator überlädt, um Ausgaben auszuführen, ist praktisch, weil Sie dadurch die Möglichkeit bekommen, denselben Operator für die Ausgabe der von Ihnen definierten Klassen zu überladen.

Betrachten Sie die Klasse `USDollar` noch einmal. Die folgende Version der Klasse enthält einen Inserter, der die gleiche Ausgabe erzeugt wie die frühere Version von `display()`:

```
// Inserter - stellt einen Inserter für USDollar
//          bereit
#include <stdio.h>
#include <iostream.h>
#include <iomanip.h>

// USDollar - repräsentiert den US Dollar
class USDollar
{
    friend ostream& operator<<(ostream& out, USDollar& d);
public:
    // ... keine Änderungen ...
};

// Inserter - Ausgabe als Zeichenkette
//          (diese Version behandelt den Fall, dass
//          die Cents kleiner als 10 sind)
ostream& operator<<(ostream& out, USDollar& d)
{
    char old = out.fill();
    out << >>$<<
        << d.nDollars
        << >>.<<
        << setfill('0') << setw(2)
        << d.nCents;

    // setze wieder das alte Füllzeichen
    out.fill(old);

    return out;
}

int main(int nArgc, char* pszArgs[])
{
    USDollar d1(1, 60);
```

344 **Sonntagnachmittag**

```

    cout << »Dollar d1 = » << d1 << »\n«;

    USDollar d2(1, 5);
    cout << »Dollar d2 = » << d2 << »\n«;

    return 0;
}

```

Der Inserter führt die gleichen elementaren Operationen aus wie die frühere Funktion `display()`, wobei hier direkt in das `ostream`-Ausgabeobjekt ausgegeben wird, das übergeben wurde. Die Funktion `main()` ist jedoch noch einfacher als vorher. Dieses Mal kann das `USDollar`-Objekt direkt in den Ausgabestream eingefügt werden.

Sie wundern sich vielleicht, warum `operator<<()` das `ostream`-Objekt zurückgibt, das übergeben wurde. Der Grund ist, dass dadurch Einfügeoperationen verkettet werden können. Weil `operator<<()` von links nach rechts bindet, wird der folgende Ausdruck

```

USDollar d1(1, 60);
cout << »Dollar d1 = » << d1 << »\n«;

```

interpretiert als

```

USDollar d1(1, 60);
((cout << »Dollar d1 = ») << d1) << »\n«;

```

Die erste Eingabe gibt die Zeichenkette »Dollar d1 = « nach `cout` aus. Das Ergebnis dieses Ausdrucks ist das Objekt `cout`, das dann an `operator<<(ostream&, USDollar&)` übergeben wird. Es ist wichtig, dass dieser Operator sein `ostream`-Objekt zurückgibt, so dass dieses Objekt an den nächsten Inserter übergeben werden kann, der das Zeilenendezeichen »\n« ausgibt.

29.6 Schlaue Inserter

Wir möchten die Inserter oft schlau machen. D.h. wir möchten gerne schreiben `cout << baseClassObjekt`, und dann C++ den passenden Inserter einer Unterklasse wählen lassen in der gleichen Weise, wie C++ die richtige virtuelle Elementfunktion gewählt hat. Weil der Inserter keine Elementfunktion ist, können wir ihn nicht direkt als `virtual` deklarieren.

Wir können das Problem leicht umgehen, indem wir den Inserter von einer virtuellen Funktion `display()` abhängig machen, wie im Programm `VirtualInsertor` in Listing 29-2 gezeigt wird.

Listing 29-2: Programm VirtualInsertor

```

// VirtualInsertor - basiere USDollar auf einer
//                  Basisklasse Currency (Währung);
//                  mache den Insertor virtual,
//                  indem er sich auf die Methode
//                  display() stützt
#include <stdio.h>
#include <iostream.h>
#include <iomanip.h>

// Currency - stellt Währung dar
class Currency

```

```

{
    friend ostream& operator<<(ostream& out, Currency& d);
public:
    Currency(int p = 0, int s = 0)
    {
        nPrimary = p;
        nSecondary = s;
    }

    // rationalize - normalisiere nSecondary durch
    //                Inkrementieren von nPrimary
    //                pro 100 in nSecondary
    void rationalize()
    {
        nPrimary += (nSecondary / 100);
        nSecondary %= 100;
    }

    // display - Schreiben des Objektes in das
    //                gegebene ostream-Objekt
    virtual ostream& display(ostream&) = 0;

protected:
    int nPrimary;
    int nSecondary;
};

// Inserter - Ausgabe als Zeichenkette
//                (diese Version behandelt den Fall, dass
//                nSecondary kleiner als 10 sind)
ostream& operator<<(ostream& out, Currency& c)
{
    return c.display(out);
}

// definiere USDollar als Unterklasse von Currency
class USDollar : public Currency
{
public:
    USDollar(int d, int c) : Currency(d, c)
    {
    }

    // Ausgaberoutine
    virtual ostream& display(ostream& out)
    {
        char old = out.fill();
        out << »$«
            << nPrimary
            << ».«
            << setfill('0') << setw(2)
            << nSecondary;

        // altes Füllzeichen aktivieren
        out.fill(old);
    }
}

```

346 Sonntagnachmittag

```

        return out;
    }
};

void fn(Currency& c, char* pszDescriptor)
{
    cout << pszDescriptor << c << »\n«;
}

int main(int nArgc, char* pszArgs[])
{
    // rufe USDollar::display() direkt auf
    USDollar d1(1, 60);
    cout << »Dollar d1 = » << d1 << »\n«;

    // rufe die gleiche Funktion virtuell auf
    // über die Funktion fn()
    USDollar d2(1, 5);
    fn(d2, »Dollar d2 = »);

    return 0;
}

```

Die Klasse `Currency` definiert eine Inserter-Funktion, die ein Nichtelement ist, und daher mit Polymorphie nichts zu tun hat. Aber statt wirklich etwas zu tun, stützt sich der Inserter auf eine virtuelle Elementfunktion `display()`, die die eigentliche Arbeit ausführt. Die Unterklasse `USDollar` muss nur die Funktion `display()` bereitstellen; das ist alles. Diese Version des Programms erzeugt die gleiche Ausgabe wie am Ende von Listing 29-1 zu sehen ist.

Dass die Einfügeoperation in der Tat polymorph ist, wird bei der Erzeugung der Ausgabefunktion `fn(Currency&, char*)` deutlich. Die Funktion `fn()` kennt nicht den Typ der Währung, den sie übergeben bekommt, und stellt die übergebene Währung mit den für `USDollar` geltenden Regeln dar. `main()` gibt `d1` direkt und `d2` über diese neue Funktion `fn()` aus. Die virtuelle Ausgabe von `fn()` sieht genauso aus, wie die des polymorphen Bruders.

Andere Unterklassen von `Currency`, wie z.B. `DMark`, `FFranc` oder `Euro`, können erzeugt werden, obwohl sie unterschiedliche Darstellungsregeln haben, indem einfach die entsprechenden `display()`-Funktionen bereitgestellt werden. Der Basiscode kann weiterhin ungestraft `Currency` verwenden.

29.7 Aber warum die Shift-Operatoren?

Sie können fragen »Warum die Shift-Operatoren für Stream-I/O verwenden? Warum nicht einen anderen Operator?«

Der Linksshift-Operator wurde aus mehreren Gründen gewählt. Erstens ist er ein binärer Operator. Das bedeutet, dass das `ostream`-Objekt das Argument auf der linken Seite, und das Ausgabeobjekt das Argument auf der rechten Seite sein kann. Zweitens ist der Links-Shift ein Operator auf einer niedrigen Ebene. Somit arbeiten Ausdrücke wie der folgende wie erwartet, weil Addition vor dem Einfügen ausgeführt wird:

```
cout << »a + b« << a + b << »\n«;
```

Drittens bindet der Linksshiftoperator von links nach rechts. Das erlaubt es uns, Ausgabeanweisungen zu verketten. Die vorige Zeile wird z.B. interpretiert als:

```
((cout << »a + b«) << a + b) << »\n«;
```



0 Min.

Trotz all dieser Gründe ist der eigentliche Grund sicherlich, dass es schön ist. Das doppelte Kleinerzeichen << sieht so aus, als wenn etwas den Code verlassen wollte, und das doppelte Größerzeichen >> sieht so aus, als wenn etwas hereinkommen wollte. Und, warum eigentlich nicht?

Zusammenfassung

Ich habe diese Sitzung mit einer Warnung begonnen, dass Stream-I/O zu komplex ist, um in einem Kapitel eines Buches abgehandelt zu werden. Sie können die Dokumentation Ihres Compilers bemühen, um eine vollständige Liste aller Elementfunktionen zu erhalten, die sie aufrufen können. Die relevanten Include-Dateien, wie `iostream.h` und `iomanip.h`, enthalten Prototypen mit erklärenden Kommentaren für alle Funktionen.

- Stream-I/O basiert auf den Klassen `istream` und `ostream`.
- Die Include-Datei `iostream.h` überlädt den Linksshift-Operator, um Ausgaben nach `ostream` auszuführen und überlädt den Rechtsshift-Operator, um Eingaben von `istream` auszuführen.
- Die Unterklasse `fstream` wird für Datei-I/O verwendet.
- Die Unterklasse `stringstream` führt I/O auf internen Speicherpuffern durch, unter Verwendung der gleichen Einfüge- und Extraktionsoperatoren.
- Der Programmierer kann die Einfüge- und Extraktionsoperatoren überladen für seine eigenen Klassen. Diese Operatoren können polymorph gemacht werden durch die Verwendung virtueller Zwischenmethoden.
- Die Manipulatorobjekte, die in `iomanip.h` definiert werden, können verwendet werden, um Formatfunktionen von `stream` aufzurufen.

Selbsttest

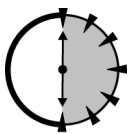
1. Wie werden die beiden Operatoren << und >> genannt, wenn sie für Stream-I/O verwendet werden? (Siehe »Wie funktioniert Stream-I/O?«)
2. Was ist die Basisklasse der beiden Default-I/O-Objekte `cout` und `cin`? (Siehe »Wie funktioniert Stream-I/O?«)
3. Wofür wird die Klasse `fstream` verwendet? (Siehe »Die Unterklassen `fstream`«)
4. Wofür wird die Klasse `stringstream` verwendet? (Siehe »Die Unterklassen `stringstream`«)
5. Welcher Manipulator setzt den numerischen Ausgabemodus auf hexadezimal? Was ist die zugehörige Elementfunktion? (Siehe »Manipulatoren«)



Ausnahmen

Checkliste

- ☒ Fehlerbedingungen zurückgeben
- ☒ Ausnahmen verwenden, ein neuer Mechanismus zur Fehlerbehandlung
- ☒ Auslösen und Abfangen von Ausnahmen
- ☒ Überladen der Ausnahmeklasse



30 Min.

Zusätzlich zu dem allgegenwärtigen Ansatz der Fehlerausgaben enthält C++ einen einfacheren und verlässlicheren Mechanismus zur Fehlerbehandlung. Diese Technik, die Ausnahmebehandlung genannt wird, ist der Gegenstand dieser Sitzung.

30.1 Konventionelle Fehlerbehandlung

Eine Implementierung des allgemeinen Beispiels der Fakultät sieht folgendermaßen aus.



CD-ROM

Die Fakultätsfunktion finden Sie auf der beiliegenden CD-ROM im Programm FactorialProgram.cpp.

```
// factorial - berechne die Fakultät von nBase, die
//             gleich nBase * (nBase - 1) *
//             (nBase - 2) * ... ist
int factorial(int nBase)
{
    // starte mit Wert 1
    int nFactorial = 1;

    // Schleife von nBase bis 1, jedes Mal den
    // Produktwert mit dem aktuellen Wert
    // multiplizieren
```



```
do
{
    nFactorial *= nBase;
} while (-nBase > 1);

// return the result
return nFactorial;
}
```

Obwohl die Funktion sehr einfach ist, fehlt ihr ein kritisches Feature: Die Fakultät von 0 ist 1, während die Fakultät einer negativen Zahl nicht definiert ist. Die obige Funktion sollte einen Test enthalten für negative Argumente und eine Fehlermeldung ausgeben, falls ein solches übergeben wird.

Der klassische Weg, einen Fehler in einer Funktion anzuzeigen, ist die Rückgabe eines Wertes, der sonst nicht von der Funktion zurückgegeben werden kann. Z.B. ist es nicht möglich, dass die Fakultät negativ ist. Wenn der Funktion also ein negativer Wert übergeben wird, könnte sie z.B. -1 zurückgeben. Die aufrufende Funktion kann den Rückgabewert überprüfen – wenn er negativ ist, weiß die aufrufende Funktion, dass ein Fehler aufgetreten ist und kann eine entsprechende Aktion auslösen (was immer das dann ist).

Das ist die Art und Weise der Fehlerbehandlung, die seit den frühen Tagen von FORTRAN praktiziert wurde. Warum sollte das geändert werden?

30.2 Warum benötigen wir einen neuen Fehlermechanismus?

Es gibt verschiedene Probleme mit dem Ansatz der Fehlerausgaben. Zum einen ist nicht jede Funktion in der glücklichen Lage wie die Fakultätsfunktion, dass keine negativen Werte zurückkommen können. Nehmen Sie z.B. den Logarithmus. Sie können den Logarithmus nicht für eine negative Zahl berechnen, aber der Logarithmus kann positiv und negativ sein – es gibt keinen Wert, der von einer Funktion `logarithm( )` zurückgegeben werden könnte, der nicht ein gültiger Logarithmuswert ist.

Zweitens gibt es zu viele Informationen, die in einem Integerwert gespeichert werden müssten. Z.B. -1 für »Argument ist negativ« und -2 für »Argument zu groß«, aber wenn das Argument zu groß ist, gibt es keine Möglichkeit, das Ergebnis zurückzugeben. Die Kenntnis dieses Wertes würde aber vielleicht zur Lösung des Problems beitragen. Es gibt keine Möglichkeit, als nur einen einzigen Rückgabewert zu speichern.

Drittens ist die Behandlung von Fehlern optional. Nehmen Sie an, jemand schreibt `factorial( )` so, dass sie die Argumente überprüft und einen negativen Wert zurückgibt, wenn das Argument negativ ist. Wenn der Code, der diese Funktion benutzt, den Rückgabewert nicht überprüft, hilft das gar nichts. Natürlich sprechen wir alle möglichen Drohungen aus »Sie werden Ihre Fehlerrückgaben überprüfen oder ...« aber wir wissen alle, dass die Sprache (und Ihr Chef) nichts tun kann, wenn Sie es unterlassen.

Selbst wenn ich die Fehlerrückgabe von `factorial( )` oder einer anderen Funktion überprüfe, was kann meine Funktion mit dem Fehler anfangen? Sicherlich nicht mehr, als eine Fehlermeldung auszugeben oder selber einen Fehlercode an die aufrufende Funktion zurückzugeben. Schnell sieht der Code dann so aus:

350 **Sonntagnachmittag**

```
// rufe eine Funktion auf, überprüfe den
// Fehlerrückgabewert, behandle ihn und kehre zurück
int nErrorRtn = someFunc();
if (nErrorRtn)
{
    errorOut(>Fehler beim Aufruf von someFunc()<);
    return MY_ERROR_1;
}

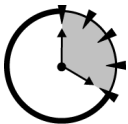
nErrorRtn = someOtherFunc();
if (nErrorRtn)
{
    errorOut(>Fehler beim Aufruf von someOtherFunc()<);
    return MY_ERROR_2;
}
```

Dieser Mechanismus hat mehrere Probleme:

- Er wiederholt vieles.
- Er zwingt den Benutzer, verschiedene Fehlermeldungen zu erfinden und abzufangen.
- Er mischt den Code zur Fehlerbehandlung und den normalen Codefluss, wodurch beide unleserlicher werden.

Diese Probleme scheinen in diesem einfachen Beispiel nicht so schlimm zu sein, aber die Komplexität nimmt rapide zu, wenn die Komplexität des aufrufenden Codes zunimmt. Nach einer Weile gibt es mehr Code zur Fehlerbehandlung als »eigentlichen« Code.

Das Ergebnis ist, dass Code zur Fehlerbehandlung nicht so geschrieben wird, dass alle Bedingungen erfasst sind, die erfasst werden müssen.



20 Min.

30.3 Wie arbeiten Ausnahmen?

C++ führt einen total neuen Mechanismus zum Abfangen und Behandeln von Fehlern ein. Dieser Mechanismus wird als Ausnahmen bezeichnet und basiert auf den Schlüsselworten `try`, `throw` und `catch`. Er arbeitet etwa so: eine Funktion versucht (`try`), durch ein Stück Code hindurch zu kommen. Wenn der Code ein Problem entdeckt, löst es einen Fehler aus (`throw`), der von der Funktion abgefangen werden kann (`catch`).

Listing 30-1 zeigt, wie Ausnahmen arbeiten.

Listing 30-1 zeigt, wie Ausnahmen arbeiten.

Listing 30-1: Ausnahmen in Aktion

```
// FactorialExceptionProgram - Ausgabe der Fakultät
//                               mit Ausnahme-basierter
//                               Fehlerbehandlung
#include <stdio.h>
#include <iostream.h>

// factorial - berechne die Fakultät von nBase, die
//               gleich nBase * (nBase - 1) *
//               (nBase - 2) * ... ist
int factorial(int nBase)
```

Lektion 30 – Ausnahmen 351

```

{
    // wenn nBase < 0...
    if (nBase <= 0)
    {
        // ... löse einen Fehler aus
        throw »Ausnahme ungültiges Argument«;
    }

    int nFactorial = 1;
    do
    {
        nFactorial *= nBase;
    } while (-nBase > 1);

    // return the result
    return nFactorial;
}

int main(int nArgc, char* pszArgs[])
{
    // rufe factorial in einer Schleife auf,
    // fange alle Ausnahmen ab, die die Funktion
    // auslösen könnte
    try
    {
        for (int i = 6; ; i-)
        {
            cout << »factorial(«
                << i
                << ») = «
                << factorial(i)
                << »\n«;
        }
    }
    catch(char* pErrorMsg)
    {
        cout << »Fehler: «
            << pErrorMsg
            << »\n«;
    }
    return 0;
}

```

Die Funktion `main()` beginnt mit einem Block, der mit dem Schlüsselwort `try` markiert ist. Einer oder mehr `catch`-Blöcke stehen unmittelbar hinter dem `try`-Block. Das Schlüsselwort `try` wird von einem einzigen Argument gefolgt, das wie eine Funktionsdefinition aussieht.

Innerhalb des `try`-Blocks kann `main()` tun, was sie will. In diesem Fall geht `main()` in eine Schleife, die die Fakultät von absteigenden Zahlen berechnet. Schließlich übergibt das Programm eine negative Zahl an die Funktion `factorial()`.

Wenn unsere schlaue Funktion `factorial()` eine solche falsche Anfrage bekommt, »wirft« sie eine Zeichenkette, die eine Beschreibung des Fehlers enthält, unter Verwendung des Schlüsselwortes `throw`.

352 **Sonntagnachmittag**

An diesem Punkt sucht C++ nach einem `catch`-Block, dessen Argument zu dem ausgelösten Objekt passt. Der Rest des `try`-Blocks wird nicht fertig abgearbeitet. Wenn C++ kein `catch` in der aktuellen Funktion findet, kehrt C++ zum Ausgangspunkt des Aufrufes zurück und setzt die Suche dort fort. Der Prozess wird fortgesetzt, bis ein passender `catch`-Block gefunden wird oder die Kontrolle `main()` verlässt.

In diesem Beispiel wird die ausgelöste Fehlermeldung durch den `catch`-Block am Ende der Funktion `main()` abgefangen, der eine Meldung ausgibt. Die nächste Anweisung ist das `return`-Kommando, wodurch das Programm beendet wird.

30.3.1 Warum ist der Ausnahmemechanismus eine Verbesserung?

Der Ausnahmemechanismus löst die Probleme, die dem Mechanismus der Fehlerausgaben inhärent sind, indem der Pfad zur Fehlerbehandlung vom Pfad des normalen Codes getrennt wird. Außerdem machen Ausnahmen die Fehlerbehandlung zwingend. Wenn Ihre Funktion die ausgelöste Ausnahme nicht verarbeitet, geht die Kontrolle die Kette der aufrufenden Funktionen nach oben, bis C++ eine Funktion findet, die diese Ausnahme behandeln kann. Das gibt Ihnen auch die Flexibilität, Fehler zu ignorieren, bei denen Sie nichts machen können. Nur die Funktionen, die das Problem beheben können, müssen die Ausnahme abfangen. Und wie funktioniert das?

30.4 Abfangen von Details, die für mich bestimmt sind

Lassen Sie uns die Schritte genauer ansehen, die der Code durchlaufen muss, um eine Ausnahme zu verarbeiten. Wenn ein `throw` ausgeführt wird, kopiert C++ das ausgelöste Objekt an einen neutralen Ort. Dann wird das Ende des `try`-Blocks untersucht. C++ sucht nach einem passenden `catch` irgendwo in der Kette der Funktionsaufrufe. Dieser Prozess wird »Abwickeln des Stacks« genannt.

Ein wichtiges Feature des Stackabwickelns ist, dass jedes Objekt, das seine Gültigkeit verliert, vernichtet wird, genauso als wenn die Funktion eine `return`-Anweisung ausgeführt hätte. Das verhindert, dass das Programm Ressourcen verliert, oder Objekte »in der Luft hängen«.

Listing 30-2 ist ein Beispielprogramm, das den Stack abwickelt:

Listing 30-2: Abwickeln des Stacks

```
#include <iostream.h>
class Obj
{
public:
    Obj(char c)
    {
        label = c;
        cout << »Konstruktor » << label << endl;
    }
    ~Obj()
    {
        cout << »Destruktor » << label << endl;
    }

protected:
    char label;
};
void f1();
```

```
void f2();

int main(int, char*[])
{
    Obj a('a');
    try
    {
        Obj b('b');
        f1();
    }
    catch(float f)
    {
        cout << »float abgefangen« << endl;
    }
    catch(int i)
    {
        cout << »int abgefangen« << endl;
    }
    catch(...)
    {
        cout << »generisches catch« << endl;
    }
    return 0;
}

void f1()
{
    try
    {
        Obj c('c');
        f2();
    }
    catch(char* pMsg)
    {
        cout << »Zeichenkette abgefangen« << endl;
    }
}

void f2()
{
    Obj d('d');
    throw 10;
}
```

Ausgabe:

```
Konstruktor a
Konstruktor b
Konstruktor c
Konstruktor d
Destruktor d
Destruktor c
Destruktor b
int abgefangen
Destruktor a
```

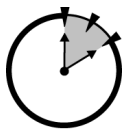
354 **Sonntagnachmittag**

Zuerst werden die vier Objekte `a`, `b`, `c` und `d` konstruiert, wenn die Kontrolle ihre Deklarationen antrifft, bevor `f2()` die `int 10` auslöst. Weil kein `try`-Block definiert ist innerhalb von `f2()`, wickelt C++ den Stack von `f2` ab, was zur Vernichtung von `d` führt. `f1()` definiert einen `try`-Block, aber ihr einziger `catch`-Block verarbeitet `char*`, und passt daher nicht zu dem ausgelösten `int`, und C++ setzt die Suche fort. Das wickelt den Stack von `f1()` ab, wodurch das Objekt `c` vernichtet wird.

Zurück in `main()` findet C++ einen weiteren `try`-Block. Das Verlassen dieses Block vernichtet `b`. Der erste `catch`-Block verarbeitet `float`, und passt daher nicht. Der nächste `catch`-Block passt exakt. Der letzte `catch`-Block, der jedes beliebige Objekt verarbeiten würde, wird nicht mehr betreten, weil bereits ein passender `catch`-Block gefunden wurde.



Eine Funktion, die als `fn(...)` deklariert ist, akzeptiert eine beliebige Anzahl von Argumenten mit beliebigem Typ. Das gleiche gilt für `catch`-Blöcke. Ein `catch(...)` fängt alles ab.



10 Min.

30.4.1 Was kann ich werfen?

Ein C++-Programm kann jeden beliebigen Typ von Objekten auslösen. C++ verwendet einfache Regeln, um einen passenden `catch`-Block zu finden.

C++ untersucht zuerst die `catch`-Blöcke, die direkt hinter dem `try`-Block stehen. Die `catch`-Blöcke werden der Reihe nach betrachtet, bis ein Block gefunden wurde, der zu dem ausgelösten Objekt passt. Ein »Treffer« wird mit den gleichen Regeln definiert, wie ein Treffer für Argumente in einer überladenen Funktion. Wenn kein passender `catch`-Block gefunden wurde, geht der Code zu den `catch`-Blocks im nächsthöheren Level, wie auf einer Spirale nach oben, bis ein passender `catch`-Block gefunden wird. Wenn kein `catch`-Block gefunden wird, beendet sich das Programm.

Die frühere Funktion `factorial()` wirft eine Zeichenkette, d.h. ein Objekt von Typ `char*`. Die zugehörige `catch`-Deklaration passt, weil sie verspricht, ein Objekt vom Typ `char*` zu verarbeiten. Eine Funktion kann jedoch ein Objekt eines beliebigen Typs auslösen.

Ein Programm kann das Ergebnis eines Ausdrucks auslösen. Das bedeutet, dass Sie so viel Information mitgeben können, wie sie möchten. Betrachten Sie die folgende Klassendefinition:

```
#include <iostream.h>
#include <string.h>
// Exception - generische Klasse für Fehlerbehandlung
class Exception
{
public:
    // konstruiere ein Ausnahmeobjekt mit einem
    // Beschreibungstext des Problems, zusammen mit
    // der Datei und der Zeilennummer, wo das
    // Problem aufgetreten ist
    Exception(char* pMsg, char* pFile, int nLine)
    {
        this->pMsg = new char[strlen(pMsg) + 1];
        strcpy(this->pMsg, pMsg);

        strncpy(file, pFile, sizeof file);
    }
};
```

Lektion 30 – Ausnahmen**355**

```

        file[sizeof file - 1] = '\0';
        lineNum = nLine;
    }

    // display - Ausgabe des Inhalts des aktuellen
    // Objektes in den Ausgabestream
    virtual void display(ostream& out)
    {
        out << »Fehler << << pMsg << »>>\n<<;
        out << »in Zeile #<<
            << lineNum
            << », in Datei »
            << file
            << endl;
    }

protected:
    // Fehlermeldung
    char* pMsg;

    // Dateiname und Zeilennummer des Fehlers
    char file[80];
    int lineNum;
};

```

Das entsprechende throw sieht wie folgt aus:

```

throw Exception(»Negatives Argument für factorial«,
               __FILE__,
               __LINE__);

```



**FILE__ und LINE__ sind elementare #defines, die auf den Namen der Quellda-
tei und die aktuelle Zeile in der Datei gesetzt sind.**

Der zugehörige catch-Block sieht so aus:

```

void myFunc()
{
    try
    {
        //... was auch immer aufgerufen wird
    }

    // fange ein Exception-Objekt ab
    catch(Exception x)
    {
        // verwende die eingebaute Elementfunktion
        // display zur Anzeige des Objektes im
        // Fehlerstream
        x.display(cerr);
    }
}

```

356 Sonntagnachmittag

Der `catch`-Block nimmt das `Exception`-Objekt und verwendet dann die eingebaute Elementfunktion `display()` zur Anzeige der Fehlermeldung.



Die Version von `factorial`, die von der Klasse `Exception` Gebrauch macht, ist auf der beiliegenden CD-ROM als `FactorialThrow.cpp` enthalten.

Die Ausgabe bei Ausführung des Fakultätsprogramms sieht so aus:

```
factorial(6) = 720
factorial(5) = 120
factorial(4) = 24
factorial(3) = 6
factorial(2) = 2
factorial(1) = 1
Error <Negatives Argument für factorial>
in Zeile #59,
in Datei C:\wecc\Programs\lesson30\FactorialThrow.cpp
```

Die Klasse `Exception` stellt eine generische Klasse zum Melden von Fehlern dar. Sie können Unterklassen von dieser Klasse ableiten. Ich könnte z.B. eine Klasse `InvalidArgumentException` ableiten, die den unzulässigen Argumentwert speichert, zusammen mit dem Fehlertext und dem Ort des Fehlers.

```
class InvalidArgumentException : public Exception
{
public:
    InvalidArgumentException(int arg, char* pFile,
                             int nLine)
        : Exception(»Unzulässiges Argument«, pFile, nLine)
    {
        invArg = arg;
    }

    // display - Ausgabe des Objektes in das
    // angegebene Ausgabeobjekt
    virtual void display(ostream& out)
    {
        // die Basisklasse gibt ihre
        // Informationen aus ...
        Exception::display(out);

        // ... und dann sind wir dran
        out << »Argument war »
            << invArg
            << »\n«;
    }

protected:
    int invArg;
};
```


Die aufrufende Funktion behandelt die neue Klasse `InvalidArgumentException` automatisch, weil `InvalidArgumentException` eine `Exception` ist und die Elementfunktion `display()` polymorph ist.



Die Funktion `InvalidArgumentException::display()` **stützt sich auf die Basisklasse** `Exception`, **um den Teil des Objektes auszugeben, der von `Exception` stammt.**

30.5 Verketteten von catch-Blöcken

Ein Programm kann seine Flexibilität in der Fehlerbehandlung dadurch erhöhen, indem mehrere `catch`-Blöcke an den gleichen `try`-Block angefügt werden. Der folgende Codeschnipsel demonstriert das Konzept:

```
void myFunc()
{
    try
    {
        // ... was auch immer aufgerufen wird
    }
    // fange eine Zeichenkette ab
    catch(char* pszString)
    {
        cout << »Fehler: » << pszString << »\n«;
    }
    // fange ein Exception-Objekt ab
    catch(Exception x)
    {
        x.display(cerr);
    }

    // ... Ausführung wird hier fortgesetzt ...
}
```

In diesem Beispiel wird ein ausgelöstes Objekt, wenn es eine einfache Zeichenkette ist, vom ersten `catch`-Block abgefangen, der die Zeichenkette ausgibt. Wenn das Objekt keine Zeichenkette ist, wird es mit der `Exception`-Klasse verglichen. Wenn das Objekt eine `Exception` ist oder aus einer Unterklasse von `Exception` stammt, wird es vom zweiten `catch`-Block verarbeitet.

Weil sich dieser Prozess seriell vorgeht, muss der Programmierer mit den spezielleren Objekttypen anfangen und bei den allgemeineren aufhören. Es ist daher ein Fehler, das Folgende zu tun:

```
void myFunc()
{
    try
    {
        // ... was auch immer aufgerufen wird
    }
    catch(Exception x)
    {
        x.display(cerr);
    }
}
```

358 Sonntagnachmittag

```

    }
    catch(InvalidArgumentException x)
    {
        x.display(cerr);
    }
}

```

**0 Min.**

Weil eine `InvalidArgumentException` eine `Exception` ist, wird die Kontrolle den zweiten `catch`-Block nie erreichen.

**Tipp**

Der Compiler fängt diesen Programmierfehler nicht ab.

**Hinweis**

Es macht im obigen Beispiel auch keinen Unterschied, dass `display()` virtuell ist. Der `catch`-Block für `Exception` ruft die Funktion `display()` in Abhängigkeit vom Laufzeittyp des Objektes auf.

**Hinweis**

Weil der generische `catch`-Block `catch(...)` jede Ausnahme abfängt, muss er als letzter in einer Reihe von `catch`-Blöcken stehen. Jeder `catch`-Block hinter einem generischen `catch` ist unerreichbar.

Zusammenfassung

Der Ausnahmemechanismus von C++ stellt einen einfachen, kontrollierten und erweiterbaren Mechanismus zur Fehlerbehandlung dar. Er vermeidet die logische Komplexität, die bei dem Standardmechanismus der Fehlerrückgabewerte entstehen kann. Er stellt außerdem sicher, dass Objekte korrekt vernichtet werden, wenn sie ihre Gültigkeit verlieren.

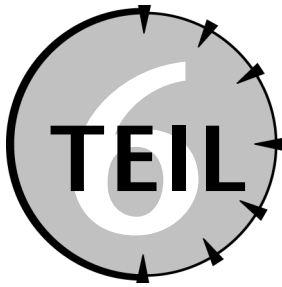
- Die konventionelle Technik, einen ansonsten ungültigen Wert zurückzugeben, um dem Typ des Fehlers anzuzeigen, hat ernsthafte Begrenzungen. Erstens kann nur eine begrenzte Menge an Information kodiert werden. Zweitens ist die aufrufende Funktion gezwungen, den Fehler zu behandeln, indem er verarbeitet oder zurückgegeben wird, ob sie nun etwas an dem Fehler machen kann oder nicht. Schließlich haben viele Funktionen keinen ungültigen Wert, der so zurückgegeben werden könnte.
- Die Technik der Ausnahmefehler ermöglicht es Funktionen, eine theoretisch nicht begrenzte Anzahl von Informationen zurückzugeben. Wenn eine aufrufende Funktion einen Fehler ignoriert, wird der Fehler die Kette der Funktionsaufrufe nach oben propagiert, bis eine Funktion gefunden wird, die den Fehler behandeln kann.

Lektion 30 – Ausnahmen 359

- Ausnahmen können Unterklassen sein, was die Flexibilität für den Programmierer erhöht.
- Es können mehrere `catch`-Blöcke verkettet werden, um die aufrufende Funktion in die Lage zu versetzen, verschiedene Fehlertypen zu behandeln.

Selbsttest

1. Nennen Sie drei Begrenzungen der Fehlerrückgabetechnik. (Siehe »Konventionelle Fehlerbehandlung«)
2. Nennen Sie die drei Schlüsselwörter, die von der Technik der Ausnahmebehandlung verwendet werden. (Siehe »Wie arbeiten Ausnahmen?«)



Sonntagnachmittag – Zusammenfassung

1. **Schreiben Sie den Kopierkonstruktor und den Zuweisungsoperator für das Modul Assign-Problem. Eine Ressource muss geöffnet werden mit dem nValue-Wert des MyClass-Objekts, und diese Ressource muss**

- **geöffnet werden bevor sie gültig ist**
- **geschlossen werden nachdem sie geöffnet wurde**
- **geschlossen werden, bevor sie wieder geöffnet werden kann**

Nehmen Sie an, dass die Prototypfunktionen irgendwo anders definiert sind.

```
// AssignProblem - demonstriert Zuweisungsoperator
#include <stdio.h>
#include <iostream.h>

class MyClass;

// Resource - die Funktion open() bereitet das
//           Objekt auf seinen Gebrauch vor,
//           die Funktion close() »tut es weg«;
//           eine Ressource muss geschlossen werden,
//           bevor sie wieder geöffnet werden kann
class Resource
{
public:
    Resource();
    void open(int nValue);
    void close();
};

// MyClass - soll vor Gebrauch geöffnet und nach
//           Gebrauch geschlossen werden
class MyClass
{
public:
    MyClass(int nValue)
    {
        resource.open(nValue);
    }

    ~MyClass()
    {

```

```

        resource.close();
    }

    // Kopierkonstruktor und Zuweisungsoperator
    MyClass(MyClass& mc)
    {
        // ... was kommt hier hin?
    }
    MyClass& operator=(MyClass& s)
    {
        // ...und was hier?
    }

protected:
    Resource resource;

    // der Wert für resource.open()
    int nValue;
};

```

2. **Schreiben Sie einen Inserter für das folgende Programm, das den Nachnamen, den Vornamen und Studenten-ID für die folgende Student-Klasse ausgibt.**

```

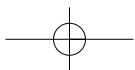
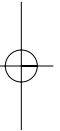
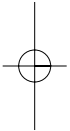
// StudentInsertor
#include <stdio.h>
#include <iostream.h>
#include <string.h>

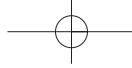
// Student
class Student
{
public:
    Student(char* pszFName, char* pszLName, int nSSNum)
    {
        strncpy(szFName, pszFName, 20);
        strncpy(szLName, pszLName, 20);
        this->nSSNum = nSSNum;
    }

protected:
    char szLName[20];
    char szFName[20];
    int nSSNum;
};

int main(int nArgc, char* pszArgs[])
{
    Student student(»Kinsey«, »Lee«, 1234);
    cout << »Mein Freund ist » << student << »\n«;
    return 0;
}

```





Antworten auf die Wiederholungsfragen



Freitagabend

1. *Unsere Lösung ist einfacher als das Programm zum Entfernen von Reifen. Wir müssen den Wagenheber nicht erst greifen. Weil das Auto bereits in der Luft ist, gehalten vom Wagenheber, können wir davon ausgehen, dass wir wissen, wo sich der Wagenheber befindet.*
 1. Greife Griff des Wagenhebers
 2. während Auto nicht auf Boden
 3. bewege Griff des Wagenhebers nach unten
 4. bewege Griff des Wagenhebers nach oben
 5. lasse Griff des Wagenhebers los
2. *Das Entfernen des Minuszeichens zwischen 212 und 32 veranlasst Visual C++ zur fehlerhaften Fehlermeldung »missing ;«.*
3. *Ich habe das Problem behoben, indem ich ein Semikolon nach 212 eingefügt habe und das Programm neu erzeugt habe. Visual C++ erzeugt das »korrigierte« Programm ohne Beanstandung.*
4. *Das korrigierte Programm berechnet eine Temperatur in Fahrenheit von 244, was offensichtlich falsch ist.*
5. *Offensichtlich sind nFactor = 212; und 32; legale Kommandos. Der falsche Ergebniswert von nFactor kommt von der fehlerhaften Durchführung der Konvertierung.*
6. *Ein Minuszeichen zu vergessen, ist ein verständlicher Fehler für einen so schlechten Tipper wie mich. Hätte Visual C++ den Fehler so behoben wie es dachte, dass ein Semikolon fehlt, wäre das Programm ohne Fehler erzeugt worden; die Ausführung liefert jedoch ein falsches Ergebnis. Ich hätte selber durch das Programm gehen müssen, um den Fehler in der Berechnung zu finden. Das schafft Misstrauen in die Umrechnungsformel zwischen Celsius und Fahrenheit, während das eigentliche Problem ein Tippfehler ist. Das ist Zeitverschwendung, wenn doch die Fehlermeldung von Visual C++, wenn auch nicht korrekt, mich direkt zum Ort des Fehlers führt.*

364 Anhang A**7. Erneutes Erzeugen von Conversion.cpp nach dem Entfernen der Hochkommata erzeugt diese Fehlermeldung:**

Conversion.cpp(31) Error: unterminated string or character constant

Diese Fehlermeldung zeigt, dass GNU C++ denkt, dass der Fehler in Zeile 31 aufgetreten ist. (Das ist die Bedeutung von »31« in der Fehlermeldung.)

8. Weil GNU C++ keine Hochkommata gefunden hat, um die Zeichenkette in Zeile 28 zu beenden, dachte es, dass die Zeichenkette bis zu den nächsten Hochkommata geht, was in Zeile 31 der Fall ist. (Diese Hochkommata sind aber tatsächlich der Anfang einer neuen Zeichenkette, aber GNU C++ weiß das nicht.)**Samstagmorgen****1.**

- *Teilen von nTons durch 1,1 verursacht einen Rundungsfehler.*
- *2/1,1 ist gleich 1,8, was auf 1 abgerundet wird. Die Funktion gibt 1000 kg zurück.*
- *Das Ergebnis der Division von 2 durch 1,1 ist ein double-Wert. Die Zuweisung dieses Werts an nLongTons resultiert in einer Demotion, die vom Compiler bemerkt werden sollte. Zusatz: »Assign double to int. Possible loss of significance.« oder etwas in dieser Art.*

2. Die folgende Funktion hat weniger Probleme mit Rundungsfehlern als ihr Vorgänger:

```
int ton2kg(int nTons)
{
    return (nTons * 1000) / 1.1;
}
```

3.

- *5000 Tonnen werden in 4.500.0000.0000 g konvertiert. Diese Zahl liegt außerhalb des Bereiches von int auf einem Intel-basierten PC.*
- *Die einzige mögliche Lösung ist die Rückgabe eines float oder eines double anstelle von int. Der Bereich von float ist viel größer als der Bereich von int.*

4.

- *falsch*
- *wahr*
- *wahr*
- *unbestimmt, aber wahrscheinlich wahr*
Sie können nicht darauf zählen, dass zwei unabhängig voneinander berechnete Gleitkom-mavariablen gleich sind.

Antworten auf die Wiederholungsfragen 365

- falsch

Der Wert von `n4` ist gleich 4. Weil die linke Seite von `&&` falsch ist, wird die rechte Seite nicht ausgewertet und `==` wird nie ausgeführt (siehe den Abschnitt über kurze Schaltkreise).

5.

- 0x5D

- 93

- 1011 10102

Es ist am einfachsten, die Addition binär auszuführen. Erinnern Sie sich an die Regeln:

0 + 0 -> 0

1 + 0 -> 1

0 + 1 -> 1

1 + 1 -> 0, übertrage die 1

Alternativ konvertieren Sie die $93 * 2 = 186$ zurück ins Binärformat.

*Zusatz: 0101 11012 * 2 hat das gleiche Bitmuster, nur um eine Position nach links geschiftet, und eine 0 an der Position ganz rechts.*

- 0101 11112

Konvertiere 2 ins Binärformat 0000 00102 und verknüpfe die beiden Zahlen mit OR.

- wahr

Das Bit 0000 00102 ist nicht gesetzt. Somit ergibt eine AND-Verknüpfung der beiden Null (0).

6. *Erinnern Sie sich daran, dass C++ Leerraum ignoriert, Tabulatoren eingeschlossen. Während der `else`-Zweig scheinbar zum äußeren `if` gehört, gehört es aber tatsächlich zum inneren `if`, so, als wenn es so geschrieben worden wäre:*

```
int n1 = 10;
if (n1 > 11)
{
    if (n1 > 12)
    {
        n1 = 0;
    }
    else
    {
        n1 = 1;
    }
}
```

Die äußere `if`-Anweisung hat keinen `else`-Zweig, und `n1` bleibt daher unverändert.

366 Anhang A

7. Weil `n1` nicht kleiner als 5 ist, wird der Body von `while( )` nie ausgeführt. Im Falle von `do...while( )` wird der Body einmal ausgeführt, obwohl `n1` nicht kleiner ist als 5. In diesem Fall erhält `n1` den Wert 11.
Der Unterschied zwischen den beiden Schleifen ist, dass `do...while( )` seinen Body immer mindestens einmal ausführt, selbst wenn die Bedingung gleich zu Beginn falsch ist.

8.

```
double cube(double d)
{
    return d * d * d;
}
```

Weil der Intel-Prozessor in Ihrem PC Integerzahlen und Gleitkommazahlen unterschiedlich behandelt, ist der Maschinencode, der von den Funktionen `cube(int)` und `cube(double)` erzeugt wird, voneinander verschieden.

9. Der Ausdruck `cube(3.0)` passt zu der Funktion `cube(double)`; somit wird `cube(double)` der Wert 3.0 übergeben, die den Wert 9.0 zurückgibt, der zu 9 demotiert wird und der Variablen `n` zugewiesen wird. Obwohl das Ergebnis das gleiche ist, ist der Weg dorthin verschieden.
10. Der Compiler erzeugt einen Fehler, weil die erste Funktion `cube(int)` und diese neue Funktion den gleichen Namen haben. Erinnern Sie sich daran, dass der Rückgabetyt nicht Teil des Namens ist.

Samstagnachmittag

1.

```
class Student
{
    public:
        char szLastName[128];
        int nGrade; // 1-> 1. Grad, 2-> 2. Grad ...
        double dAverage;
}
```

2.

```
void readAndDisplay()
{
    Student s;

    // Eingabe Studenteninformation
    cout << »Name des Studenten:<<;
    cin.getline(s.szLastName);
    cout << »Grad (1, 2, ...)\\n<<;
    cin >> s.nGrade;
    cout << »Durchschnitt:<<;
    cin >> s.dGPA;

    // Ausgabe der Studenteninformationen
    cout << »\\nStudenteninformationen:\\n<<;
    cout << s.szLastName << »\\n<<;
    cout << s.nGrade << »\\n<<;
    cout << s.dAverage << »\\n<<;
}
```

3.

```
void readAndDisplayAverage()
{
    Student s;

    // Eingabe Studenteninformationen
    cout << »Name des Stundenten:<<;
    cin.getline(s.szLastName);
    cout << »Grad (1, 2, ...)\\n<<;
    cin >> s.nGrade;
    cout << »Durchschnitt:<<;

    // drei Grade, die gemittelt werden
    double dGrades[3];
    cin >> dGrade[0];
    cin >> dGrade[1];
    cin >> dGrade[2];
    s.dAverage = (dGrade[0] + dGrade[1] + dGrade[2]) / 3;

    // Ausgabe der Studenteninformationen
    cout << »\\nStudenteninformationen:\\n<<;
    cout << s.szLastName << »\\n<<;
    cout << s.nGrade << »\\n<<;
    cout << s.dAverage << »\\n<<;
}
```

368 Anhang A

4.

- 16 Bytes ($4 + 4 + 4$)
- 80 Bytes ($4 * 20$)
- 8 ($4 + 4$) *Erinnern Sie sich daran, dass die Größe eines Zeigers 4 Bytes ist, unabhängig davon, auf was der Zeiger zeigt.*

5.

- Ja.
- Sie alloziert Speicher vom Heap, gibt diesen aber nicht zurück, bevor die Funktion verlassen wird (so etwas wird als Speicherloch bezeichnet).
- Bei jedem Aufruf geht ein wenig Speicher verloren, bis der Heap aufgebraucht ist.
- Es kann sehr lange dauern, bis der Speicher aufgebraucht ist. Bei einem solch kleinen Speicherloch muss das Programm viele Stunden laufen, nur damit das Problem überhaupt sichtbar wird.

6. `dArray[0]` ist bei `0x100`, `dArray[1]` ist bei `0x108` und `dArray[2]` ist bei `0x110`. Das Array erstreckt sich von `0x100` bis `0x118`.

7. Zuweisung 1 hat den gleichen Effekt wie `dArray[1] = 1,0`. Die Zuweisung 2 zerstört den Gleitkommawert in `dArray[2]`, das ist aber nicht fatal, weil 4 Bytes, die für `int` benötigt werden, in die 8 Bytes, die für `double` alloziert wurden, hineinpassen.

8.

```
LinkableClass* removeHead()
{
    LinkableClass* pFirstEntry;
    pFirstEntry = pHead;
    if (pHead != 0)
    {
        pHead = pHead->pNext;
    }
    return pFirstEntry;
}
```

Die Funktion `removeHead()` überprüft erst, ob der Kopfzeiger null ist. Wenn dies der Fall ist, ist die Liste bereits leer. Wenn nicht, speichert die Funktion einen Zeiger auf das erste Element in `pFirstEntry`. Sie bewegt dann `pHead` ein Element weiter, und gibt schließlich `pFirstEntry` zurück.

9.

```
LinkableClass* returnPrevious(LinkableClass* pTarget)
{
    // gib null zurück, wenn die Liste leer ist
    if (pHead == 0)
    {
        return 0;
    }

    // jetzt iteriere durch die Liste
    LinkableClass* pCurrent= pHead;
    while(pCurrent->pNext)
    {
        // wenn der pNext-Zeiger des aktuellen
        // Eintrags gleich pTarget ist...
        if (pCurrent->pNext == pTarget)
        {
            // ... dann gib pCurrent zurück
            return pCurrent;
        }
    }

    // wenn wir durch die gesamte Liste durch sind,
    // ohne pTarget zu finden, gib null zurück
    return 0;
}
```

Die Funktion `returnPrevious( )` **gibt den Eintrag in der Liste zurück, der unmittelbar vor** `*pTarget` **steht. Die Funktion beginnt damit, zu überprüfen, ob die Liste leer ist. Wenn sie leer ist, gibt es keinen Vorgängereintrag, und die Funktion gibt null zurück.**

`returnPrevious( )` **iteriert dann durch die Liste, wobei** `pCurrent` **auf den aktuellen Listeneintrag zeigt. In jedem Schleifendurchlauf überprüft die Funktion, ob das nächste Element von** `pCurrent` **aus gleich** `pTarget` **ist. Ist dies der Fall, gibt die Funktion** `pCurrent` **zurück. Wenn** `returnPrevious( )` **durch die gesamte Liste durchgegangen ist, ohne** `pTarget` **zu finden, gibt die Funktion null zurück.**

10.

```
LinkableClass* returnTail()
{
    // gib den letzten Eintrag der
    // Liste zurück; das ist der Eintrag,
    // dessen Nachfolger null ist
    return returnPrevious(0);
}
```

370 Anhang A

Der Eintrag, auf den null folgt, ist der letzte Eintrag in der Liste.

Zusatzaufgabe:

```
LinkableClass* removeTail()
{
    // finde den letzten Eintrag der Liste; wenn er
    // null ist, dann ist die Liste leer
    LinkableClass* pLast = returnPrevious(0);
    if (pLast == 0)
    {
        return 0;
    }

    // jetzt finde den Eintrag, der auf diesen
    // letzten Eintrag verweist
    LinkableClass* pPrevious = returnPrevious(pLast);

    // wenn pPrevious null ist ...
    if (pPrevious == 0)
    {
        // ... dann ist pLast der einzige Eintrag;
        // setze den Kopfzeiger auf null
        pHead = 0;
    }
    else
    {
        // ... sonst entferne pLast aus pPrevious
        pPrevious->pNext = 0;
    }

    // in jedem Fall gib den Zeiger pLast zurück
    return pLast;
}
```

Die Funktion `removeTail()` entfernt das letzte Element in einer verketteten Liste. Sie beginnt damit, die Adresse des letzten Elements durch einen Aufruf von `returnPrevious(0)` zu bestimmen. Sie speichert diese Adresse in `pLast`. Wenn `pLast` gleich null ist, dann ist die Liste leer und die Funktion gibt sofort null zurück.

Das letzte Element zu finden reicht nicht aus. Um das letzte Element zu entfernen, muss `removeTail()` den Eintrag vor `pLast` finden, damit die beiden getrennt werden können.

`removeTail()` bestimmt die Adresse des Eintrages von `pLast` durch einen Aufruf `returnPrevious(pLast)`. Wenn es keinen solchen Eintrag gibt, enthält die Liste nur ein Element.

`removeTail()` setzt den entsprechenden Zeiger auf null und gibt dann `pLast` zurück.

11. Ich habe für diese Lösung Visual C++ verwendet, um einen Vergleich zu `rhide` zu bekommen.

Ich beginne damit, das Programm auszuführen, einfach um zu sehen, was passiert. Wenn das Programm funktioniert, dann ist es ja gut. Das Ergebnis bei Eingabe von »diese Zeichenkette« und »DIESE ZEICHENKETTE« sieht wie folgt aus:

Dieses Programm verbindet zwei Zeichenketten:
(Diese Version stürzt ab)

Zeichenkette #1:diese Zeichenkette
Zeichenkette #2:DIESE ZEICHENKETTE

DIE
Press any key to continue

Weil ich mir sicher bin, dass das Problem in `concatString()` liegt, setzte ich einen Haltepunkt auf den Anfang der Funktion, und starte erneut. Nachdem der Haltepunkt angetroffen wurde, scheinen die beiden Zeichenketten, Quelle und Ziel, korrekt zu sein, wie sie in Abbildung 1.1 sehen können.

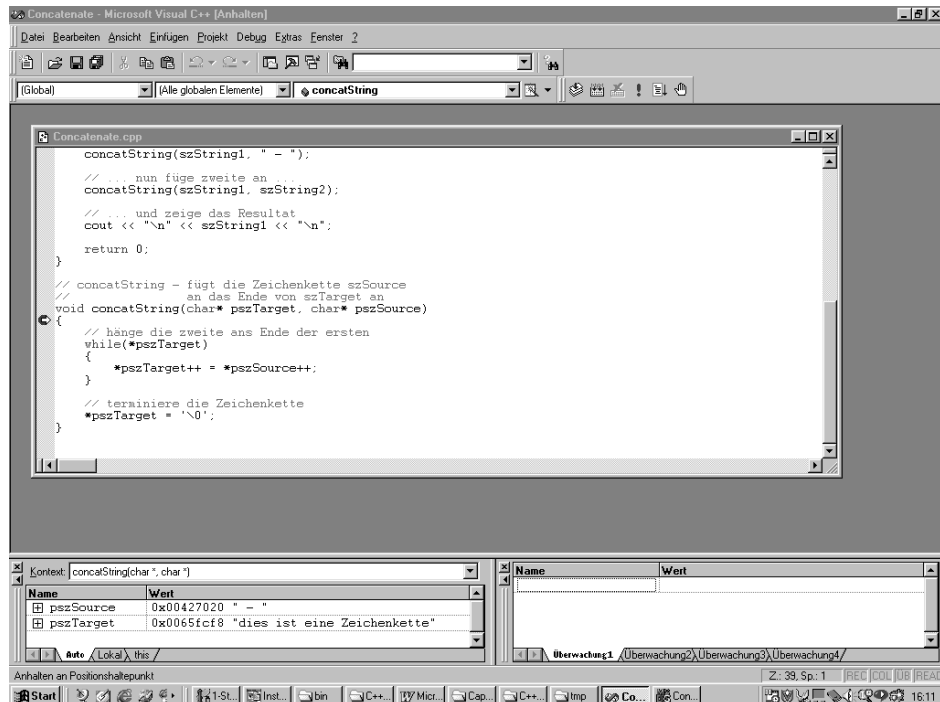


Abbildung A.1: Das Fenster zeigt die Quellzeichenkette und die Zielzeichenkette an.

372 Anhang A

Ausgehend vom Haltepunkt gehe ich in Einzelschritten vor. Zuerst scheinen die lokalen Variablen in Ordnung zu sein. Sobald jedoch die `while`-Schleife betreten wird, wird sofort klar, dass die Quellzeichenkette die Zielzeichenkette überschreibt; die Funktion `concatString( )` ist eher eine Überschreibefunktion.

Die Funktion `concatString( )` sollte damit anfangen, `pszTarget` an das Ende der Zeichenkette zu bewegen, bevor der Kopierprozess beginnt. Das Problem lässt sich nicht so einfach im Debugger beheben, und ich füge den Extracode in das Programm ein.



Tatsächlich ist es möglich, dieses Problem mit Hilfe des Debuggers zu lösen. In der Anzeige der lokalen Variablen, klicken Sie auf den Wert des Zeigers in der Spalte rechts neben `pszTarget`. Was immer dort für ein Wert steht, addieren Sie `0x12` (es gibt 18 Zeichen in »diese Zeichenkette«). Der Zeiger `pszTarget` zeigt jetzt auf das Ende der Zielzeichenkette, und das Kopieren der Zeichen kann beginnen.

Die geänderte Funktion `concatString( )` sieht wie folgt aus:

```
void concatString(char* pszTarget, char* pszSource)
{
    // bewege pszTarget ans Ende der Zielzeichenkette
    while(*pszTarget)
    {
        pszTarget++;
    }

    // füge die zweite ans Ende der ersten an
    while(*pszTarget)
    {
        *pszTarget++ = *pszSource++;
    }

    // terminate the string properly
    *pszTarget = '\0';
}
```

Ich setze den Haltepunkt auf das zweite `while`, das ist die Schleife, die das Kopieren ausführt, und starte das Programm erneut mit der gleichen Eingabe. Die lokalen Variablen sind korrekt, wie in Abbildung A.2 zu sehen ist.

Antworten auf die Wiederholungsfragen 373

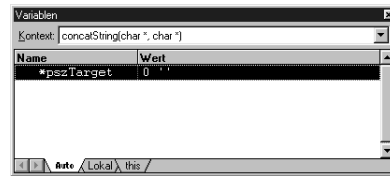


Abbildung A.2: `pszTarget` sollte auf eine Null zeigen, bevor das Kopieren durchgeführt wird.



Die Variable `pszTarget` sollten auf die Null am Ende der Zielzeichenkette zeigen, bevor das Programm die Quellzeichenkette kopieren kann.

Voller Vertrauen, versuche ich, durch die `while`-Schleife hindurchzugehen. Sobald ich jedoch »Step Over« drücke, überspringt das Programm die Schleife. Offensichtlich ist die `while`-Bedingung selbst zu Beginn nicht wahr. Nach kurzem Nachdenken stelle ich fest, dass die Bedingung falsch ist. Anstatt anzuhalten, wenn `pszTarget` gleich null ist, sollte ich anhalten, wenn `pszSource` gleich null ist. Umschreiben der `while`-Schleife, wie folgt, löst das Problem:

```
while(*pszSource)
```

Ich starte das Programm erneut mit der gleichen Eingabe, und der Debugger zeigt an, dass alles in Ordnung zu sein scheint. Und das Programm erzeugt auch die richtige Ausgabe.

Samstagabend

1. Ich finde Hemden und Hosen, die Unterklassen von Kleidungsstück sind, was Unterklasse von Bekleidung ist. In der Reihe der Bekleidungen stehen auch Schuhe und Sockenpaare. Die Sockenpaare können weiter unterteilt werden in die Paare, die zusammenpassen, und in die Paare, die nicht passen, usw.
2. Schuhe haben zumindest eine Öffnung, um den Fuß hineinzustecken. Schuhe haben eine Art Befestigungssystem, um sie am Fuß zu halten. Schuhe haben eine Sohle, um sie gegen den Untergrund zu schützen. Das ist alles, was ich über meine Schuhe sagen kann.
3. Ich habe Anzugsschuhe und Fahrradschuhe. Ich kann meine Fahrradschuhe zur Arbeit anziehen, es ist aber sehr schwer, darin zu laufen. Sie umschließen jedoch die Füße, und die Arbeit würde dadurch nicht zum Erliegen kommen.

374 Anhang A

So nebenbei – ich habe auch ein paar Kombinationsschuhe, die das Verbindungselement zur Pedale in die Sohle eingelassen haben. In diesen Schuhen zur Arbeit zu gehen wäre nicht ganz so schlecht.

- 4. Der Konstruktor eines Objektes der verketteten Liste muss sicherstellen, dass der Zeiger auf das nächste Element auf null zeigt, wenn das Objekt konstruiert wird. Es ist nicht notwendig, etwas mit den statischen Elementen zu tun.**

```
class Link
{
    static Link* pHead;
    Link* pNextLink;

    Link()
    {
        pNextLink = 0;
    }
};
```

Der Punkt ist, dass das statische Element pHead nicht im Konstruktor initialisiert werden kann, weil es sonst jedes Mal initialisiert wird, wenn ein neues Objekt erzeugt wird.

- 5. Dies ist meine Version des Destruktors und des Kopierkonstruktors:**

```
// Destruktor - vernichte Objekt und Eintrag
~LinkedList()
{
    // wenn aktuelle Objekt in der Liste ist ...
    if (pNext)
    {
        // ... dann entferne es
        removeFromList();
    }

    // wenn das Objekt Speicher belegt ...
    if (pszName)
    {
        // ... gib ihn an den Heap zurück
        delete pszName;
    }
    pszName = 0;
}

// Kopierkonstruktor - macht eine Kopie eines
// existierenden Objektes
LinkedList(LinkedList& l)
{
    // alloziere einen Block der gleichen
    // Größe vom Heap
    int nLength = strlen(l.pszName) + 1;
    this->pszName = new char[nLength];
```

Antworten auf die Wiederholungsfragen 375

```
// kopiere den Namen in diesen Block
strcpy(this->pszName, l.pszName);

// setze Zeiger auf null, da Element nicht mehr
// in verketteter Liste ist
pNext = 0;
}
```

Wenn das Objekt in einer verketteten Liste steht, dann muss der Destruktor es entfernen, bevor das Objekt wiederverwendet und der Zeiger pNext verloren ist. Wenn das Objekt zusätzlich einen Speicherbereich »besitzt«, muss dieser zurückgegeben werden. In gleicher Weise führt der Kopierkonstruktor eine tiefe Kopie aus, indem er Speicher vom Heap alloziert, in dem der Name gespeichert wird. Der Kopierkonstruktor fügt das Objekt nicht in die existierende Liste ein (obwohl er das könnte).

Sonntagmorgen**1. Die Ausgabe lautet:**

```
Advisor:Student Datenelement
Student
Advisor:Student lokal
Advisor:GraduateStudent Datenelement
GraduateStudent
Advisor: GraduateStudent lokal
```

Lassen Sie uns jede Zeile der Ausgabe ansehen:

Die Kontrolle geht an den Konstruktor von GraduateStudent und von dort an den Konstruktor der Basisklasse Student über.

Das Datenelement Student::adv wird konstruiert.

Die Kontrolle geht an den Body des Konstruktors Student über.

Ein lokales Objekt aus der Klasse Advisor wird vom Heap erzeugt.

Die Kontrolle kehrt zum Konstruktor GraduateStudent zurück, der das Datenelement GraduateStudent::adv erzeugt.

Die Kontrolle betritt den Konstruktor GraduateStudent, der ein Advisor-Objekt vom Heap alloziert.

376 Anhang A**2.**

```
// PassProblem - nutze Polymorphie, um zu
//                entscheiden, ob ein Student
//                besteht oder durchfällt
#include <stdio.h>
#include <iostream.h>

class Student
{
public:
    virtual int pass(double dGrade)
    {
        // wenn es zum Bestehen reicht ...
        if (dGrade > 1.5)
        {
            // ... bestanden
            return 1;
        }
        // ... sonst durchgefallen
        return 0;
    }
};

class GraduateStudent : public Student
{
public:
    virtual int pass(double dGrade)
    {
        if (dGrade > 2.5)
        {
            return 1;
        }
        return 0;
    }
};
```

3. Meine Version der Klasse Checking ist wie folgt:

```
// Checking - gleich mit Sparkonten, außer dass
//                Abhebungen einen Dollar kosten
class Checking : public CashAccount
{
public:
    Checking(unsigned nAccNo,
              float fInitialBalance = 0.0F)
        : CashAccount(nAccNo, fInitialBalance)
    {
    }

    // ein Girokonto weiß, wie Abhebungen
    // durchgeführt werden (machen Sie sich keine
    // Gedanken zu Überziehungen)
    virtual void withdrawal(float fAmount)
    {
    }
};
```

```

        // Abhebung ausführen
        fBalance -= fAmount;

        // Gebühr erheben
        fBalance -= 1;
    }
};

```



Das gesamte Programm ist auf der beiliegenden CD-ROM enthalten unter dem Namen AbstractProblem.

4. Das ist meine Lösung des Problems:

```

// MultipleVirtual - erzeuge Klasse PrinterCopier
//                      durch Erben von Printer und
//                      Copier
#include <stdio.h>
#include <iostream.h>
class ElectronicEquipment
{
public:
    ElectronicEquipment(int nVoltage)
    {
    }
    int nVoltage;
};
class Printer : virtual public ElectronicEquipment
{
public:
    Printer() : ElectronicEquipment()
    {
    }
    void print()
    {
        cout << »drucken\n«;
    }
};
class Copier : virtual public ElectronicEquipment
{
public:
    Copier() : ElectronicEquipment()
    {
    }
    void copy()
    {
        cout << »kopieren\n«;
    }
};
class PrinterCopier : public Printer, public Copier
{

```

378 Anhang A

```

public:
    PrinterCopier(int nVoltage) : Copier(), Printer()
    {
        this->nVoltage = nVoltage;
    }
};

int main(int nArgs, char* pszArgs)
{
    PrinterCopier ss(220);

    // erst drucken
    ss.print();

    // dann kopieren
    ss.copy();

    // Spannung ausgeben
    cout << »Spannung = » << ss.nVoltage << »\n«;

    return 0;
}

```

Sonntagnachmittag**1. Meine Version der beiden Funktionen sieht wie folgt aus:**

```

MyClass(MyClass& mc)
{
    nValue = mc.nValue;
    resource.open(nValue);
}
MyClass& operator=(MyClass& s)
{
    resource.close();

    nValue = s.nValue;
    resource.open(nValue);
}

```

Der Kopierkonstruktor öffnet das aktuelle Objekt mit dem Wert des Quellobjektes.

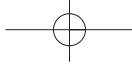
Der Zuweisungsoperator schließt zuerst die aktuelle Ressource, weil sie mit einem anderen Wert geöffnet wurde. Sie öffnet dann die Ressource wieder mit dem neuen Wert, der übergeben wurde.

2. Meine Klasse und Inserter sind wie folgt:

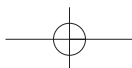
```
// Student
class Student
{
    friend ostream& operator<<(ostream& out, Student& d);
public:
    Student(char* pszFName, char* pszLName, int nSSNum)
    {
        strncpy(szFName, pszFName, 20);
        strncpy(szLName, pszLName, 20);
        this->nSSNum = nSSNum;
    }

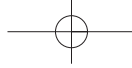
protected:
    char szLName[20];
    char szFName[20];
    int nSSNum;
};

// Inserter - Zeichenkettenbeschreibung
ostream& operator<<(ostream& out, Student& s)
{
    out << s.szLName
        << », »
        << s.szFName
        << »(»
        << s.nSSNum
        << »)«;
    return out;
}
```



vakat





Ergänzende Probleme



Dieser Anhang enthält ergänzende Probleme für verschiedene Teile des Buches, die Ihnen zusätzliche Erfahrung bei der Arbeit mit C++ bringen und ihre neuen Fähigkeiten festigen sollen. Die Probleme finden Sie in Kapiteln, die den einzelnen Buchteilen zugeordnet sind, jeweils gefolgt von einem Abschnitt mit Lösungen für die Probleme.

1.1 Probleme

1.1.1 Samstagmorgen

1. Welche der folgenden Anweisungen sollte

a. keine Meldungen erzeugen?

b. Warnungen erzeugen?

c. Fehler erzeugen?

```
1. int n1, n2, n3;  
2. float f1, f2 = 1;  
3. double d1, d2;  
4. n1 = 1; n2 = 2; n3 = 3; f2 = 1;  
5. d1 = n1 * 2.3;  
6. n2 = n1 * 2.3;  
7. n2 = d1 * 1;  
8. n3 = 100; n3 = n3 * 1000000000;  
9. f1 = 200 * f2;
```

2. Gegeben, dass `n1` gleich 10 ist, werten sie das Folgende aus:

a. `n1 / 3`

b. `n1 % 3`

c. `n1++`

d. `++n1`

e. `n1 %= 3`

f. `n1 -= n1`

g. `n1 = -10; n1 = +n1; was ist n1?`

382 Anhang B**3. Was ist der Unterschied der beiden folgenden for-Schleifen:**

```
for(int i = 0; i < 10; i++)
{
    // ...
}
for (int i = 0; i < 10; ++i)
{
    // ...
}
```

4. Schreiben Sie die Funktion `int cube(int)`, die $n * n * n$ für n berechnet.**5. Beschreiben Sie genau, was passiert, wenn die Funktion aus Problem 4 wie folgt verwendet wird:**

```
int n = cube(3.0);
```

6. Das folgende Programm zeigt eine sehr wenig durchdachte Funktion, die die Integerquadratwurzel einer Zahl berechnet, durch Vergleiche mit dem Quadrat eines Zählers. Mit anderen Worten, es ist $4 * 4 = 16$, woraus folgt, dass 4 die Quadratwurzel von 16 ist. Diese Funktion berechnet 3 als Quadratwurzel von 15, weil $3 * 3$ kleiner ist als 15, aber $4 * 4$ größer ist als 15.

Das Programm erzeugt jedoch unerwartete Ergebnisse. Beheben Sie den Fehler!

```
// Lesson - diese Funktion zeigt eine wenig
//          durchdachte, aber effektive Methode
//          zur Berechnung der Quadratwurzel, wobei
//          nur Integers verwendet werden.
//          Diese Version funktioniert nicht.
#include <stdio.h>
#include <iostream.h>

// squareRoot - gegeben eine Zahl n, gib ihre
//               Quadratwurzel zurück, indem
//               nRoot * nRoot für aufsteigende Werte
//               nRoot berechnet wird, bis das Quadrat
//               größer ist als n
void squareRoot(int n)
{
    // starte bei 1
    int nRoot = 1;

    // Endlosschleife
    for(;;)
    {
        // überprüfe Quadrat des aktuellen Werts
        // (und inkrementiere zum nächsten)
        if ((nRoot++ * nRoot) > n)
```

```

        {
            // so nah wie möglich, gib diesen
            // Wert zurück
            return nRoot;
        }

    // hier sollten wir nicht hinkommen
    return 0;
}

// teste die Funktion squareRoot() mit einem
// einzelnen Wert (ein ausführlicherer Test
// wäre angebracht, aber schon dieser Test
// funktioniert nicht)
int main(int argc, char* pszArgs[])
{
    cout << »Dieses Programm funktioniert nicht!\n«;

    cout << »Quadratwurzel von »
        << 16
        << » ist »
        << squareRoot(16)
        << »\n«;
    return 0;
}

```

Hinweis: Achten Sie auf die Features Autoinkrement und Postinkrement.

1.1.2 Samstagnachmittag

1. Die C++-Bibliotheksfunktion `strchr( )` gibt den Index eines Zeichens in einer Zeichenkette zurück. So gibt z.B. `strchr(»abcdef«, 'c')` den Index 2 zurück. `strchr( )` gibt -1 zurück, wenn das Zeichen in der Zeichenkette nicht vorkommt.

Schreiben Sie eine Funktion `myStrchr( )` die das Gleiche tut wie `strchr( )`. Wenn Sie denken, dass Ihre Funktion fertig ist, testen Sie sie mit dem Folgenden:

```

// MyStrchr - suche ein gegebenes Zeichen in einer
//           Zeichenkette. Gib den Index des
//           zurück
#include <stdio.h>
#include <iostream.h>

// myStrchr - gib den Index eines Zeichens in einer
//           Zeichenkette zurück; gib -1 zurück,
//           wenn das Zeichen nicht gefunden wird
int myStrchr(char target[], char testChar);

// teste die Funktion myStrchr mit verschiedenen
// Kombinationen von Zeichenketten

```

384 Anhang B

```
void testFn(char szString[], char cTestChar)
{
    cout << »Der Offset von »
        << cTestChar
        << » in »
        << szString
        << » ist »
        << myStrchr(szString, cTestChar)
        << »\n«;
}

int main(int nArgs, char* pszArgs[])
{
    testFn(»abcdefg«, 'c');
    testFn(»abcdefg«, 'a');
    testFn(»abcdefg«, 'g');
    testFn(»abcdefc«, 'c');
    testFn(»abcdefg«, 'x');

    return 0;
}
```

Hinweis: Achten Sie darauf, dass Sie nicht aus der Zeichenkette herauslaufen, wenn das Zeichen nicht gefunden wird.

2. Obwohl das Folgende schlecht programmiert ist, verursacht es doch keine Probleme. Erklären Sie warum nicht.

```
void fn(void)
{
    double d;
    int* pnVar = (int*)&d;
    *pnVar = 10;
}
```

3. Schreiben Sie eine Zeigerversion der folgenden Funktion `displayString( )`. Nehmen Sie an, dass das eine Null-terminierte Zeichenkette ist (übergeben Sie nicht die Länge als Argument an die Funktion).

```
void displayCharArray(char sArray[], int nSize)
{
    for(int i = 0; i < nSize; i++)
    {
        cout << sArray[i];
    }
}
```

4. Kompilieren Sie das folgende Programm, und führen Sie es aus. Erklären Sie die Ergebnisse:

```

#include <stdio.h>
#include <iostream.h>
// MyClass - Testklasse ohne Bedeutung
class MyClass
{
public:
    int n1;
    int n2;
};

int main(int nArg, char* nArgs[])
{
    MyClass* pmc;
    cout << »n1 = » << pmc->n1
        << »;n2 = » << pmc->n2
        << »\n«;
    return 0;
}

```

1.1.3 Sonntagmorgen

1. Schreiben Sie eine Klasse `Car`, die von der Klasse `Vehicle` erbt, und die einen Motor besitzt. Die Anzahl der Reifen wird im Konstruktor der Klasse `Vehicle` angegeben, und die Anzahl der Zylinder im Konstruktor von `Motor`. Beide Werte werden an den Konstruktor der Klasse `Car` übergeben.

1.2 Antworten**1.2.1 Samstagmorgen**

1. 1. Kein Problem.
2. Keine Warnung; sie können initialisieren, wenn sie wollen.
3. Alles klar.
4. Kein Problem.
5. Kein Problem; `n1` wird automatisch in ein `double` verwandelt, um die Multiplikation ausführen zu können. Die meisten Compiler werden diese Konvertierung nicht kommentieren.
6. `n1 * 2.3` ist ein `double`. Die Zuweisung an eine `int`-Variable führt zu einer Warnung wegen Demotion.

386 Anhang B

7. Ähnlich wie 6. Obwohl 1 ein `int` ist, ist `d1` ein `double`. Das Ergebnis ist ein `double`, das nach `int` konvertiert wird (Demotion).
8. Keine Warnung, aber es funktioniert nicht. Das Ergebnis liegt außerhalb des Bereiches von `int`.
9. Das sollte eine Warnung erzeugen (Visual C++ tut es auch, GNU C++ tut es nicht). Das Ergebnis einer Multiplikation von `int` und `float` ist `double` (alle Berechnungen werden in `double` ausgeführt). Das Ergebnis muss nach `float` konvertiert werden (Demotion).
2. a. 3 – Rundungsfehler, die in Sitzung 5 erklärt wurden, konvertieren das erwartete Ergebnis 3.3 nach 3.
b. 1 – Der Teiler, der 10 / 3 am nächsten ist, ist 3. 10 – (3 * 3) ist 1, der Rest der Division.
c. 10 – `n1++` gibt den Wert von `n1` zurück, bevor `n1` inkrementiert wird. Nach der Auswertung des Ausdrucks ist `n1 = 11`.
d. 11 – `++n1` inkrementiert `n1`, bevor der Wert zurückgegeben wird.
e. 1 – Das ist das Gleiche wie `n1 = n1 % 3`
f. 0 – Das ist das Gleiche wie `n1 = n1 - n1`, aber `n1 - n1` ist immer Null.
g. -10 – Der unäre Plusoperator (+) hat keinen Effekt. Insbesondere ändert er nicht das Vorzeichen einer negativen Zahl.
3. Kein Unterschied. Die Inkrementklausel einer `if`-Anweisung wird als separater Ausdruck behandelt. Der Wert von `i` ist nach einem Präinkrement und nach einem Postinkrement gleich (nur der Wert des Ausdrucks ist verschieden).
4.

```
int cube(int n)
{
    return n * n * n;
}
```
5. Der `double`-Wert 3.0 wird demotiert in den `int`-Wert 3, und das Ergebnis wird als Integerwert 9 zurückgegeben.
6. Fehler #1: Das Programm kann nicht kompiliert werden, weil die Funktion `squareRoot( )` so deklariert wurde, dass sie `void` als Rückgabewert hat. Ändern Sie den Rückgabewert auf `int` und erzeugen Sie das Programm erneut.
Fehler #2: Das Programm behauptet nun, dass die Quadratwurzel von 16 gleich 6 ist. Um das Problem zu verstehen, splitte ich die zusammengesetzte `if`-Bedingung, damit ich den Ergebniswert jeweils ausgeben kann:

```
// Endlosschleife
for(;;)
{
    // überprüfe Quadrat des aktuellen Wertes
    // (und inkrementiere zum nächsten)
    int nTest = nRoot++ * nRoot;
    cout << »Testwurzel ist »
        << nRoot
        << » Quadrat ist »
        << nTest
        << »\n«;
    if (nTest > n)
    {
        // so nah wie möglich, gib diesen
        // Wert zurück
        return nRoot;
    }
}
```

Die Ausgabe des Programms sieht wie folgt aus:

```
Dieses Programm funktioniert nicht!
Testwurzel ist 2 Quadrat ist 1
Testwurzel ist 3 Quadrat ist 4
Testwurzel ist 4 Quadrat ist 9
Testwurzel ist 5 Quadrat ist 16
Testwurzel ist 6 Quadrat ist 25
Quadratwurzel von 16 ist 6
```

Die Ausgabe des Programms ist überhaupt nicht korrekt. Eine genaue Untersuchung zeigt jedoch, dass die linke Seite um eins verschoben ist. Das Quadrat von 3 ist 9, aber der angezeigte Wert von `nRoot` ist 4. Das Quadrat von 4 ist 16, aber der angezeigte Wert von `nRoot` ist 5. Durch die Inkrementierung von `nRoot` im Ausdruck, ist `nRoot` um eins größer als das `nRoot`, das in der Berechnung verwendet wird. Somit muss `nRoot` nach der `if`-Anweisung inkrementiert werden.

Die neue Funktion `squareRoot( )` sieht so aus:

```
// Endlosschleife
for(;;)
{
    // überprüfe Quadrat des aktuellen Wertes
    int nTest = nRoot * nRoot;
    cout << »Testwurzel ist »
        << nRoot
        << » Quadrat ist »
        << nTest
        << »\n«;
    if (nTest > n)
    {
        // so nah wie möglich, gib diesen
```

388 Anhang B

```

        // Wert zurück
        return nRoot;
    }
    // versuche nächsten Wert für nRoot
    nRoot++;
}

```

Das Autoinkrement wurde hinter den Test platziert (das Autoinkrement war die ganze Zeit verdächtig). Die Ausgabe des neuen, verbesserten Programms sieht wie folgt aus:

```

Dieses Programm funktioniert nicht!
Testwurzel ist 1 Quadrat ist 1
Testwurzel ist 2 Quadrat ist 4
Testwurzel ist 3 Quadrat ist 9
Testwurzel ist 4 Quadrat ist 16
Testwurzel ist 5 Quadrat ist 25
Quadratwurzel von 16 ist 5

```

Das Quadrat wirkt korrekt berechnet, aber aus irgendeinem Grund stoppt die Funktion nicht, wenn `nRoot` gleich 4 ist. Es gilt aber doch $4 * 4 == 16$. Das ist genau das Problem – der Test überprüft `nTest > n`, wo er eigentlich `nTest >= n` überprüfen sollte. Das korrigierte Programm erzeugt die erwartete Ausgabe:

```

Dieses Programm funktioniert!
Testwurzel ist 1 Quadrat ist 1
Testwurzel ist 2 Quadrat ist 4
Testwurzel ist 3 Quadrat ist 9
Testwurzel ist 4 Quadrat ist 16
Quadratwurzel von 16 ist 4

```

Nachdem ich verschiedene Werte getestet habe, bin ich davon überzeugt, dass das Programm korrekt ist, und ich entferne die Ausgabeanweisungen.

1.2.2 Samstagnachmittag

1. Die folgende Funktion `myStrchr( )` ist meine Lösung des Problems:

```

// myStrchr - gib den Index eines Zeichens in einer
//             Zeichenkette zurück; gib -1 zurück,
//             wenn das Zeichen nicht gefunden wird
int myStrchr(char target[], char testChar)
{
    // Schleife über alle Zeichen der Zeichenkette;
    // spätestens am Ende der Zeichenkette stoppen
    int index = 0;
    while(target[index])
    {
        // wenn das aktuelle Zeichen der

```



```
// Zeichenkette gleich dem gesuchten
// Zeichen ist ...
if (target[index] == testChar)
{
    // ... dann stoppen
    break;
}

// gehe zum nächsten Zeichen
index++;
}

// wenn wir am Ende der Zeichenkette ankommen,
// ohne das Zeichen zu finden, ...
if (target[index] == '\0')
{
    // ... gib -1 und nicht die Länge
    // des Arrays zurück
    index = -1;
}

// gib den berechneten Index zurück
return index;
}
```

Die Funktion `myStrchr( )` beginnt damit, durch die Zeichenkette `target` zu iterieren, wobei gestoppt wird, wenn das aktuelle Zeichen `target[index]` gleich 0 ist, was bedeutet, dass die Funktion das Ende der Zeichenkette erreicht hat. Dieser Test stellt sicher, dass die Funktion nicht zu weit geht, wenn das Zeichen nicht gefunden wird.

Innerhalb dieser Schleife vergleicht die Funktion das aktuelle Zeichen mit dem gesuchten Zeichen `testChar`. Wenn das Zeichen gefunden wird, verläßt die Funktion die Schleife vorzeitig.

Wenn die Schleife verlassen worden ist, wurde entweder das Ende der Zeichenkette angetroffen oder das zu suchende Zeichen wurde gefunden. Wenn das Ende der Zeichenkette der Grund ist, dann ist `target[index]` gleich 0, bzw. `'\0'` um das Zeichenäquivalent zu verwenden. In diesem Fall wird `index` auf -1 gesetzt.

Der Wert von `index` wird an den Aufrufenden zurückgegeben.

Das Ergebnis der Programmausführung sieht wie folgt aus:

```
Der Offset von c in abcdefg ist 2
Der Offset von a in abcdefg ist 0
Der Offset von g in abcdefg ist 6
Der Offset von c in abcdefg ist 2
Der Offset von x in abcdefg ist -1
Press any key to continue
```

390 Anhang B

Das obige Programm kann dadurch vereinfacht werden, dass die Funktion verlassen wird, wenn das Zeichen gefunden wird:

```
int myStrchr(char target[], char testChar)
{
    // Schleife über alle Zeichen der Zeichenkette;
    // spätestens am Ende der Zeichenkette stoppen
    int index = 0;
    while(target[index])
    {
        // wenn das aktuelle Zeichen der
        // Zeichenkette gleich dem gesuchten
        // Zeichen ist ...
        if (target[index] == testChar)
        {
            // ... dann gib Index zurück
            return index;
        }

        // gehe zum nächsten Zeichen
        index++;
    }

    // wenn wir die Schleife durchlaufen haben,
    // sind wir am Ende der Zeichenkette angekommen,
    // ohne das Zeichen zu finden
    return -1;
}
```

Wenn das Zeichen gefunden wird, wird es sofort zurückgegeben. Wenn die Kontrolle die Schleife anders verläßt, kann das nur bedeuten, dass das Ende der Zeichenkette gefunden wurde, ohne das Zeichen zu finden.

Ich persönlich bevorzuge diesen Stil. Es gibt aber Organisationen, bei denen mehrere Rückgaben pro Funktion verboten sind.

2. **Ein double belegt 8 Bytes, ein int belegt 4 Bytes. Es ist möglich, dass C++ 4 von den 8 Bytes verwendet, um den int-Wert 10 zu speichern, ohne die anderen 4 Bytes zu verwenden. Das verursacht keinen Fehler; sie sollten jedoch nicht davon ausgehen, dass sich Ihr Compiler so verhält.**

3.

```
void displayString(char* pszString)
{
    while(*pszString)
    {
        cout << *pszString;
        pszString++;
    }
}
```

4. Die Ausgabe von `pmc->n1` und `pmc->n2` sind vollkommen falsch, weil der Zeiger `pmc` nicht initialisiert wurde, auf etwas zu zeigen. In der Tat kann das Programm abstürzen, ohne überhaupt eine Ausgabe zu erzeugen, wegen dieses nicht initialisierten Zeigers.

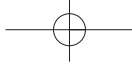
1.2.3 Sonntagmorgen

```
1. class Vehicle
{
    public:
        Vehicle(int nWheels)
        {
        }
};

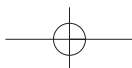
class Motor
{
    public:
        Motor(int nCylinders)
        {
        }
};

class Car : public Vehicle
{
    public:
        Car(int nCylinders, int nWheels)
            : Vehicle(nWheels), motor(nCylinders)
        {
        }

        Motor motor;
};
```



vakat



Was ist auf der CD-ROM



Die CD-ROM, die diesem Buch beiliegt, enthält Material, das Ihnen bei dem Durcharbeiten der Sitzungen und beim Erlernen von C++ innerhalb eines Wochenendes hilft:

- Installationsdateien für den GNU C++-Compiler
- Alle Programme in diesem Buch

1.1 GNU C++

Die Installationsdateien für das GNU C++, die Sie auf der beiliegenden CD-ROM finden, stammen von der Delorie Website, die in Sitzung 3 erwähnt wurde. Wir stellen die kompletten Dateien für Windows 95, 98, und NT/2000 bereit.

Um GNU C++ mit diesen Dateien zu installieren, führen Sie die folgenden Schritte durch:

1. Erzeugen Sie ein Verzeichnis \DJGPP
2. Kopieren Sie alle Zip-Dateien in dem Ordner auf der CD-ROM, der zu Ihrem Betriebssystem passt, in das Verzeichnis DJGPP.
3. Entpacken Sie die Zip-Dateien.
4. Fügen Sie die folgenden Kommandos in Ihre Datei AUTOEXEC.BAT ein:

```
set PATH=C:\DJGPP\BIN;%PATH%
set DJGPP=C:\DJGPP\DJGPP.ENV
```



Es wurde angenommen, dass das Verzeichnis DJGPP direkt unter C:\ steht. Wenn Sie Ihr Verzeichnis DJGPP an einer anderen Stelle platziert haben, müssen Sie den Pfad in den obigen Kommandos entsprechend anpassen.

5. Starten Sie Ihr System neu, um die Installation abzuschließen.

Das Verzeichnis \BIN, das beim Entpacken der Dateien erzeugt wurde, enthält die eigentlichen Programme der GNU-Umgebung. Die Datei DJGPP.ENV setzt eine Reihe von Optionen, um die GNU C++-Umgebung zu beschreiben.



Bevor Sie GNU C++ verwenden, stellen Sie sicher, dass in DJGPP.ENV lange Dateinamen angeschaltet sind. Diese Option ausgeschaltet zu haben ist der häufigste Fehler bei der Installation von GNU C++.

394 **Anhang C**

Öffnen Sie die Datei DJGPP.ENV mit einem Texteditor, z.B. mit Microsoft WordPad. Erschrecken Sie nicht, wenn Sie nur eine lange Zeichenkette sehen, die von kleinen schwarzen Kästchen unterbrochen ist. Unix verwendet ein anderes Zeichen für Zeilenumbrüche als Windows. Suchen Sie nach der Phrase »LFN=y« oder »LFN=Y« (Groß- und Kleinschreibung spielt also keine Rolle). Wenn Sie stattdessen »LFN=n« finden (oder »LFN« überhaupt nicht vorkommt), ändern Sie das »n« in »y«. Speichern Sie die Datei. (Stellen Sie sicher, dass Sie die Datei als Textdatei speichern und nicht in einem anderen Format, z.B. als Word .DOC-Datei.)

1.2 Beispielprogramme

Das Verzeichnis \Programs enthält den Sourcecode aller Programme in diesem Buch. Ich empfehle Ihnen, den gesamten Ordner und alle Unterverzeichnisse auf Ihre Festplatte zu kopieren; sie können aber auch einzelne Dateien kopieren, wenn Sie das möchten.

Die Programme sind nach Sitzungen aufgeteilt. Der Ordner einer Sitzung enthält die .CPP-Dateien und die ausführbaren Programme; letztere sollen es dem Leser bequemer machen. Sie können die Programme direkt von der CD-ROM aus ausführen. Die ausführbaren .EXE-Dateien wurden mit GNU C++ erzeugt. Die mit Visual C++ erzeugten .EXE-Dateien befinden sich in einem Unterverzeichnis DEBUG.

Sitzung 2 enthält Anweisungen, wie Programme mit Visual C++ erzeugt werden. Sitzung 3 erklärt das Gleiche für GNU C++.

Hinweis: Zwei Quelldateien wurden abweichend von diesem Buch modifiziert:

1. Die Funktion `ltoa( )` wurde durch einen Aufruf der funktional äquivalenten Funktion `itoa( )` in `ToStringWStream.cpp` in Sitzung 29 ersetzt.
2. GNU C++ verwendet den vollständigen Namen von `strstream.h`, während Visual C++ das 8.3-Format `strstrea.h` verwendet. Die Referenz auf diese Include-Datei muss in `ToStringWStream.cpp` entsprechend angepasst werden. Der Sourcecode verwendet ein `#ifdef`, um zu entscheiden, welche Include-Datei eingebunden werden soll. Der Sourcecode in diesem Buch weist mit einem Kommentar auf dieses Problem hin.

Index

#define Direktive 299 - 302
#defines __FILE__ 355
#defines __LINE__ 355
#else Zweig 302
#if Anweisung 302
#if Direktive 302, 303
#ifdef Direktive 303
#ifdef Direktive, Debug-Code 304, 305
#ifdef Direktive, Einschlusskontrolle 303, 304
#ifndef Direktive, Einschlusskontrolle 303, 304
#include <iostream.h> Direktive 295
#include <stdio.h> Direktive 295
#include 16, 17
#include Direktive 200, 290, 291, 298, 299
#include Kommando, Namen, Konstruktion 298, 299
(Doppelkreuz) 298, 299
! (einfacher logischer Operator) 60
!= (einfacher logischer Operator) 60
'_' (einfache Anführungszeichen und Unterstrich) 34
' ' (einfache Hochkommata) 113
" " (doppelte Hochkommata) 113, 298, 299
% (Prozentzeichen) 117
% mathematischer Operator 53
%= mathematischer Operator 53
& (AND bitweiser Operator) 64
& (unärer) mathematischer Operator 53
& (unärer) Operator 129
& (Und-Zeichen) 125
&& (einfacher logischer Operator) 60
() (Klammern) 81, 82, 157
, (Komma) 87
; (Semikolon) 12, 24, 32
\ (Backslash) 49
*= mathematischer Operator 53
* mathematischer Operator 53
* (unärer) 53
*/ 31
/ mathematischer Operator 53
// (doppelter Slash) 31
^ (XOR bitweiser Operator) 64
{ } (geschweifte Klammern) 70
~ (NOT bitweiser Operator) 65
~ (Tilde) 213
+ (unärer) mathematischer Operator 53
+ mathematischer Operator 53
++ (unärer) mathematischer Operator 53
+= mathematischer Operator 53
- (unärer) mathematischer Operator 53
— (unärer) mathematischer Operator 53
- mathematischer Operator 53
-= mathematischer Operator 53
\ BIN Verzeichnis 19
FILE 355
LINE 355
| (OR bitweiser Operator) 64
| (Umleitungssymbol) 153

396 *Index*

|| (einfacher logischer Operator) 60
 < (einfacher logischer Operator) 60
 < (Umleitungssymbol) 153
 <= (einfacher logischer Operator) 60
 <> (spitze Klammern) 295, 298, 299
 = (mathematischer Operator) 53
 == (einfacher logischer Operator) 60
 > (einfacher logischer Operator) 60
 > (Umleitungssymbol) 153
 >= (einfacher logischer Operator) 60
 0xff Exit-Code rhide 96

A

Abschneiden von Integer 44, 46
 abstrakte Klassen 267
 Basisklassen 268
 deklarieren 267 - 269
 Klasse konkret machen 269 - 271
 konkrete Elementfunktion 269
 rein virtuelle Funktion 272 - 274
 Syntax rein virtuelle Funktion 267, 268
 Unterklassen 271 - 273
 abstrakte Objekte an Funktionen übergeben 272, 273
 Abstraktion, objektorientierte Programmierung 187, 188
 Absturz, Programme, Funktion concatString() 173
 Abwickeln des Stacks (Ausnahmebehandlung), Code 352, 353
 Account Klasse 266
 konkrete Klasse, aus abstrakter Klasse erzeugen 269, 271
 Syntax rein virtuelle Funktion 267, 268
 Account Programm-Code 267 - 270
 acht-Zeichen-Integer 342
 Add Watch Kommando 169
 addHead() Funktion 217, 218
 Additions-Operatoren 53
 addTail() Funktion 162, 163, 167

Adressen
 nInt, speichern in plnt 131
 Speicher 128, 129
 aktive Klassen 192
 Bereichsauflösungsoperator 196
 Dateien einbinden 199, 200
 definieren 195, 197
 Definitionen definieren 198
 Deklarationen definieren 198
 Include-Dateien 199, 200
 aktive Klassen, Funktionen
 Elemente aufrufen 200 - 203
 Elemente definieren 197, 198
 Elemente überladen 203 - 205
 inline 197, 198
 Namen für Elemente 196, 197
 Nichtelement 195
 aktuelles Verzeichnis 295
 Algorithmen
 definieren 4
 Division-vor-Summe 44
 menschliches Programm 4
 Prozessoren 4
 Alt+F5 Tastaturkürzel 26
 AmbiguousBinding Programm-Code 254, 255
 AmbiguousInheritance Programm-Code 280 - 282
 AND bitweiser Operator (&) 63, 64
 AND Operator (&&) 61, 62
 Anführungszeichen
 " " (doppelte Anführungszeichen) 133, 298, 299
 ' ' (einfache Anführungszeichen) 113
 '_ ' (einfache Anführungszeichen und Unterstrich) 34
 Ansehen von Variablen 175 - 180
 Anweisungen
 #if 302
 ; (Semicolon) 32
 Anzahl von, verglichen mit Anzahl
 Instruktionen 147
 ausführbare, definiert 172
 Ausgabe 92 - 95
 break, switch Kommando 79
 C++-Programme 32
 definiert 32
 Eingabe/Ausgabe 33

- if, demonstriert 69 - 71
- include 31
- inp> Extraktor 337, 338
- Leerraum 32
- return, void-Funktion 85
- Schleifen 71
- switch 79
- Verzweigung 69 - 71
- WRITE 92, 93
- Anwendung, verkettete Liste, Debuggen 176
- Anwendungs-Code aufteilen 291 - 293
- Arbeitsbereich
 - anlegen 11
 - Conversion.pdw Arbeitsbereich 11
 - Kommando (View Menü) 294
 - schließen, Kommando (Menü Datei) 10
- Argumente
 - , (Komma) 87
 - an Funktionen als Zeigervariablen übergeben 133 - 135
 - binäre Operatoren 55, 56
 - binäre Operatoren, Werte verändern 318
 - Default-Werte 223
 - durch Projekteinstellungen übergeben 154
 - Funktionen 84 - 87
 - in Konstruktoren definieren 221, 222
 - in rhide 154
 - Konstruktoren 220 - 223
 - main()-Funktion 152 - 154
 - out, ostream-Objekt 341
 - Programme 150 - 154
 - unärer Operator 55 - 58
 - unärer Operator++() 317, 318
- arithmetische Operatoren 53, 54
- Arrays
 - ArrayDemo Programm-Code 108 - 110
 - Ausgabe-Funktionen 117
 - CharDisplay Programm-Code 111, 112
 - Datenstruktur 159, 160
 - definiert 106
 - Deklarationen 106
 - DisplayString-Programm 112, 113
 - Manipulation von Zeichenketten und Zeiger 145, 146
 - Matrix 111, 112
 - nArray zugreifen 108
 - nInputValue 110
 - parallele 120 - 122
 - wide characters 117
 - zählen 106 - 108
 - Zeichen 111 - 113, 145 - 147
 - Zeichenketten 113, 151, 152
 - Zeichenketten, manipulieren 114 - 117
 - Zeiger 140, 150
 - Zeiger, Differenz zwischen 148 - 150
 - Zeiger-Offsets 142
- Arrays zählen 106 - 108
- auflösen (nicht eindeutig) 204
- aufrufen
 - Elementfunktionen 200 - 203
 - sumSequence()-Funktion 83
- aufteilen
 - Anwendungs-Code 291 - 293
 - Programm 288 - 290
- Ausdrücke
 - C++ Programme 33, 34
 - definiert 34
 - gemischter Modus 50
 - mathematische Operatoren 54
- ausführbare Anweisung, definiert 172
- ausführen
 - Conversion.exe 14, 15
 - GNU C++ Programme 26
 - Programme 14, 15
- Ausgabe
 - Anweisungen 92 - 95, 209
 - C++-Programme 16
 - ConstructElements Programm mit Destruktor 213
 - Funktionen 117
 - Insertter 343, 344
 - Konstruktoren 225
- Ausgabefenster 10
- Ausnahme-Klasse, InvalidArgumentOutOfRangeException::display() 356
- Ausnahmen 350 - 352
- Ausnahmen behandeln
 - Ausnahme-basierte Stream-Ausgabe 335
 - Ausnahme-Klasse
 - InvalidArgumentOutOfRangeException::display() 356
 - catch-Phrasen 354 - 357
 - Fehler behandeln 348, 349

398 Index

Fehlerrückgaben 349, 350
 fn(...) Deklaration 354
 Funktionen, fn(...) Deklaration 354
 InvalidArgumentException::display(), Ausnahme-Klasse 356
 Objekte auslösen 354 - 356
 Stack abwickeln, Code 352, 353
 try Schlüsselwort 351
 auto Variable 90

B

\ Backslash () 49
 Basisklassen
 abstrakte Klassen 268
 Destruktor 248
 konstruieren 248
 Bedingungen überprüfen 71
 Begrenzungen
 Gleitkomma-Variablen 46
 int Variablentyp 42 - 45
 Beispiel-Code, Funktionen 81, 83
 Benutzer-Interfaces rhide 20 - 28
 Bereich, begrenzt für int Variable 44
 Bereichsauflösungsoperator 196
 Bibliotheken, MSDN Bibliothek 16
 BIN Verzeichnis 19
 binäre Operatoren 55, 56, 318
 binäre Zahlen 62, 63
 Binden zur Compile-Zeit 253
 Bindung
 Bits, definiert 62
 Compile-Zeit 253
 EarlyBinding, Programm-Code 253
 frühe 253
 späte 255
 bitweise logische Operatoren 58, 63, 64 - 67
 Code zum Testen 65, 66
 Einzelbit-Operatoren 63
 Masken 67
 Sinn 64, 66, 67

BranchDemo Programm-Code 69, 70, 71
 break Anweisung, switch Kommando 79
 break Kommando 76
 BreakDemo Programm-Code 76, 77
 breakpoint Kommando 175
 Build Kommando 169
 Buttons, Neue Textdatei, Textdateien erzeugen 10

C

C++
 Code eingeben 20 - 23
 Compiler 11, 298, 299
 Präprozessor
 C++ Programmiersprache
 Conversion.cpp Programm-Code 30
 EXE-Programme 10
 Fehlermeldungen 14, 25, 26
 GNU C++ 8
 Programm, Ausgabe 16
 Programme, ausführen 14
 Programme, erzeugen 10, 14
 Unterscheidung Groß- und Kleinschreibung 9
 Visual C++ 8
 Zeilen einrücken 9
 CashAccount
 Klasse, abstrakte Unterklasse 271, 272
 Programm-Code 271, 272
 Cast 87
 Cast-Operator 321 - 323
 catch-Phrasen 354 - 357
 CD-ROM
 C++-Wochenend-Crashkurs 8
 Concatenate(Error).cpp Datei 171
 ConstructMembers Programm 213
 Conversion.cpp Datei 10, 21
 factorial() Funktion 348
 FactorialThrow.cpp 355; 365
 StudentID Programm 227
 ToStringWStreams Programm 340; 341
 cerr Objekt 333
 char Variable 47

- CharDisplay Programm-Code 111, 112
 - Checking Klasse 263 - 266
 - cin Eingabe-Objekt 332
 - cin Objekt 333
 - class Schlüsselwort 123
 - class StudentID Elementobjekte, Student Konstruktor 223 - 225
 - ClassData Programm-Code 124, 125
 - clog Objekt 333
 - Code
 - Abwickeln des Stacks (Ausnahme-Behandlung) 352; 353
 - Anwendungen, aufteilen 291 - 293
 - Beispiel-Code 81, 82, 83
 - bitweiser Operator, testen 65, 66
 - C++, Eingabe 20 - 23
 - Cast-Operator überladen 321 - 323
 - cpp Quellcode-Datei 12
 - debuggen, #ifdef Direktive 304, 305
 - display()-Funktion mit Manipulatoren 341
 - displayArray()-Funktion, Integer anzeigen 144, 145
 - factorial()-Funktion 349
 - Konstanten definieren 299, 300
 - leere Dateien erzeugen 21, 22
 - leere Textdateien erzeugen 10, 11
 - Makros definieren 300 - 302
 - Makros Fehler 300 - 302
 - MYNAME Datei öffnen und schreiben 334, 335
 - Objekte auslösen 354 - 356
 - Operatoren als Elementfunktion implementieren 319, 320
 - 0xff Exit-Code rhide 96
 - Student Programm 246 - 248, 290
 - Student-Klasse Elementfunktions, deklarieren außerhalb der Klasse 296
 - USDollar::outpout(), Klasse ostrstream 340, 341
 - Zahlen mitteln 43
 - Zuweisungsoperator überladen 327 - 329
 - Compile-Menü-Kommandos, Make 295
 - Compiler
 - C++ 8, 11, 298, 299
 - Objekte, konvertieren 323
 - Visual C++ 8
 - Computer
 - Concatenate (Error).cpp Datei 171
 - Maschinensprache 23
 - Prozessoren 4
 - Concatenate Programm-Code 114, 115, 171
 - ConcatenatePtr Programm-Code 146, 147
 - concatString()-Funktion
 - Programmabsturz 173
 - Segmentverletzung 175
 - 178, 179
 - ConstructElements-Programm
 - Ausgabe nach Destruktor eingefügt 213
 - CD-ROM 213
 - Code 211, 212
 - Container 168
 - Conventional Klasse 266
 - Conversion Programme, mathematische Operatoren 53
 - Conversion.cpp
 - Datei CD-ROM 10, 21
 - erzeugen 11
 - erzeugen in GNU C++ 24
 - Programm 11
 - Programm-Code 30
 - Conversion.exe
 - ausführen 15
 - Programm 15
 - Conversion.pdw Arbeitsbereich 11
 - CopyStudent Programm-Code 232, 233
 - cout Objekt 332, 333
 - cpp Dateien, mit C++-Präprozessor bearbeiten 298, 299
 - cpp Quellcode-Datei 12
 - Ctrl+F Tastaturkürzel 26
- D**
- Datei Menü Kommandos
 - Arbeitsbereich schließen 10
 - Neu 21
 - Speichern als 22, 23

400 Index

Datei MYNAME öffnen, Code zum Öffnen und Schreiben 334, 335

Dateien

Concatenate (Error).cpp 171
 Conversion.cpp CD-ROM 10, 21
 Conversion.exe, ausführen 14, 15
 Conversion.pdw Arbeitsbereich 11
 cpp mit C++-Präprozessor bearbeiten 298, 299
 cpp Quellcode 12
 DJGPP.ENV 20
 einbinden 199, 200, 291
 einbinden, Funktionen 298, 299
 EXE erzeugen 11
 fstream.h 334
 .h #include Direktive 298, 299
 .h 295
 Include-Dateien #include Direktive 200
 ios Konstanten zum Öffnen 334
 iostream.h Prototypen 333
 leer, Code zu Erzeugen 21, 22
 leere Textdatei, Code zum Erzeugen 10, 11
 myClass.h 303, 204
 MYNAME, Code zum Öffnen und Schreiben 334, 335
 Projekt erzeugen in GNU C++ 295
 Projekt erzeugen in Visual C++ 294
 Quellmodul 89, 90
 README.1ST 19
 SeparatedFn.cpp 292
 SeparatedKlasse.cpp 288, 289, 291, 292
 strtrea.h, Definition der Unterklassen ostream 337
 student.cpp Datei 199
 Tex, erzeugen 9 - 11

Daten gruppieren 119 - 122

Datenelemente

Klassen, konstruieren 210 - 212
 Klassen, static 217, 218
 Objekte, konstruieren 223 - 228
 static, Syntax zum Deklarieren 217, 218
 Syntax zum Deklarieren 228

DEBUG Parameter 305

Debug Windows Kommando (View Menü) 179, 180

debuggen 169

Ausgabeeweisungen 92 - 95
 Code, #ifdef Direktive 304, 205

concatString()-Funktion 178, 179

debug-Funktion 304, 305

debug-Kommandos 170

debug-Modus 97

Debugger-Programm 92, 93

Debugger Visual C++ 179, 180

Debugger, Aktion anhalten 173

Einzelschritte durch Programme 172, 173

ErrorProgram korrigieren 99, 100

ErrorProgram 93, 94, 95

Fehlertypen 92, 93

Funktion 304, 305

GNU C++ 96, 97

Haltepunkt 175

Kommandos 170

lokale Variablen ansehen 179, 180

Modus 97

nNums auswerten 98

Probleme, reproduzieren 95

Program Reset 174

Programme in Einzelschritten 172, 173

Programme mit Kommando Step Over 172, 173

Release-Modus 97

rhide-Debugger 172

Segmentverletzung 175

Step In Kommando 174

Testprogramm 171, 172

Variablen modifizieren 175 - 179

Variablen, Werte 99

verkettete Liste, Anwendung 176

Visual C++ 95

Visual C++, Debugger 179, 180

Visual C++, nNums 95, 96

Zeichenketten mit null terminieren 177

Zeigerfehler, Ausgabeeweisungen 170

Zielzeichenkette 177

Zugang der Ausgabeeweisungen 92, 93

Debugger

Aktion anhalten 173

Konstruktoren, Ausgabeeweisungen einfügen 209

Programm 92, 93

Visual C++ 179, 180

dec Manipulatoren 342

Default

Definition, operator=() 325, 326

Konstruktoren 222, 223, 209

Werte, Argumente 223

DefaultStudentID Programm-Code 223 - 225

definieren

aktive Klassen 195 - 197

Algorithmen 4

Anweisungen 32

Arrays 105

Ausdrücke 34

ausführbare Anweisung 172

Bits 62

CNU C++ 18

Container 168

Definitionen 198

Deklarationen 32, 198

Elementfunktionen 197, 198

Faktorisieren 265

Haltepunkte 175

ios::out 334

Kommentare 31

Konstanten, Code 299, 300

Konstruktoren mit Argumenten 221, 222

Makros Code 300 - 302

Manipulatoren 341

nicht uneindeutig 204

Objekte 120

Operationen auf Zeigertypen 140, 141

Operatoren 34

ostream Klasse 337

Polymorphie 255

Programmierung 3, 4

Seiteneffekt 229

stringstream Klasse 337

sumSequence() Funktion 83

vertrauenswürdige Funktionen 217

Zeichenketten 113

Ziffern 62

Zugriffsfunktionen 216, 217

Definitionen

definieren 198

operator=(), Default 325, 326

Deklarationen

abstrakte Klassen 267, 268, 269

Arrays 105

C++-Programme 32, 33

Datenelemente, Syntax 228

Definition 32, 198

Elementfunktionen außerhalb der Klasse 296

Elementfunktionen identisch 259, 260

linkbare Klassen 160

nSum verändern 98

Objekte initialisieren 206, 207

static Datenelements, Syntax 217, 218

Variablen 32, 33

von Funktionen 89, 90

Dekrement-Operatoren 56, 57

Delorie Web-Site 19

DemoAssign Programm-Code 327 - 329

Destruktoren

aufrufen 231

Ausgabe nach Einfügen in Programm

ConstructElements 213

Basisklassen 248

ofstream Klasse 336

virtual 261, 262

von Objekten 207, 212, 214

dezimale Zahlen 42 - 46

Direktiven

#define 299 - 302

#if 302, 303

#ifdef 303

#ifdef, Einschlusskontrolle 303, 304

#include <iostream.h> 295

#include <stdio.h> 295

#include 200, 290, 291

#include, .h-Dateien 298, 299

display() Elementfunktion (virtual), schlaue Inserter 344

display() -Funktion, Code mit Manipulatoren 341

displayArray() -Funktion 110, 144, 145

DisplayString-Programm 112, 113

displayString() -Funktion 113

Divisions-Operator 53

Division-vor-Addition Algorithmus 44

DJGPP.ENV-Datei 20

do while-Schleife 72

Doppelkreuz (#) 298, 299

Doppelslash (//) 31

doppelte Hochkommata (") 113

double-Variable 47, 128

dumpState() -Funktion 304, 305

402 Index**E**

EarlyBinding Programm-Code 253

Eigenschaften

- aktive, Klassen hinzufügen 193, 194
- Objekte, Syntax für Zugriff 123
- verkettete Listen 164

einen platten Reifen wechseln

- Algorithmus 4
- Programm 4
- Prozessor 4

einfache Funktionen 85

einfache Hochkommata (') 113

einfache Hochkommata und Unterstrich ('_') 34

einfache logische Operatoren 59 - 62,

- != 60
- && 60
- < 60
- <= 60
- == 60
- > 60
- >= 60

Eingabe von C++-Code 20 - 23

Eingabe/Ausgabe Anweisungen 33

Einschlusskontrolle

- #ifdef Direktive 303, 304

Einzelbit-Operatoren 63

Einzel Schritte Programme 172 - 175

Elemente von Klassen konstruieren 230, 231

Elemente am Kopf der Liste angefügt 162, 163

ends Konstante, Inserter-Kommando 338

ErrorProgram 92, 93, 94, 99, 100

- Code 92, 93, 94, 99, 100
- rhide Oxff Exit-Code 96

erzeugen

- Arbeitsbereiche 11
- C++-Programme 10, 11
- Conversion.cpp Programm 11
- Conversion.cpp Programm in GNU C++ 23
- EXE-Dateien 11
- geschützter Kopierkonstruktor 234, 235
- GNU C++ Programme 23 - 26
- leere Dateien 21
- leere Dateien, Code 21, 22

Objekte 189, 190, 206 - 213

Programme 10, 14

Projektdateien in GNU C++ 295

Projektdateien in Visual C++ 294

Textdateien 9, 11

EXE-Dateien erzeugen 11

EXE-Programme 8

Extraktoren

- inp> Extraktor-Anweisung 337, 338
- operator>() 333

F

F1 key 17, 27

factorial Funktion()

- CD-ROM 348
- Code 348
- Fehlerrückgaben 349

FactorialException Programm-Code 350, 351

FactorialThrow.cpp CD-ROM 355, 356

Faktorisieren

- Definition 265
- Vererbung 263 - 266 - 274

Faktorisieren von Klassen 263

- Account 266
- Checking 263 - 266
- Conventional 266
- Savings 263 - 266

FalseStudentID Programm-Code 225, 226

Feature Autodekrement 72, 73

Fehler

- Ausnahmebehandlung 352, 353
- behandeln 348, 349
- Compile-Zeit 92, 93
- Fehlerrückgaben 348 - 350
- fn(...) Deklaration 354
- Funktionen, fn(...) Deklaration 354
- GNU C++ Installation 24
- Laufzeit 92, 93
- Makros 300 - 302
- Typen von 92, 93
- Zeichenketten, abschließende Null 177
- Zeiger, Ausgabeanweisungen 170

- Fehler, Behandlung
 - Ausnahme-basierte Stream-Ausgabe 335
 - catch-Phrase 354 - 356
 - catch-Phrasen, verbunden mit try-Block 357
 - Objekte, auslösen, catch-Phrasen 354 - 356
- Fehler-Flag
 - filebuf::openprot Wert 334
 - filebuf::sh_none Wert 334
 - filebuf::sh_read Wert 334
 - filebuf::sh_write Wert 2334
 - ifstream Objekt 337
 - ios::ate Konstante 334
 - ios::binärer Konstante 334
 - ios::in Konstante 334
 - ios::nocreate Konstante 334
 - ios::noreplace Konstante 334
 - ios::out Konstante 334
 - ios::trunc Konstante 334
 - nicht null 336
- Fehlermeldung durch null geteilt 95
- Fehlermeldungen 12, 97
 - C++ 12, 25
 - Teilen durch null 95
 - undeklarerter Bezeichner 293
 - Visual C++ 12, 95, 96
- Fenster
 - Arbeitsbereich 9
 - Ausgabe 10
 - Auswerten und Modifizieren 176
 - rhide 22
 - Schließen 9
 - Variablen 179, 180
- FIFO (first-in-first-out) 168
- filebuf::openprot Wert 334
- filebuf::sh_none Wert 334
- filebuf::sh_read Wert 334
- filebuf::sh_write Wert 334
- first-in-first-out (FIFO) 168
- flache Kopien und tiefe Kopien 233, 234
- float-Variable, Speicher 128
- Flusskontrolle, Kommandos 69
 - Autdekrement Feature 72, 73
 - break 76
 - break-Anweisungen 79
 - for-Schleife 74, 75, 76
 - geschachtelte Schleifen 78, 79
 - Schleifen 71 - 77
 - Schleifenkontrollen 76 - 77
 - switch-Anweisung 79
 - switch-Kommando 79
 - Verzweigung 69, 70, 71
- fn()-Funktion 90
- ForDemo Programm-Code 74, 75
- Formate für Klassen 122 - 124
- for-Schleife 74 - 76, 98
- FORTTRAN, WRITE-Anweisung 92, 93
- Free Software Foundation GNU C++ 18
- friends von Klassen 315
- frühe Bindung 253
- fstream-Unterklassen 334 - 337
- fstream.h-Datei 334
- FunktionDemo, Programm-Code 81, 82, 83
- Funktionen 81, 89, 90, 197, 198, 204, 305
 - abstrakte Objekte übergeben 272 - 274
 - addHead() 217, 218
 - addTail() 162, 163, 167
 - Argumente 84 - 87
 - Ausgabe 117
 - Beispiel-Code 81 - 83
 - concatString() 173, 175, 178, 179
 - debuggen 304, 305
 - display(), Code mit Manipulatoren 341
 - displayArray() 110, 144, 145
 - displayString() 113
 - dumpState() 304, 305
 - Einzelschritte 173 - 175
 - factorial() 348, 349
 - fn() 90
 - fn(...) Deklaration 354
 - getData() 167
 - Include-Dateien 199, 200, 298, 299
 - int strcmp(source1 source2) 116
 - int strlen(string) 116
 - int strstr 116
 - konkretes Element 269
 - main() 88-90, 115, 152 - 154
 - Namen 88
 - Nichtelement 195, 320, 321
 - Operatoren 312, 315
 - parseString() 337, 338

404 Index

- printf() 117
- protected, Zuweisungsoperator überladen 330, 331
- Prototypen 89, 90
- rationalize() 316
- rein virtuell 267, 268, 272 - 274
- remove() 162, 163
- Rückgabetypp 85
- setw()-Funktion 342
- someFunktion() 88
- square() 85 - 87
- Step In Kommando 174
- sumArray() 110
- sumSequence(), aufrufen oder definieren 83
- überladen 222, 223
- Variablen speichern 90
- vertrauenswürdige Funktionen, definiert 217
- virtual 259 - 261
- void Schlüsselwort 84, 85, 116
- width() 342
- Zeichenketten manipulieren 116, 117
- Zeigervariablen, Argumente übergeben an 133 - 135
- Zugriffsfunktion, definiert 216, 217
- Funktionen, Elemente 195
 - aufrufen 200 - 203
 - außerhalb der Klasse deklarieren 296
 - definieren 197, 198
 - identisch deklarieren 259, 260
 - Namen vergeben 196, 197
 - operator=() 330
 - Operatoren 321
 - schreiben 198 - 200
 - überladen 204, 205
 - überladen in Unterklassen 252
 - überschreiben 252 - 253
 - Zugriff auf Elemente 201 - 203
- G**
 - gebrochene Werte, Integer 42
 - Geltungsbereich von Variablen 135 - 138
 - gemischte Ausdrücke, definiert 50
 - geschachtelte Schleifen 78, 79
 - getData()-Funktion 167
 - gleich 34
 - Gleichheitsoperator (==) 60
 - Gleitkomma
 - Variablen, Begrenzungen 46
 - Zahlen (floats) 45
 - globale Objekte konstruieren 229, 230
 - GNU C++ 8
 - BIN Verzeichnis 19
 - C++-Code eingeben 20 - 23
 - Conversion.cpp, Programm erzeugen 23
 - Dateien, erzeugen 21
 - definiert 18
 - Delorie Web-Site 19
 - DJGPP.ENV Datei 20
 - Fehlermeldungen 96, 97
 - Free Software Foundation 18
 - Hilfe 27, 28
 - Installation vom Web aus 19, 20
 - Installationsfehler 24
 - Maschinensprache 23
 - Meldungen 24
 - Proektdatei erstellen 295
 - Programme, ausführen 26
 - Programme, erstellen 21 - 23
 - Programme, erzeugen 23 - 26
 - Prozess des Fehlermeldens 24, 25
 - rhide-Fenster 22
 - rhide-Interface 21, 22
 - Windows-Programme entwickeln 27
 - zip-Dateien 19
 - Go Kommando 169
 - Go Menü Kommandos 14
 - Gold-Stern-Programme 92
 - GraduateStudent-Objekt 248 - 250
 - Größer-als-Operator (>) 60, 61
 - Größer-oder-gleich-Operator (>=) 60
 - grundlegender Rahmen von C++-Programmen 30
 - GSInherit Programm-Code 242 - 245

H

.h-Dateien 295, 298, 299
 Haltepunkte 175
 HAS_A Beziehung, GraduateStudent-Objekt 249, 250
 Heap
 Container 168
 Speicher 135 - 138, 234, 316, 317
 Zeiger 159
 hex Manipulatoren 342
 hexadezimale Zahlen 63, 66
 Hilfe
 GNU C++ 27, 28
 MSDN Bibliothek 16
 Visual C++ 16, 17
 Hilfe Menü Kommandos, Index 16, 27
 hinzufügen
 Ausgabe-Anweisungen im Konstruktor 209
 Element ans Ende einer Liste 162, 163
 Objekt am Kopf einer Liste 162

I

if-Anweisung demonstrieren 69 - 71

ifstream

 Klasse, Dateieingabe 336
 Objekt, Fehlerflag 337

Icons, Neue Textdatei 9

implementieren

 SleeperSofa-Klasse 282, 283, 284
 Vererbung 242 - 245

include-Anweisungen 31

Include-Dateien 199, 291

 #include-Direktive 200
 Funktionen 298, 299

Index-Kommando (Hilfe Menü) 16, 27

Initialisierung

 Objekte 206, 207
 Zeichenketten 113
 Zeiger 160

Inkrement-Operatoren 56 ,57

inline

 Funktionen 197, 198
 Versionen von Funktionen die nicht funktionieren 305
 virtuelle Funktionen 261

inp> Extraktor-Anweisung 337, 338

Insert-Kommando, ends Konstante 338

Installation

 GNU C++ vom Web aus 19, 20
 Visual C++ 8

Installationsfehler, GNU C++ 24

Instanz einer Klasse 189

int strcmp(source1 source2) Funktion 116

int strlen(string) Funktion 116

int strstr Funktion 116

int Variable

 begrenzte Bereiche 45
 Begrenzungen 42 - 44
 Division-vor-Addition Algorithmus 44
 Integer runden 42 - 45
 Integer abschneiden 44, 46
 Speicher 128

Integer

 Abschneiden 44
 Abschneiden, Problem lösen 44, 46
 displayArray() Funktion Code 144, 145
 gebrochene Werte 42
 Runden 42 - 45

Integrität von Objekten 206

Interface rhide 21, 27, 28

InvalidArgumentException::display(), Ausnahme-Klasse 356

ios-Klasse

 ios::out, definiert 334
 Konstanten zum Öffnen von Dateien 334

ios::ate, Konstante 334

ios::binary, Konstante 334

ios::in, Konstante 334

ios::nocreate, Konstante 334

ios::noreplace, Konstante 334

ios::out, definiert 334

ios::out, Konstante 334

406 Index

ios::trunc, Konstante 334
 iostream.h Datei, Prototypen 333
 istrstream Klasse, definiert 337

K

Kapselung 288, 289
 Klammern () 69, 70, 81, 82, 157
 Klassen 125
 Account 266 - 268
 aktive 192
 aktive, definieren 193 - 197
 Ausnahme, InvalidArgumentException::display() 356
 Basisdestruktor 248, 268
 Bereichsauflösungsoperator 196
 CashAccount, abstrakte Unterklasse 271, 272
 Checking 263 - 266
 Conventional 266
 Dateien, einbinden 199, 200
 Datenelemente konstruieren 210 - 212, 217, 218, 223 - 228
 Definitionen, definieren 198
 Deklarationen, definieren 198
 Destrukturen von Objekten 212, 213
 Elemente erzeugen 230, 231
 Faktorien 263 - 266
 Formate 122 - 124
 friend 315
 fstream Unterklassen 334 - 337
 ifstream, Dateieingabe 336
 Include-Dateien 199, 200
 Inline-Funktionen 197, 198
 Instanzen 189
 ios, ios::out definiert 334
 istrstream, definieren 337
 konkrete aus abstrakter Klasse erzeugen 269 - 271
 konkrete Unterklassen, Implementierung rein virtueller Funktionen 273
 linkbar, definieren 160
 Nichtelement-Funktionen 195
 Objekt, Zugriff auf Eigenschaften 123
 ofstream 334, 336
 ostrstream USDollar::outpout()-Code 340;341

ostrstream, definieren 337
 Savings 263 - 266
 Savings, abstrakte Unterklasse 271, 272
 selbstenthaltend 190, 191
 SleeperSofa, implementieren 282 - 284
 strstream, definieren 337
 struct Schlüsselwort, Grenzen 193, 194
 Struktur 192, 193
 Student, Konstruktoren 222, 223
 Superklasse 189
 Unterklassen 189, 246, 247 - 249, 253
 vertrauenswürdige Funktionen, definiert 217
 Zugriffsfunktionen, definiert 216, 217
 Zusammenfassung 192 - 194
 Zuweisungsoperator, mit geschützter Funktion überladen 330, 331
 Klassen aktive Eigenschaften hinzufügen 193, 194
 Klassen, abstrakt 271, 272
 Basis 268
 deklarieren 267 - 269
 konkrete Elementfunktion 269
 rein virtuelle Funktionen 267, 268, 273, 274
 rein virtuelle Funktionen überladen 272
 Klassen, Elementfunktionen 195
 aufrufen 200 - 203
 definieren 197, 198
 deklarieren außerhalb von 296
 Namen vergeben 196, 197
 schreiben 198 - 200
 überladen 204, 205
 Klassen, Funktionen
 Element aufrufen 200 - 203
 Element definieren 197, 198
 Element 195
 inline 197, 198
 Element Namen geben 196, 197
 Nichtelement 195
 Element überladen 204, 205
 Klassen, Vererbung
 implementieren 242 - 245
 konstruieren Unterklassen 246 - 249
 public Schlüsselwort 241
 Vorteile 241
 Klassifizierung in der objektorientierten Programmierung 188, 189
 Kleiner-als-Operator (<) 60

- Kleiner-oder-gleich-Operator (<=) 60
- Komma (,) 87
- Kommandos
 - #include Namen konstruieren 298, 299
 - Add Watch 169
 - break 76
 - Build 169
 - C++-Programme 31
 - debuggen 170
 - definiert 32
 - Flusskontrolle 69
 - Go 169
 - Go Menü 14
 - Haltepunkt 175
 - Haltepunkt setzen 169
 - Insert, Konstante ends 338
 - Program Reset 169
 - Schleife, innere Schleife 78, 79
 - Step In 169, 174
 - Step Over 169
 - Step Over, Programme debuggen 172, 173
 - switch()-Kontrolle 151
 - switch, break-Anweisungen 79
 - View User Screen 169
 - View Variable 169
- konkrete Element-Funktion 269
- konkrete Klasse aus abstrakter Klasse erstellen 269 - 271
- konkrete Unterklassen, rein virtuelle Funktionen implementieren 273
- Konstanten
 - #define Direktive 299, 300
 - Code zum Definieren 299, 300
 - ends, Insert-Kommando 338
 - ios::ate 334
 - ios::binärer 334
 - ios::in 334
 - ios::nocreate 334
 - ios::noreplace 334
 - ios::out 334
 - ios::trunc 334
 - Variablen 48
- konstruieren
 - Basisklassen 248
 - Datenelemente von Klassen 210 - 212
 - Datenelement-Objekte 223 - 228
 - Elementobjekte mit Klassentyp 223 - 228
 - globale Objekte 229, 230
 - Klassenelemente 230, 231
 - lokale Objekte 229
 - Namen in #include Kommando 298, 299
 - Objekte mehrfacher Vererbung 285
 - static Objekte 229
 - UnterKlassen 246 - 249
- Konstruktoren
 - Argumente 220 - 223
 - Ausgabe 225
 - Ausgabeelemente einfügen 209
 - Begrenzungen 209, 210
 - Datenelemente, Syntax der Deklaration 228
 - Default 209, 222, 223
 - Destruktoren, aufrufen 231
 - flache Kopie gegen tiefe Kopie 233, 234
 - globale Objekte konstruieren 229, 230
 - Heap-Speicher 234
 - Kopier- (protected) erzeugen 234, 235
 - Kopier- 231, 232
 - lokale Objekte, konstruieren 229
 - ofstream Klasse 334
 - Reihenfolge der Konstruktion 228 - 231
 - Seiteneffekt, definiert 229
 - static Objekte, konstruieren 229
 - Student-Klasse 222, 223
 - Student-Klasse, StudentID Element-Objekte 223 - 225
 - überladen 221, 222
 - Vergleich Kopieren und Zuweisen 326, 327
 - virtual 261
 - void 209, 222, 223
 - von Objekten 207
- Kontrolle für Schleifen 75 - 77
- Konvertieren von Objekten 323
- Kopfzeiger auf Liste, Objekte einfügen 161
- Kopierkonstruktor 231, 232
 - Kopie, flache gegen tiefe 233, 234
 - protected, erzeugen 234, 235
 - Vergleich mit Zuweisungsoperator 326, 327
- kurze Schaltkreise 62
- Kurznamen von Funktionen 88

408 Index**L**

last-in-first-out (LIFO) 168
 LateBinding-Programm-Code 257 - 259
 Laufzeitfehler 92, 93
 Layout Programm-Code 128, 129
 LayoutError Programm-Code 132, 133
 leere Datei, Code zur Erzeugung 21, 22
 leere Textdatei, Code zur Erzeugung 10, 11
 Leerraum (Anweisungs-) 32
 Lesbarkeit, verbessern mit Überladen von Operatoren 311, 312
 LFN=y Phrase 21
 LIFO (last-in-first-out) 168
 linkbare Klassen deklarieren 160
 LinkedListData Programm-Code 165 - 167
 Links-Shift (<<) Operator 311, 312
 Links-Shift Operator 346, 347
 logische Operatoren
 ! (einfacher) 60
 != (einfacher) 60
 && (einfacher) 60
 & (einfacher) 60
 < (einfacher) 60
 <= (einfacher) 60
 == (einfacher) 60
 > (einfacher) 60
 >= (einfacher) 60
 bitweise 58, 63 - 67
 bitweise, binäre Zahlen 63
 einfacher 58, 60 - 62
 kurze Schaltkreise 62
 logische Variablen 62
 lokale Objekte konstruieren 229
 lokale Variablen ansehen 179, 180
 long-Variable 47, 128

M

main()-Funktion 88 - 90, 115, 152, 154
 Make Kommando (Compile Menü) 295

Makros

#define Direktive 299, 300 - 302
 Code zum Definieren 300 - 302
 Fehler 300 - 302

Manipulation von Zeichenketten: Arrays gegen
 Zeiger 145, 146

Manipulatoren

dec 342
 definieren 341
 display()-Funktion 341
 hex 342
 oct 342
 setfill(c) 342
 setprecision(c) 342
 setw(n) 342
 stream I/O 341 - 343

Manipulieren von Zeichenketten 114 - 117

Maschine

Anzahl Instruktionen ind Anzahl Anweisungen
 147
 Sprachen 23

Masken 67**mathematische Operatoren 53**

% 53
 %= 53
 - (unär) 53
 — (unär) 53
 -, 53
 * 53
 *= 53
 /, 53
 + (unär) 53
 ++ (unär) 53
 += 53
 = 53
 -= 53
 arithmetische 53, 54
 Ausdrücke 54
 binärer 55, 56
 Dekrement 56, 57
 Inkrement 56, 57
 unär 55 - 58
 Zuweisung 57, 58

mathematischer Operator 53

Matrix, Arrays 111, 112
 aufrufen 200- 203

- definieren 197, 198
- deklarieren, außerhalb der Klasse 296
- deklarieren, identisch 259, 260
- display() (virtual), schlaue Inserter 344
- Elemente zugreifen 201 - 203
- Namen geben 196, 197
- nicht uneindeutig, definiert 204
- operator=() 330
- Operatoren 321
- schreiben 198 - 200
- Student-Klasse, Code außerhalb der Klasse
- deklarieren 296
- überladen 204, 205
- überladen in Unterklassen 252
- überschreiben 253
- mehrfache Vererbung
 - Nachteile 285, 286
 - Objekte konstruieren 285
 - SleeperSofa-Programm 277, 278
 - SleeperSofa-Speicheranordnung 281, 282, 284
 - Uneindeutigkeit 278, 279
 - virtuelle Vererbung 279 - 285
- mehrfache Vererbung 276
- Meldung von Fehlern
 - C++ 25
 - Prozess des Meldens 24, 25
- menschliches Programm
 - Algorithmus 4
 - Programme 4
 - Prozessoren 4
- Mischen von Variablen 50, 51
- Mittlung Programm, mathematische Operatoren 53
- mittleren Grad berechnen (dAverage) 193, 194
- Modelle (Paradigmas) 191
- Modi, Debug oder Release 97 97
- Module 89, 90, 288, 289
- Module linken 288, 289
- modulo-Operator 53, 54
- MSDN Bibliothek 16
- MS-DOS Backslash () 49
- MultipleVirtual Programm-Code 282 - 284
- Multiplikation, Operatoren 53
- MyClass, myclass.h Datei 303, 304

- MYNAME-Datei, Code zum Öffnen und Schreiben 334, 335
- MySpecialClass-Objekt, Debug-Funktion 304, 305

N

- Name& (Rückgabotyp) operator=() 329, 330
- NamedStudent Programm-Code 221, 222
- Namen
 - konstruieren in #include Kommando 298, 299
 - von Funktionen 88
- Namen vergeben
 - Elementfunktionen 196, 197
 - Variablen 33, 51
 - Zeichenketten 113
- nArray, Zugriff 108
- NestedDemo Programm-Code 78, 79
- Neu Kommando (Datei Menü) 21
- Neue TextDatei
 - Button, neue Textdatei erzeugen 10
 - Icon 9
- nGlobal Variable 90
- nicht initialisierte verkettete Zeiger 167
- nicht null Fehler-Flag 336
- nicht uneindeutig, definiert 204
- Nichtelement-Funktionen 195, 320, 321
- nInputValues Array 110
- nInt Adresse, speichern in pInt 131
- nLocal Variable 90
- nLoopCount Variable 72
- nNums 95, 96, 98
- NOT bitweiser Operator (~) 64
- NOT Operator (!) 60
- nStatic Variable 90
- nSum
 - auswerten 98
 - deklarieren, ändern 98
- null (terminieren), Zeichenketten 177
- numerische Typen, Cast 87
- nValue, auswerten 98

410 Index**O****Objekte**

- abstrakt, an Funktionen übergeben 272, 273
- am Kopfzeiger der Liste eingefügt 162
- auslösen `__FILE__` 355
- auslösen `__LINE__` 355
- auslösen, catch Phrasen 354 - 356
- cerr 333
- cin 333
- cin Eingabe 332
- clog 333
- cout 332, 333
- Datenlemente 210 - 212, 225, 226
- definiert 120
- Destruktoren 207, 212, 213
- Eigenschaften, Syntax für Zugriff 123
- erzeugen 189, 190, 206 - 212, 214
- globale, konstruieren 229, 230
- GraduateStudent 248 - 250
- identifizieren in objektorientierter Programmierung 188, 189
- ifstream, Fehler-Flag 337
- initialisieren 206, 207
- Integrität 206
- Klasse StudentID Element, Student Konstruktor 223 - 225
- Konstruktor 207
- Konstrukturen, Begrenzungen 209, 210
- konvertieren 325
- Kopierkonstruktor im Vergleich mit Zuweisungsoperator 326, 327
- lokale, konstruieren 229
- Manipulatoren 341
- Mehrfachvererbung, konstruieren 285
- MySpecialClass-Objekt, Debug-Funktion 304, 305
- ostream, out Argument 341
- out 337, 338
- parallele Arrays 120 - 122
- Reihenfolge der Konstruktion 228 - 231
- static, konstruieren 229, 230
- StreamI/O 333
- temporäre 317
- übergeben 157 - 159
- vernichten 206 - 212, 214
- vertrauenswürdige Funktionen, definiert 217
- Zeiger 156 - 159
- Zugriffsfunktion, definiert 216, 217
- Zugriffskontrolle 214, 217, 218
- Zugriffskontrolle, private Schlüsselwort 216
- Zugriffskontrolle, protected Schlüsselwort 214 - 217
- Zugriffskontrolle, public Schlüsselwort 214
- Zuweisungsoperator, Vergleich mit Kopierkonstruktor 326, 327
- Objekte auslösen
 - `__FILE__` 355
 - `__LINE__` 355
 - catch Phrasen 354 - 356
- Objekte identifizieren, in objektorientierter Programmierung 188, 189
- Objekte übergeben 157 - 159
- objektorientierte Programmierung
 - Abstraktion 187, 188
 - Abstraktionslevel 188
 - Kapselung 288, 289
 - Klassen, selbstenthaltend 190, 191
 - Klassifizierung 188, 189
 - Objekte, erzeugen 189, 190
 - Objekte, identifizieren 188, 189
 - Paradigmas 191
 - Polymorphie 255 - 257
- oct Manipulatoren 342
- Offsets, Zeiger-Arrays 142
- ofstream Klasse
 - Destruktor 336
 - Konstruktor 333
- ofstream, Konstruktor 335
- Operationen
 - definiert auf Zeigertypen 140, 141
 - Zeiger + Offset 140, 141
 - Zeiger-Offset 140, 141
 - Zeiger2 Zeiger1 140, 141
 - Zeigertypen 148
- Operator, Operatoren überladen
 - Beziehungen dazwischen 316
 - binäre, Verändern von Argumentwerten 318
 - Cast 321 - 323
 - Code als Elementfunktion 319, 320
 - Elementfunktionen 321
 - Funktionen, Beziehungen dazwischen 312
 - Nichtelement-Funktionen 321
 - unärer operator++(), Argumente 317, 318

- operator+() 315 - 318
 - operator++()-Funktion 315
 - operator<<() 333, 343, 344
 - Ausgabe 343, 344
 - schlau 344 - 347
 - USDollar Programm-Code 343, 344
 - operator<<() (Insertor), Stream I/O 333
 - operator=()
 - Default-Definition 325, 326
 - Elementfunktion 330
 - Name& (Rückgabotyp) 329, 330
 - Operator>() (Extraktor) Stream I/O 332
 - Operatoren
 - Addition 53
 - AND (&&) 60 - 61
 - AND 63
 - arithmetische 53, 54
 - Bereichsauflösung 196
 - Beziehung dazwischen 316
 - binäre 55, 56, 318
 - Cast 321 - 323
 - Code als Elementfunktion 319, 320
 - definiert 34
 - Dekrement 56, 57
 - Division 53
 - einfacher logischer 73, 62
 - Einzelbit 63
 - Elementfunktionen 321
 - Funktionen, Beziehungen dazwischen 312
 - Gleichheit (==) 60
 - größer als (>) 60 - 62
 - größer oder gleich (>=) 60, 61
 - Inkrement 56, 57
 - kleiner als (<) 60, 61
 - kleiner oder gleich (<=) 60, 61
 - Links-Shift 346, 347
 - Links-Shift (<<) 311, 312
 - logische 62 - 67
 - mathematische 53
 - modulo 53, 54
 - Multiplikation 53
 - Nichtelement-Funktions 321
 - NOT (!) 60
 - operator+() 316 - 318
 - operator<< (Insertor) Stream I/O 333
 - operator<<() 343, 344
 - operator>() (Extraktor) Stream I/O 333
 - OR () 60, 64
 - Postfix x++ 316
 - Präfix ++x 316
 - Rechts-Shift (>) 311, 312
 - Shift 346; 347
 - Subtraktion 53
 - überladen, operator<<() 333
 - unäre 55 - 58
 - unärer operator++(), Argumente 317, 318
 - Ungleichheit (!=) 60
 - XOR 64
 - Zeiger 129
 - Zuweisung (=) 311, 312
 - Operatoren bitweise
 - Code zum Testen 65, 66
 - logische 58, 63 - 67
 - Masken 67
 - Zweck 66, 67
 - Operatoren, Zuweisung 34, 57
 - Code zum Überladen 327 - 329
 - überladen 325 - 330
 - Vergleich mit Kopierkonstruktor 326, 327
 - OR bitweiser Operator (!) 64
 - OR Operator (||) 60
 - OR Operator 64
 - ostream-Objekt, out Argument 341
 - ostrstream Klasse
 - defining 337
 - USDollar::output() Code 340, 341
 - out Argument, ostream Objekt 341
 - out Objekt 337, 338
- P**
- p (Präfix) Zeiger Variablen 130
 - Paradigmas 191
 - parallele Arrays 120 - 122
 - ParallelData Programm-Code 120 - 122
 - Parameter
 - DEBUG 305
 - parseString()-Funktion 337, 338
 - PassObjektPtr-Programm-Code 157 - 159

412 Index

- Phrase LFN=y 21
- plnt, speichert Adresse von nlnt 131
- platter Reifen
 - Algorithmus zum Wechseln 4
 - Programm zum Wechseln 4
 - Prozessor zum Wechseln 4
 - Polymorphie 252
- abstrakte Objekte, an Funktionen übergeben 272, 273
 - AmbiguousBinding Programm 254, 255
 - definieren 255
 - Destruktoren, virtual 261, 262
 - Elementfunktionen, identisch deklarieren 259, 260
 - Elementfunktionen, überschreiben 252, 253
 - Konstruktoren, virtual 261
 - objektorientierte Programmierung 256, 257
 - virtual Schlüsselwort 257
 - virtuelle Funktionen 260, 261
 - virtuelle Funktionen, Zeiger 261
- Postfix-Operator x++ 316
- Präfix-Operator ++x 316
- Präprozessor 298, 299
 - #constants, Code zum Definieren 299, 300
 - #define-Direktive 299 - 302
 - #else-Zweig 302
 - #if-Anweisung 302
 - #if-Direktive 302, 303
 - #ifdef-Direktive 303 - 305
 - #include Direktive 298, 299
 - #include Kommando 298, 299
 - #macros 300 - 302
 - DEBUG-Parameter 305
 - dumpState()-Funktion 304, 305
 - Einschlusskontrolle, #ifdef Direktive 303, 304
 - Funktionen, Inline-Versionen funktionieren nicht 305
 - MyClass, myclass.h-Datei 303, 304
- printf()-Funktion 117
- private Schlüsselwort, Zugriffskontrolle auf Objekte 216
- Probleme, reproduzieren 95
- Program Reset-Kommando 169
- Program Reset, Debug-Operationen 174
- Programme
 - abstürzende concatString()-Funktion 173
 - aktuelles Verzeichnis 295
 - Anweisungen 32
 - Anwendungs-Code, aufteilen 291 - 293
 - Argumente 150 - 154
 - Ausdrücke 33, 34, 54
 - Average, mathematische Operatoren 53
 - binäre Zahlen 62, 63
 - ConstructElements, Ausgabe mit Destruktor 213
 - Conversion.cpp 11, 23
 - Debuggen 169, 171 - 173
 - Deklarationen 32, 33
 - dezimale Zahlen 42 - 46
 - DisplayString 112, 113
 - Division-vor-Addition Algorithmus 44
 - Eingabe/Ausgabe-Anweisungen 33
 - Einzelschritte 172, 173
 - elementarer Programmrahmen 30
 - erzeugen 9, 10, 11, 14, 18
 - EXE 10
 - gemischte Ausdrücke 50
 - gleich 39
 - Gleitkomma-Zahlen 45
 - GNU C++ 18, 20 - 26
 - Gold-Stern 92
 - Groß- und Kleinschreibung unterscheiden 32
 - h-Dateien 295
 - hexadezimale Zahlen 63, 66
 - include-Anweisungen 31
 - Integer 42, 44 - 46
 - Kapselung 288, 289
 - Kommentare definieren 31
 - kurze Schaltkreise 62
 - Leerraum 32
 - Masken 67
 - menschliches Programm 4
 - Module 288, 289
 - MS-DOS Backslash (\) 49
 - platte Reifen wechseln 4
 - Projektdateien 293 - 295
 - Rahmen 32
 - SeparatedClass.cpp-Datei 288, 289, 291, 292
 - Student 246 - 248
 - StudentID CD-ROM 227
 - teilen 288 - 290
 - Uneindeutigkeit vermeiden 34
 - Windows mit GNU C++ entwickeln 27

Programme, Code

- Account 267 - 270
- AmbiguousBinding 254, 255
- AmbiguousVererbung 280, 281, 282
- ArrayDemo 108 - 110
- BranchDemo
- BreakDemo
- CashAccount 271;272
- CharDisplay 111, 112
- ClassData 124, 125
- Concatenate 114, 115, 171
- ConcatenatePtr 146, 147
- ConstructElements 211, 212
- Conversion 30
- CopyStudent 232, 233
- DefaultStudentID 223 - 225
- DemoAssign 327 - 329
- EarlyBinding 253
- Error Program 92 - 94, 99, 100
- FactorialException 350, 351
- FalseStudentID 226
- ForDemo 74, ForDemo 75
- FunktionDemo 81 - 83
- GSInherit 243 - 245
- LateBinding 257 - 259
- Layout 128, 129
- LayoutError 132, 133
- LinkedListData 165 - 167
- MultipleVirtual 282 - 284
- NamedStudent 221, 222
- NestedDemo 78, 79
- ParallelData 120 - 122
- PassObjektPtr 157 - 159
- ProtectedElements 216
- SeparatedClass 288 - 290
- SleeperSofa 277, 278
- SquareDemo 85 - 87
- Student 246 - 248, 290
- Student Klasse 296
- USDollar 319, 320, 338 - 340
- USDollarAdd 313 - 315
- USDollarCast 321 - 323
- VirtualInserter 344 - 346
- WhileDemo 71, 72

Programme, Operatoren 34

- arithmetische 53, 54
- binäre 55, 56

- bitweise 58, 63 - 67
- Dekrement 56, 57
- einfache logische 58, 60 - 62
- Enzelbit 63
- Inkrement 56, 57
- logische 58
- unäre 55 - 58
- Vorrang 55, 56
- Zuweisung 34, 57, 58

Programme, Variablen 33, 34, 41

- char 47
- double 47
- Grenzen von Gleitkomma 46
- int 42 - 45
- Konstanten 48
- logische 62
- long 47
- mischen 50, 51
- Namen geben 33, 51
- Sonderzeichen 48, 49
- Typen 46 - 49
- Zeichenketten 47

Programmierung

- Computerprozessoren 4
- definieren 3, 4
- menschliches Programm 4
- Sprachen, C++ 8

Projektdateien 293

- erzeugen in GNU C++ 295
- erzeugen in Visual C++ 294, 295

Projekteinstellung, Argumente übergeben 154

- protected-Funktionen, überladener Zuweisungsoperator 330, 331

- protected keyword, Zugriffskontrolle für Objekte 214 - 217

ProtectedElements Programm-Code 216

Prototypen

- Funktionen 89, 90
- iostream.h Datei 333

Prozentzeichen (%) 117

Prozessoren

- Computer 4
- menschliches Programm 4
- platten Reifen wechseln 4
- Programmausgabe 16

414 Index

public Schlüsselwort
 Vererbung 245
 Zugriffskontrolle für Objekte 214
 public Schlüsselwort 123, 192, 193

Q

Quellcode-Dateien,
 Quellcode-Dateien, cpp 12
 Quelldateien, Module 89, 90

R

Rahmen für C++-Programme 30, 32
 rationalize() Funktion 316, 317
 README.1ST Datei 19
 Rechts-Shift (>) Operator 311, 312
 Referenzen, Zeiger 158, 159
 Reihenfolge der Konstruktion von Objekten 228 - 231
 rein virtuelle Funktionen 274
 Features in konkreten Unterklassen implementieren 273
 Syntax 267, 268
 überladen 272
 Zweck bitweiser Operatoren 66, 67
 Release-Modus 97
 remove()-Funktion 162, 163
 return-Anweisung, void-Funktion 85
 rhide
 Benutzer-Interface, GNU C++ Hilfe 27, 28
 Debugger 172
 Fenster 22
 Interface 21
 Oxff Exit-Code 96
 Programmargumente 154
 Rückgabety, Funktionen 85
 Runden von Integer 42 - 45
 Rundungsfehler 46

S

Savings-Klasse 263, 264
 abstrakte Unterklasse 271, 272
 konkrete Klasse, erzeugen aus abstrakter Klasse 269, 270
 schlaue Inserter 344 - 347
 Schleifen
 Bedingungen überprüfen 71
 do while 72
 endlos 73, 74
 for 74 - 76
 for modifizieren 98
 geschachtelte 78, 79
 Kontrollen 76, 77
 nLoopCount-Variable 72
 Variablen 77
 while 71 - 74, 115
 Schleifen-Kommandos oder Anweisungen 71 - 77
 Schleifen-Kommandos, geschachtelte Schleifen 78, 79
 Schlüsselwörter
 Klasse 123
 private, Zugriffskontrolle auf Objekte 216
 protected, Zugriffskontrolle auf Objekte 214, 215, 217
 public 123, 192, 193
 public, Vererbung 245
 public, Zugriffskontrolle auf Objekte 214
 struct 123, 192, 193
 try 351
 virtual 257, 258, 284
 void 84
 schreiben
 Elementfunktionen 198 - 200
 MYNAME Datei, Code zum Öffnen und Schreiben 334, 335
 Segmentverletzung 175
 Seiteneffekt 229
 selbstenthaltende Klassen 190, 191
 Semikolon (;) 12, 24, 32
 SeparatedClass Programm-Code 288 - 290
 SeparatedClass.cpp-Datei 288, 289, 291 - 293
 SeparatedFn.cpp-Datei 292
 Set Breakpoint-Kommando 169

- setfill(c)-Manipulatoren 342
- setprecision(c)-Manipulatoren 342
- setw()-Funktion 342
- setw(n)-Manipulatoren 342
- shift-Operatoren 346;347
- Slash
 - \ (Backslash) 49
 - // (doppelter) 31
- SleeperSofa
 - Klasse, Implementierung 282 - 284
 - Programm-Code 277, 278
 - Speicheranordnung 281, 282, 284
- Software, Free Software Foundation GNU C++ 18
- someFunktion() 88
- Sonderzeichen, Variablen 48, 49
- späte Bindung 255
- Speicher
 - Adressen 128, 129
 - double Variable 128
 - float Variable 128
 - Heap 135 - 138, 234, 316, 317
 - int Variable 128
 - long Variable 128
 - SleeperSofa, Speicheranordnung 281, 282, 284
- Speicher als Kommando (Datei Menü) 22, 23
- spitze Klammern < > 295, 298, 299
- Sprachen
 - Maschine 23
 - Programmieren in C++ 8
- square()-Funktion 85 - 87
- SquareDemo Programm-Code 85 - 87
- static-Datenelemente
 - Klassen 217, 218
 - Syntax zur Deklaration 217, 218
- static-Objekte, konstruieren 229
- Step In-Kommando 169, 174
- Step Over-Kommando 169, 172, 173
- Stream I/O
 - Ausgabe Insertter 343, 344
 - cin Eingabe-Objekt 332
 - cout Ausgabe-Objekt 332, 333
 - display()-Elementfunktion (virtual), schlaue Insertter 344
 - display() Funktion, Code mit Manipulatoren 341
 - ends-Konstante, Insertter-Kommando 338
 - fstream Unterklassen 334 - 337
 - fstream.h-Datei 334
 - ifstream-Klasse, Dateieingabe 336
 - inp> Extraktor-Anweisung 337, 338
 - Insertter 343 - 347
 - ios-Klasse, Konstanten zum Öffnen einer Datei 334
 - ios::out, definiert 334
 - iostream.h-Datei 333
 - istream-Klasse, definiert 337
 - Klassen, ofstream, Konstruktoren 333
 - Links-shift-Operator 346; 347
 - Manipulatoren 341 - 343
 - MYNAME-Datei, Code zum Öffnen und Schreiben 334, 335
 - nicht null Fehler-Flag 336
 - Objekte 333
 - ofstream-Klasse, Destruktor 336
 - ofstream-Klasse, Konstruktoren 334 - 335
 - Operator<<() (Insertter) 333
 - Operator<<() 343;344
 - Operator>() (Extraktor) 332
 - ostream-Klasse, definieren 337
 - out-Objekt 337, 338
 - parseString()-Funktion 337, 338
 - Prototypen, iostream.h-Datei 333
 - schlaue Insertter 344 - 347
 - setw()-Funktion 342
 - Shift-Operatoren 346, 347
 - Stream-Ausgabe, Ausnahme-basierte Fehlerbehandlung 335
 - strstream.h-Datei, definieren strstream-Unterklassen 337
 - strstream-Klasse, definiert 337
 - strstream-Unterklassen 337 - 341
 - USDollar::output()-Code 340;341
 - VirtualInsertter Programm-Code 344 - 346
 - width-Parameter 342
 - width()-Funktion 342
 - Zeichenketten, behandeln 338 - 341
- strstream.h-Datei, definiert strstream-Unterklassen 337
- strstream
 - Klasse, definieren 337
 - Unterklassen 337 - 341

416 Index

struct-Schlüsselwort 123, 192 - 194
 Struktur, Klassen 192, 193
 Student-Klasse
 Elementfunktionen, Code zur Deklaration außerhalb der Klasse 296
 Konstruktoren 222, 223
 Student-Konstruktor, Klasse StudentID-Element-Objekt 223 - 225
 Student Programm-Code 246 - 248, 290
 student.cpp-Datei 199
 StudentID-Programm CD-ROM 227
 Subtraktions-Operatoren 53
 sumArray()-Funktion 110
 sumSequence()-Funktion, aufrufen oder definieren 83
 Superklassen 189
 switch
 Anweisung 79
 Kommando, break-Anweisungen 79
 switch(), Kontrollkommando 151
 Syntax
 Datenelement deklarieren 228
 Objekteigenschaften, zugreifen 123
 rein virtuelle Funktionen 267, 268
 statische Datenelemente, deklarieren 217, 218

T

Tastaturkürzel
 Alt+4 179, 180
 Alt+F4 26
 Ctrl+F5 14
 Ctrl+F9 26
 Tasten 17, 27
 temporäre Objekte 317
 terminieren
 null, Zeichenketten 177
 Zeichenketten 115
 testen
 bitweise Operatoren, Code 65, 66
 Programm debuggen 171, 172
 Textdateien, erzeugen 9 - 11

tiefe Kopie gegen flache Kopie 233, 234
 Tilde (~) 132
 Ton, Visual C++ 12
 ToStringWStreams-Programm CD-ROM 340, 341
 try-Schlüsselwort 351
 try-Block, mit catch-Phrasen verbinden 357
 Typen von
 Fehlern 92, 93
 Variablen 46 - 49
 Variablen, int Begrenzungen 42 - 45
 Zeiger 132, 133

U

überladen
 Elementfunktionen 203 - 205
 Elementfunktionen in Unterklassen 252
 Funktionen 222, 223
 Konstruktoren 221, 222
 rein virtuelle Funktionen 272
 überladen des Zuweisungsoperator 325, 326
 Code 327 - 329
 protected-Funktionen 330, 331
 Überladen von Operatoren 311, 313, 315
 binäre Operatoren, Verändern von Werten 318
 Cast-Operator 321 - 323
 Elementfunktionen, Operatoren 321
 Funktionen, nicht Element 320
 Funktionen Operatoren, ihre Beziehung 312
 Funktionen, rationalize() 316
 Heap-Speicher 317
 Klassen, Freunde 315
 Lesbarkeit, verbessern 311, 312
 Links-Shift (<<)-Operator 311, 312
 Nichtelement-Funktionen 320, 321
 Objekte, konvertieren 323
 Objekte, temporär 317
 operator+() 315 - 318
 operator++() Funktion 315
 Postfix-Operator x++ 316
 Präfix-Operator ++x 316
 rationalize()-Funktion 316
 Rechts-Shift (>)-Operator 311, 312
 temporäre Objekte 317

unärer operator++(), Argumente 317, 318
 Verwendung von 311 - 313
 Zuweisungs (=)-Operator 311, 312
 überladene Operatoren
 operator<<() 333
 operator>() 333
 Überschreiben von Elementfunktionen 253
 Umlenkungssymbol (<) 153
 Umlenkungssymbol (>) 153
 unäre Operatorer 55 - 58
 unärer Operator++(), Argumente 317, 318
 unäres * Zeichen 130
 Und-Zeichen (&) 125
 Uneindeitigkeit, Mehrfachvererbung 278, 279
 Uneindeutigkeit vermeiden 34
 Ungleichheits-Operator (!=) 60
 universelle Zeiger 161
 Unterklassen
 abstrakte 271, 272
 Elementfunktionen überladen 252
 fstream 334 - 337
 konkrete, rein virtuelle Funktionen implementiert 273
 konstruieren 246 - 249
 strstream 337 - 341
 Unterklassen 189
 Unterscheidung Groß- und Kleinschreibung 9, 32
 Unterstrich und einfache Hochkommata ('_') 34
 USDollar-Programm
 Code 319, 320
 Code zur Konvertierung in Zeichenkette für Ausgabe 338 - 340
 Insertter, Code für Ausgabe 343, 344
 USDollar::output(), ostream Klasse, Code 340;341
 USDollarAdd Programm-Code 313 - 315
 USDollarCast Programm-Code 321 - 323
 User Screen-Kommando (Windows Menü) 27

V

Variablen 41
 ansehen 175 - 179
 auto 90
 benennen 33, 51
 C++-Programme 32, 33
 char 47
 dezimale Zahlen 42 - 46
 double 47
 double, Speicher 128
 float, Speicher 128
 gemischte Ausdrücke 50
 Gleitkomma Begrenzungen 46
 int-Variablentyp, Begrenzungen 42 - 45
 int, Speicher 128
 Konstanten 48
 logische 62
 lokale ansehen 179, 180
 long 47
 long, Speicher 128
 mischen 50, 51
 modifizieren 176 - 179
 nGlobal 90
 nLocal 90
 nLoopCount Variable 72
 nStatic 90
 Schleifen 77
 Sonderzeichen 48, 49
 speichern 90
 Typen 46 - 49
 von Zeiger 129, 130, 132
 Werte 99
 Zeichenketten 47
 Variablen speichern 90
 Variablen Window 179, 180
 Vererbung 252
 Basisklassen, konstruieren 248
 Basisklassen, vernichten 248
 Faktorisieren 263 - 266
 GraduateStudent-Objekt HAS_A Beziehung 249, 250
 implementieren 242 - 245
 Polymorphie 255
 public-Schlüsselwort 245
 Unterklassen, konstruieren 246 - 249
 virtual 279 - 285
 Vorteile 241

418 Index

- Vererbung, mehrfache 276 - 278
 - Nachteile 285, 286
 - SleeperSofa, Speicheranordnung 281, 282, 284
 - Uneindeutigkeit 278, 279
 - virtuelle Vererbung 279 - 285
 - verkettete Liste 160, 161, 163
 - Eigenschaften 164
 - Elemente am Kopf einfügen 161
 - Elemente ans Ende anfügen 162, 163
 - Anwendung, debugging 176
 - verkettete Zeiger, nicht initialisiert 167
 - Vernichten von Objekten 206 - 213
 - vertrauenswürdige Funktionen, definiert 217
 - Verzeichnis, aktuelles 295
 - Verzeichnisse, BIN 19
 - Verzweigungs-Kommando oder Anweisung 69 - 71
 - View Menü-Kommandos
 - Arbeitsbereich 294
 - Debug-Fenster 179, 180
 - View User Screen-Kommando 169
 - View Variable-Kommando 169
 - virtuelle Vererbung 279 - 285
 - virtual Destruktoren 261, 262
 - virtual display()-Elementfunktion, schlaue Inserter 344
 - virtual Funktionen 259, 260
 - inline 261
 - rein, Syntax 267, 268
 - Zeiger 261
 - virtual Konstruktoren 261
 - virtual Schlüsselwort 257, 258, 284
 - VirtuallInserter Programm-Code 344 - 346
 - Visual C++ 8
 - Arbeitsbereich einrichten 11
 - Debugger 178 - 180
 - Fehlermeldungen 14, 95, 96
 - Hilfe 16, 17
 - Installation 8
 - nNums 96
 - Projektdateien anlegen 294, 295
 - void-Funktion, return-Anweisung 85
 - void-Konstruktoren 209, 222, 223
 - void-Schlüsselwort 84
 - void strcat(target source)-Funktion 116
 - Vorrang von Operatoren 55, 56
 - Vorteile von Funktionen 84, 85
- ## W
- Web-Sites, Delorie 19
 - Werte
 - Argumente, Default-Werte 223
 - filebuf::openprot 334
 - filebuf::sh_none 334
 - filebuf::sh_read 334
 - filebuf::sh_write 334
 - von Variablen 100
 - while-Anweisung, breakpoint Kommando 175
 - WhileDemo Programm-Code 71, 72
 - while-Schleife 71, 72, 74, 115
 - wide characters 117
 - width-Parameter 342
 - width()-Funktion 342
 - Windows-Menü, Kommandos, User Screen 27
 - Windows-Programme, entwickeln mit GNU C++ 27
 - WRITE-Anweisung 92, 93
 - WRITE-Anweisung, Zeigerfehler 170
 - WWW (World Wide Web), GNU C++ installieren von 19, 21
- ## X
- XOR
 - bitweiser Operator (~) 64
 - Operator 64
- ## Z
- Zahlen
 - binäre 62, 63

- dezimale 42 - 46
- Division-vor-Addition Algorithmus 44
- Gleitkoma 45
- hexadezimal 63, 66
- Integer, gebrochene Werte 42
- Integer, Abschneiden 44, 45
- Integer, Lösen des Abschneideproblems 45, 46
- Integer, Runden 42 - 45
- mitteln, Code 43
- Rundungsfehler 46
- Zeichen 101
 - Arrays von 111 - 113, 145 - 147
 - spezielle, Variablen 48, 49
 - Variablen, Namensgebung 51
 - wide 117
 - Zeichen-Begrenzung, überprüfen 121, 122
 - Zeichenketten, Arrays 151, 152
- Zeichen-Grenzen überprüfen 121, 122
- Zeichenketten
 - behandeln 338 - 341
 - benennen 113
 - definieren 113
 - Funktionen zur Manipulation 116, 117
 - initialisieren 113
 - manipulieren 114 - 117
 - mit null terminieren 177
 - terminieren 115
 - wide characters 117
 - Ziel 177
- Zeichenketten I/O, USDollar Programm-Code, Konvertierung in Zeichenkette für Ausgabe 338 - 340
- Zeichenketten, Variablen 47
- Zeichen-Limit, überprüfen 121, 122
- Zeiger 127, 140
 - Argumente über Projekteinstellung übergeben 154
 - Array-basierte Manipulation von Zeichenketten 145, 146
 - Arrays 140 - 147
 - Arrays, Datenstruktur 159, 160
 - Arrays, Unterschiede zwischen 148 - 150
 - auf Objekte 156 - 159
 - Container FIFO (first-in-first-out) 168
 - double Variable, Speicher 128
 - Elementfunktionen, aufrufen 201, 202
 - Fehler, Schreibweise 170
 - float-Variable, Speicher 128
 - Geltungsbereich der Variable 135 - 138
 - Heap-Speicher 135 - 138, 159
 - initialisieren 161
 - int-Variable, Speicher 128
 - linkbare Klassen, deklarieren 160
 - long-Variable, Speicher 128
 - Objekte übergeben 157 - 159
 - Offsets und Arrays 142
 - Operationen auf Typen 148
 - Operationen definiert auf Typen 140, 141
 - Operatoren 129
 - Projekteinstellungen, Argumente übergeben 154
 - Referenzen 158, 159
 - Speicheradressen 128, 129
 - Typ von 132, 133
 - überwachen 176
 - universaler 161
 - Variablen 129 - 131
 - Variablen p (Präfix) 130
 - Variablen, Argumente an Funktionen übergeben 133 - 135
 - verkettet, nicht initialisiert 167
 - virtuelle Funktionen 261
 - Zeichenarrays 145 - 147
- Zeiger + Offset-Operation 140, 141
- Zeiger-Offset-Operation 140, 141
- Zeiger überwachen 176
- Zeiger, verkettete Listen 160, 163
 - Eigenschaften 164
 - Elemente am Ende-Zeiger anfügen 162, 163
 - Objekte am Kopf-Zeiger einfügen 162
- Zeiger2 Zeiger1-Operation 140, 141
- Zeilen einrücken 9
- Zielzeichenkette 177
- Ziffer, definiert 62
- zip-Dateien 19
- ZIP-Picker 19
- Zugriff auf
 - Eigenschaften von Objekten, Syntax 123
 - Elemente mit Elementfunktion 201 - 203
 - nArray 108
- Zugriff auf Objekte kontrollieren 214 - 218
- Zugriffsfunktion, definiert 216, 217

420 Index

Zugriffskontrolle, Objekte 214, 217, 218

private Schlüsselwort 216

protected Schlüsselwort 214 - 217

public Schlüsselwort 214

Zuweisungs (=)-Operator 311, 312

Zuweisungsoperator 34, 57, 58

Code zum Überladen 327 - 329

Elementfunktion operator=() 330

Name& (return type), operator=() 329, 330

operator=(), Default-Definition 325, 326

operator=(), Elementfunktion 330

operator=(), Name& (return type) 329, 330

überladen 325 - 330

überladen mit geschützten Funktionen 330, 331

Vergleich mit Kopierkonstruktor 326, 327

Zweige #else 302

GNU General Public License

Die folgende deutsche Übersetzung wurde im Auftrag der SuSE GmbH [suse@suse.de] von Katja Lachmann Übersetzungen [na194@fim.uni-erlangen.de] erstellt und von Peter Gerwinski [peter.gerwinski@uni-essen.de] (31. Oktober 1996) modifiziert. Diese Übersetzung wird mit der Absicht angeboten, das Verständnis der GNU General Public License (GNU-GPL) zu erleichtern. Es handelt sich jedoch nicht um eine offizielle oder im rechtlichen Sinne anerkannte Übersetzung.

Die Free Software Foundation (FSF) ist nicht der Herausgeber dieser Übersetzung, und sie hat diese Übersetzung auch nicht als rechtskräftigen Ersatz für die Original-GNU GPL anerkannt. Da die Übersetzung nicht sorgfältig von Anwälten überprüft wurde, können die Übersetzer nicht garantieren, dass die Übersetzung die rechtlichen Aussagen der GNU GPL exakt wiedergibt. Wenn Sie sicher gehen wollen, dass von Ihnen geplante Aktivitäten im Sinne der GNU GPL gestattet sind, halten Sie sich bitte an die englischsprachige Originalversion. Die Free Software Foundation möchte Sie darum bitten, diese Übersetzung nicht als offizielle Lizenzbedingungen für von Ihnen geschriebene Programme zu verwenden. Bitte benutzen Sie hierfür stattdessen die von der Free Software Foundation herausgegebene englischsprachige Originalversion.

GNU General Public License.

Deutsche Übersetzung der Version 2, Juni 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc. 675 Mass Ave, Cambridge, MA 02139, USA

Jeder hat das Recht, diese Lizenzurkunde zu vervielfältigen und unveränderte Kopien zu verbreiten; Änderungen sind jedoch nicht gestattet. Diese Übersetzung ist kein rechtskräftiger Ersatz für die englischsprachige Originalversion!

Vorwort

Die meisten Softwarelizenzen sind daraufhin entworfen worden, Ihnen die Freiheit zu nehmen, Software weiterzugeben und zu verändern. Im Gegensatz dazu soll Ihnen die GNU General Public License, die allgemeine öffentliche GNU-Lizenz, eben diese Freiheit garantieren. Sie soll sicherstellen, dass die Software für alle Benutzer frei ist. Diese Lizenz gilt für den Großteil der von der Free Software Foundation herausgegebenen Software und für alle anderen Programme, deren Autoren ihr Werk dieser Lizenz unterstellt haben. Auch Sie können diese Möglichkeit der Lizenzierung für Ihre Programme anwenden. (Ein anderer Teil der Software der Free Software Foundation unterliegt statt dessen der GNU Library General Public License, der allgemeinen öffentlichen GNU-Lizenz für Bibliotheken.)

422 GNU General Public License

Die Bezeichnung »freie« Software bezieht sich auf Freiheit, nicht auf den Preis. Unsere Lizenzen sollen Ihnen die Freiheit garantieren, Kopien freier Software zu verbreiten (und etwas für diesen Service zu berechnen, wenn Sie möchten), die Möglichkeit, die Software im Quelltext zu erhalten oder den Quelltext auf Wunsch zu bekommen. Die Lizenzen sollen garantieren, dass Sie die Software ändern oder Teile davon in neuen freien Programmen verwenden dürfen – und dass Sie wissen, dass Sie dies alles tun dürfen.

Um Ihre Rechte zu schützen, müssen wir Einschränkungen machen, die es jedem verbietet, Ihnen diese Rechte zu verweigern oder Sie aufzufordern, auf diese Rechte zu verzichten. Aus diesen Einschränkungen folgen bestimmte Verantwortlichkeiten für Sie, wenn Sie Kopien der Software verbreiten oder sie verändern.

Beispielsweise müssen Sie den Empfängern alle Rechte gewähren, die Sie selbst haben, wenn Sie – kostenlos oder gegen Bezahlung – Kopien eines solchen Programms verbreiten. Sie müssen sicherstellen, dass auch sie den Quelltext erhalten bzw. erhalten können. Und Sie müssen ihnen diese Bedingungen zeigen, damit sie ihre Rechte kennen.

Wir schützen Ihre Rechte in zwei Schritten: (1) Wir stellen die Software unter ein Urheberrecht (Copyright), und (2) wir bieten Ihnen diese Lizenz an, die Ihnen das Recht gibt, die Software zu vervielfältigen, zu verbreiten und/oder zu verändern.

Um die Autoren und uns zu schützen, wollen wir darüber hinaus sicherstellen, dass jeder erfährt, dass für diese freie Software keinerlei Garantie besteht. Wenn die Software von jemand anderem modifiziert und weitergegeben wird, möchten wir, dass die Empfänger wissen, dass sie nicht das Original erhalten haben, damit von anderen verursachte Probleme nicht den Ruf des ursprünglichen Autors schädigen.

Schließlich und endlich ist jedes freie Programm permanent durch Software-Patente bedroht. Wir möchten die Gefahr ausschließen, dass Distributoren eines freien Programms individuell Patente lizenzieren – mit dem Ergebnis, dass das Programm proprietär würde. Um dies zu verhindern, haben wir klargestellt, dass jedes Patent entweder für freie Benutzung durch jedermann lizenziert werden muss oder überhaupt nicht lizenziert werden darf.

Es folgen die genauen Bedingungen für die Vervielfältigung, Verbreitung und Bearbeitung:

Bedingungen für die Vervielfältigung, Verbreitung, und Bearbeitung.

Paragraph 0

Diese Lizenz gilt für jedes Programm und jedes andere Werk, in dem ein entsprechender Vermerk des Copyright-Inhabers darauf hinweist, dass das Werk unter den Bestimmungen dieser General Public License verbreitet werden darf. Im Folgenden wird jedes derartige Programm oder Werk als »das Programm« bezeichnet; die Formulierung »auf dem Programm basierendes Werk« bezeichnet das Programm sowie jegliche Bearbeitung des Programms im urheberrechtlichen Sinne, also ein Werk, welches das Programm, auch auszugsweise, sei es unverändert oder verändert und/oder in eine andere Sprache übersetzt, enthält. (Im Folgenden wird die Übersetzung ohne Einschränkung als »Bearbeitung« eingestuft.) Jeder Lizenznehmer wird im Folgenden als »Sie« angesprochen.

Andere Handlungen als Vervielfältigung, Verbreitung und Bearbeitung werden von dieser Lizenz nicht berührt; sie fallen nicht in ihren Anwendungsbereich. Der Vorgang der Ausführung des Programms wird nicht eingeschränkt, und die Ausgaben des Programms unterliegen dieser Lizenz nur, wenn der Inhalt ein auf dem Programm basierendes Werk darstellt (unabhängig davon, dass die Ausgabe durch die Ausführung des Programmes erfolgte). Ob dies zutrifft, hängt von den Funktionen des Programms ab.

Paragraph 1

Sie dürfen auf beliebigen Medien unveränderte Kopien des Quelltextes des Programms, wie Sie ihn erhalten haben, anfertigen und verbreiten. Voraussetzung hierfür ist, dass Sie mit jeder Kopie einen entsprechenden Copyright-Vermerk sowie einen Haftungsausschluss veröffentlichen, alle Vermerke, die sich auf diese Lizenz und das Fehlen einer Garantie beziehen, unverändert lassen und des Weiteren allen anderen Empfängern des Programms zusammen mit dem Programm eine Kopie dieser Lizenz zukommen lassen.

Sie dürfen für den eigentlichen Kopiervorgang eine Gebühr verlangen. Wenn Sie es wünschen, dürfen Sie auch gegen Entgelt eine Garantie für das Programm anbieten.

Paragraph 2

Sie dürfen Ihre Kopie(n) des Programms oder eines Teils davon verändern, wodurch ein auf dem Programm basierendes Werk entsteht; Sie dürfen derartige Bearbeitungen unter den Bestimmungen von Paragraph 1 vervielfältigen und verbreiten, vorausgesetzt, dass zusätzlich alle folgenden Bedingungen erfüllt werden:

- (a) Sie müssen die veränderten Dateien mit einem auffälligen Vermerk versehen, der auf die von Ihnen vorgenommene Modifizierung und das Datum jeder Änderung hinweist.
- (b) Sie müssen dafür sorgen, dass jede von Ihnen verbreitete oder veröffentlichte Arbeit, die ganz oder teilweise von dem Programm oder Teilen davon abgeleitet ist, Dritten gegenüber als Ganzes unter den Bedingungen dieser Lizenz ohne Lizenzgebühren zur Verfügung gestellt wird.
- (c) Wenn das veränderte Programm normalerweise bei der Ausführung interaktiv Kommandos einliest, müssen Sie dafür sorgen, dass es, wenn es auf dem üblichsten Wege für solche interaktive Nutzung gestartet wird, eine Meldung ausgibt oder ausdruckt, die einen geeigneten Copyright-Vermerk enthält sowie einen Hinweis, dass es keine Gewährleistung gibt (oder anderenfalls, dass Sie Garantie leisten), und dass die Benutzer das Programm unter diesen Bedingungen weiter verbreiten dürfen. Auch muss der Benutzer darauf hingewiesen werden, wie eine Kopie dieser Lizenz ansehen kann. (Ausnahme: Wenn das Programm selbst interaktiv arbeitet, aber normalerweise keine derartige Meldung ausgibt, muss Ihr auf dem Programm basierendes Werk auch keine solche Meldung ausgeben).

Diese Anforderungen betreffen das veränderte Werk als Ganzes. Wenn identifizierbare Abschnitte des Werkes nicht von dem Programm abgeleitet sind und vernünftigerweise selbst als unabhängige und eigenständige Werke betrachtet werden können, dann erstrecken sich diese Lizenz und ihre Bedingungen nicht auf diese Abschnitte, wenn sie als eigenständige Werke verbreitet werden. Wenn Sie jedoch dieselben Abschnitte als Teil eines Ganzen verbreiten, das ein auf dem Programm basierendes Werk darstellt, dann muss die Verbreitung des Ganzen nach den Bedingungen dieser Lizenz erfolgen, deren Bedingungen für weitere Lizenznehmer somit auf die Gesamtheit ausgedehnt werden – und damit auf jeden einzelnen Teil, unabhängig vom jeweiligen Autor.

424 GNU General Public License

Somit ist es nicht die Absicht dieses Abschnittes, Rechte für Werke in Anspruch zu nehmen oder zu beschneiden, die komplett von Ihnen geschrieben wurden; vielmehr ist es die Absicht, die Rechte zur Kontrolle der Verbreitung von Werken, die auf dem Programm basieren oder unter seiner auszugsweisen Verwendung zusammengestellt worden sind, auszuüben.

Ferner bringt ein einfaches Zusammenstellen eines anderen Werkes, das nicht auf dem Programm basiert, zusammen mit dem Programm oder einem auf dem Programm basierenden Werk auf ein- und demselben Speicher- oder Vertriebsmedium nicht in den Anwendungsbereich dieser Lizenz.

Paragraph 3

Sie dürfen das Programm (oder ein darauf basierendes Werk gemäß Paragraph 2) als Objektcode oder in ausführbarer Form unter den Bedingungen von Paragraph 1 und 2 vervielfältigen und verbreiten – vorausgesetzt, dass Sie außerdem eine der folgenden Leistungen erbringen:

- (a) Liefern Sie das Programm zusammen mit dem vollständigen zugehörigen maschinenlesbaren Quelltext auf einem für den Datenaustausch üblichen Medium aus, wobei die Verteilung unter den Bedingungen der Paragraphen 1 und 2 erfolgen muss. Oder:
- (b) Liefern Sie das Programm zusammen mit einem mindestens drei Jahre lang gültigen schriftlichen Angebot aus, jedem Dritten eine vollständige maschinenlesbare Kopie des Quelltextes zur Verfügung zu stellen – zu nicht höheren Kosten als denen, die durch den physischen Kopiervorgang anfallen –, wobei der Quelltext unter den Bedingungen der Paragraphen 1 und 2 auf einem für den Datenaustausch üblichen Medium weitergegeben wird. Oder:
- (c) Liefern Sie das Programm zusammen mit dem schriftlichen Angebot der Zurverfügungstellung des Quelltextes aus, das Sie selbst erhalten haben. (Diese Alternative ist nur für nicht-kommerzielle Verbreitung zulässig und nur, wenn Sie das Programm als Objektcode oder in ausführbarer Form mit einem entsprechenden Angebot gemäß Absatz b erhalten haben.)

Unter dem Quelltext eines Werkes wird diejenige Form des Werkes verstanden, die für Bearbeitungen vorzugsweise verwendet wird. Für ein ausführbares Programm bedeutet »der komplette Quelltext«: Der Quelltext aller im Programm enthaltenen Module einschließlich aller zugehörigen Modulschnittstellen-Definitionsdateien sowie der zur Kompilation und Installation verwendeten Skripte. Als besondere Ausnahme jedoch braucht der verteilte Quelltext nichts von dem zu enthalten, was üblicherweise (entweder als Quelltext oder in binärer Form) zusammen mit den Hauptkomponenten des Betriebssystems (Kernel, Compiler usw.) geliefert wird, unter dem das Programm läuft – es sei denn, diese Komponente selbst gehört zum ausführbaren Programm.

Wenn die Verbreitung eines ausführbaren Programms oder von Objektcode dadurch erfolgt, dass der Kopierzugriff auf eine dafür vorgesehene Stelle gewährt wird, so gilt die Gewährung eines gleichwertigen Zugriffs auf den Quelltext als Verbreitung des Quelltextes, auch wenn Dritte nicht dazu gezwungen sind, den Quelltext zusammen mit dem Objektcode zu kopieren.

Paragraph 4

Sie dürfen das Programm nicht vervielfältigen, verändern, weiter lizenzieren oder verbreiten, sofern es nicht durch diese Lizenz ausdrücklich gestattet ist. Jeder anderweitige Versuch der Vervielfältigung, Modifizierung, Weiterlizenzierung und Verbreitung ist nichtig und beendet automatisch Ihre Rechte unter dieser Lizenz. Jedoch werden die Lizenzen Dritter, die von Ihnen Kopien oder Rechte unter dieser Lizenz erhalten haben, nicht beendet, solange diese die Lizenz voll anerkennen und befolgen.

Paragraph 5

Sie sind nicht verpflichtet, diese Lizenz anzunehmen, da Sie sie nicht unterzeichnet haben. Jedoch gibt Ihnen nichts anderes die Erlaubnis, das Programm oder von ihm abgeleitete Werke zu verändern oder zu verbreiten. Diese Handlungen sind gesetzlich verboten, wenn Sie diese Lizenz nicht anerkennen. Indem Sie das Programm (oder ein darauf basierendes Werk) verändern oder verbreiten, erklären Sie Ihr Einverständnis mit dieser Lizenz und mit allen ihren Bedingungen bezüglich der Vervielfältigung, Verbreitung und Veränderung des Programms oder eines darauf basierenden Werks.

Paragraph 6

Jedes Mal, wenn Sie das Programm (oder ein auf dem Programm basierendes Werk) weitergeben, erhält der Empfänger automatisch vom ursprünglichen Lizenzgeber die Lizenz, das Programm entsprechend den hier festgelegten Bestimmungen zu vervielfältigen, zu verbreiten und zu verändern. Sie dürfen keine weiteren Einschränkungen der Durchsetzung der hierin zugestandenen Rechte des Empfängers vornehmen. Sie sind nicht dafür verantwortlich, die Einhaltung dieser Lizenz durch Dritte durchzusetzen.

Paragraph 7

Sollten Ihnen infolge eines Gerichtsurteils, des Vorwurfs einer Patentverletzung oder aus einem anderen Grunde (nicht auf Patentfragen begrenzt) Bedingungen (durch Gerichtsbeschluss, Vergleich oder anderweitig) auferlegt werden, die den Bedingungen dieser Lizenz widersprechen, so befreien Sie diese Umstände nicht von den Bestimmungen dieser Lizenz. Wenn es Ihnen nicht möglich ist, das Programm unter gleichzeitiger Beachtung der Bedingungen in dieser Lizenz und Ihrer anderweitigen Verpflichtungen zu verbreiten, dann dürfen Sie als Folge das Programm überhaupt nicht verbreiten. Wenn zum Beispiel ein Patent nicht die gebührenfreie Weiterverbreitung des Programms durch diejenigen erlaubt, die das Programm direkt oder indirekt von Ihnen erhalten haben, dann besteht der einzige Weg, sowohl das Patentrecht als auch diese Lizenz zu befolgen, darin, ganz auf die Verbreitung des Programms zu verzichten.

Sollte sich ein Teil dieses Paragraphen als ungültig oder unter bestimmten Umständen nicht durchsetzbar erweisen, so soll dieser Paragraph seinem Sinne nach angewandt werden; im übrigen soll dieser Paragraph als Ganzes gelten.

Zweck dieses Paragraphen ist nicht, Sie dazu zu bringen, irgendwelche Patente oder andere Eigentumsansprüche zu verletzen oder die Gültigkeit solcher Ansprüche zu bestreiten; dieser Paragraph hat einzig den Zweck, die Integrität des Vertriebssystems der freien Software zu schützen, das durch die Praxis öffentlicher Lizenzen verwirklicht wird. Viele Leute haben großzügige Beiträge zu dem großen Angebot der mit diesem System verbreiteten Software im Vertrauen auf die konsistente Anwendung dieses Systems geleistet; es liegt am Autor/Geber zu entscheiden, ob er die Software mittels irgendeines anderen Systems verbreiten will; ein Lizenznehmer hat auf diese Entscheidung keinen Einfluss.

Dieser Paragraph ist dazu gedacht, deutlich klarzustellen, was als Konsequenz aus dem Rest dieser Lizenz betrachtet wird.

426 GNU General Public License**Paragraph 8**

Wenn die Verbreitung und/oder die Benutzung des Programms in bestimmten Staaten entweder durch Patente oder durch urheberrechtlich geschützte Schnittstellen eingeschränkt ist, kann der Urheberrechtsinhaber, der das Programm unter diese Lizenz gestellt hat, eine explizite geografische Begrenzung der Verbreitung angeben, in der diese Staaten ausgeschlossen werden, sodass die Verbreitung nur innerhalb und zwischen den Staaten erlaubt ist, die nicht ausgeschlossen sind. In einem solchen Fall beinhaltet diese Lizenz die Beschränkung, als wäre sie in diesem Text niedergeschrieben.

Paragraph 9

Die Free Software Foundation kann von Zeit zu Zeit überarbeitete und/oder neue Versionen der General Public License veröffentlichen. Solche neuen Versionen werden vom Grundprinzip her der gegenwärtigen entsprechen, können aber im Detail abweichen, um neuen Problemen und Anforderungen gerecht zu werden.

Jede Version dieser Lizenz hat eine eindeutige Versionsnummer. Wenn in einem Programm angegeben wird, dass es dieser Lizenz in einer bestimmten Versionsnummer oder »jeder späteren Version« (»any later version«) unterliegt, so haben Sie die Wahl, entweder den Bestimmungen der genannten Version zu folgen oder denen jeder beliebigen späteren Version, die von der Free Software Foundation veröffentlicht wurde. Wenn das Programm keine Versionsnummer angibt, können Sie eine beliebige Version wählen, die je von der Free Software Foundation veröffentlicht wurde.

Paragraph 10

Wenn Sie den Wunsch haben, Teile des Programms in anderen freien Programmen zu verwenden, deren Bedingungen für die Verbreitung andere sind, schreiben Sie an den Autor, um ihn um die Erlaubnis zu bitten. Für Software, die unter dem Copyright der Free Software Foundation steht, schreiben Sie an die Free Software Foundation; wir machen zu diesem Zweck gelegentlich Ausnahmen. Unsere Entscheidung wird von den beiden Zielen geleitet werden, zum einen den freien Status aller von unserer freien Software abgeleiteten Werke zu erhalten und zum anderen das gemeinschaftliche Nutzen und Wiederverwenden von Software im Allgemeinen zu fördern.

Keine Gewährleistung**Paragraph 11**

Da das Programm ohne jegliche Kosten lizenziert wird, besteht keinerlei Gewährleistung für das Programm, soweit dies gesetzlich zulässig ist. Sofern nicht anderweitig schriftlich bestätigt, stellen die Copyright-Inhaber und/oder Dritte das Programm so zur Verfügung, »wie es ist«, ohne irgendeine Gewährleistung, weder ausdrücklich noch implizit, einschließlich – aber nicht begrenzt auf – Marktreife oder Verwendbarkeit für einen bestimmten Zweck. Das volle Risiko bezüglich Qualität und Leistungsfähigkeit des Programms liegt bei Ihnen. Sollte sich das Programm als fehlerhaft herausstellen, liegen die Kosten für notwendigen Service, Reparatur oder Korrektur bei Ihnen.

Paragraph 12

In keinem Fall, außer wenn durch geltendes Recht gefordert oder schriftlich zugesichert, ist irgendein Copyright-Inhaber oder irgendein Dritter, der das Programm wie oben erlaubt modifiziert oder verbreitet hat, Ihnen gegenüber für irgendwelche Schäden haftbar, einschließlich jeglicher allgemeiner oder spezieller Schäden, Schäden durch Seiteneffekte (Nebenwirkungen) oder Folgeschäden, die aus der Benutzung des Programms oder der Unbenutzbarkeit des Programms folgen (einschließlich – aber nicht beschränkt auf – Datenverluste, fehlerhafte Verarbeitung von Daten, Verluste, die von Ihnen oder anderen getragen werden müssen, oder dem Unvermögen des Programms, mit irgendeinem anderen Programm zusammenzuarbeiten), selbst wenn ein Copyright-Inhaber oder Dritter über die Möglichkeit solcher Schäden unterrichtet worden war.

