C++ Programmieren mit Stil

http://kickme.to/tiger/

Vorige Seite: _ **Eine Ebene höher:** _ **Nächste Seite:** 1 Vorbemerkungen

Inhalt

Vorige Seite: Inhalt **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** Was das Buch nicht

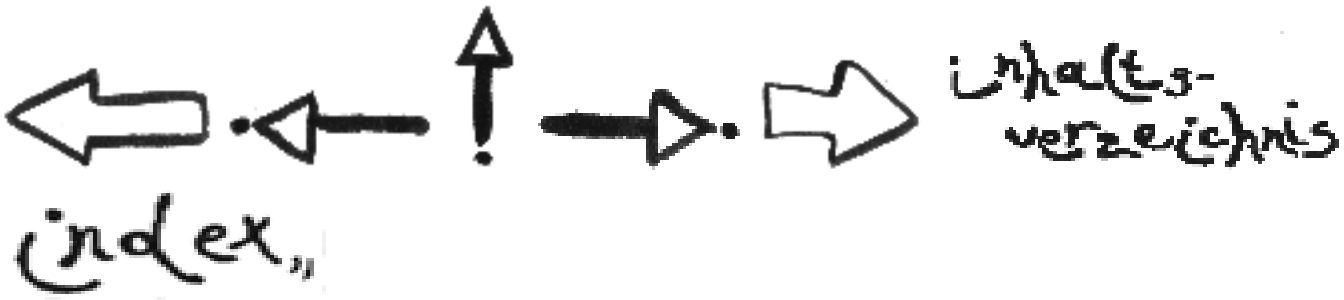
1 Vorbemerkungen

Das vorliegende Buch beschreibt die Programmiersprache C++ nach dem ANSI/ISO-Standard. Zum Zeitpunkt der Entstehung dieses Buchs (Sommer/Herbst 1996) ist der Standard allerdings noch nicht verabschiedet. Es gibt daher kein offizielles Dokument, das den endgültigen Standard beschreibt. Alle existierenden Dokumente sind entweder *Draft*, das heißt vorläufig, oder aber inoffzielle Beschreibungen.

Ein Buch über etwas schreiben zu wollen, das noch nicht existiert, wirft eine Reihe von Problemen auf: Die zur Verfügung stehenden Informationen sind meist vage und unsicher, es stehen noch keine Entwicklungssysteme zur Verfügung, die genau dem kommenden Sprachstandard entsprechen usw.

Die Bedeutung des Sprachstandards ist aber absehbar. Über kurz oder lang werden alle C++-Compiler dieser Sprachdefinition folgen. Es wäre also widersinnig, den existierenden 2147618 Büchern über C++ ein weiteres hinzuzufügen, das nicht auf die kommenden Sprachfeatures eingeht. Zudem sind die *wesentlichen* neuen Sprachelemente schon seit einiger Zeit bekannt und werden sich nicht mehr verändern.

Die folgenden Abschnitte sollen kurz in die Thematik und den Aufbau des Buchs einführen.



Vorige Seite: 1 Vorbemerkungen **Eine Ebene höher:** 1 Vorbemerkungen **Nächste Seite:** 1.2 Aufbau des Buchs

---

1.1 Was das Buch nicht ist

Als erstes soll erläutert werden, was dieses Buch *nicht* ist:

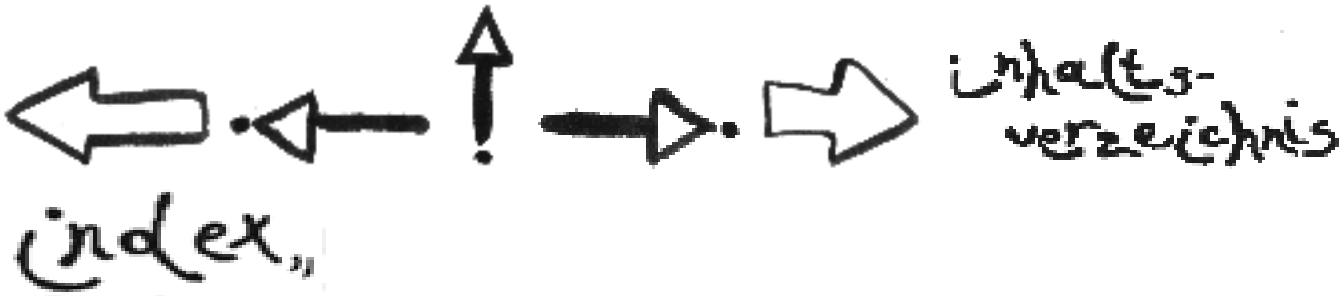
- Das Buch ist keine komplette und umfassende Referenz für die Programmiersprache C++ nach dem ANSI/ISO-Standard .

  Die Standard-Bibliothek von C++ weist inzwischen einen derart großen Umfang auf, daß sie nur in sehr geringem Ausmaß berücksichtigt werden konnte (Ein-/Ausgabe).

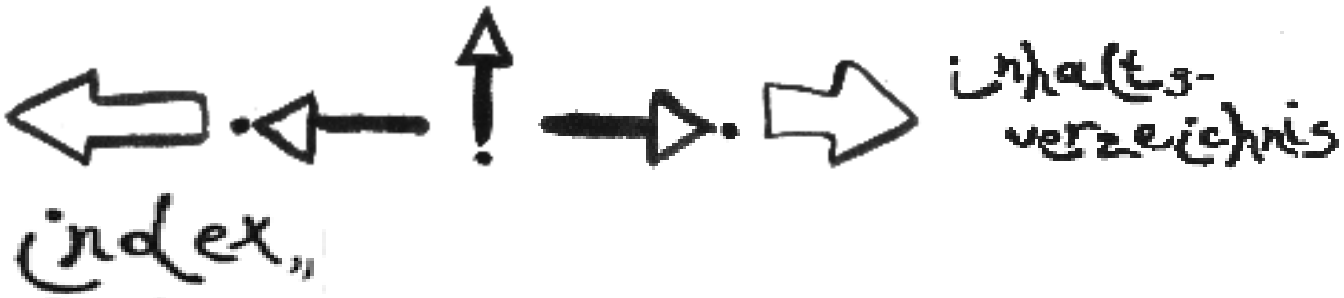
  Wesentliche Bereiche der Bibliothek wie die unter dem Namen *STL* (*Standard Template Library*) bekannten generischen Algorithmen und Datenstrukturen wurden ausgeklammert. Zum Erlernen der Programmiersprache und der wesentlichen Konzepte sind sie zum einen nicht notwendig, und zum andern erfordert ihre Komplexität eine umfangreiche Abhandlung, wie sie in diesem Buch nicht möglich wäre.

  Der Schwerpunkt dieses Buchs liegt auf der Vermittlung des (nach Meinung des Autors) für C++-*Einsteiger* und -*Anfänger* wesentlichen Wissens.
- Das Buch ist kein Hersteller-spezifisches C++-Lehrbuch, sondern bezieht sich ausschließlich auf den *Draft*-ANSI/ISO-Standard. Die hier besprochenen Konzepte sind damit auf allen ANSI/ISO-kompatiblen Entwicklungssystemen gültig, die hier gebrachten Programme können unabhängig von Compiler und Plattform übersetzt werden (vorausgesetzt, der Compiler entspricht dem Standard).
- Das Buch setzt keine C-Kenntnisse des Lesers voraus. Der Autor ist der Meinung, daß C++ eine eigenständige Programmiersprache ist, die auch als solche gelehrt werden sollte. C-Kenntnisse sind nützlich, aber nicht erforderlich.
- Das Buch konzentriert sich zudem ausschließlich auf den Bereich der Programmierung. Andere Themenbereiche wie etwa die Objektorientierte Analyse, der Entwurf von objektorientierten Systemen oder Entwurfsmuster (*Design Patterns*) werden in diesem Buch daher nicht erläutert.
- Weiterführende Themen wie *Handles*, *Master Pointer*, Speichermanagement und dergleichen werden ebenfalls nicht behandelt. Diese Themenbereiche haben eine andere Zielgruppe und setzen ein bereits gefestigtes Wissen um die Sprache C++ voraus.

Die in Anhang E angeführte Übersicht bietet dem Leser eine (subjektive) Auswahl von C++-Titeln, die auch weiterführende Themenbereiche abdecken.



Vorige Seite: 1 Vorbemerkungen **Eine Ebene höher:** 1 Vorbemerkungen **Nächste Seite:** 1.2 Aufbau des Buchs



Vorige Seite: 1.1 Was das Buch nicht **Eine Ebene höher:** 1 Vorbemerkungen **Nächste Seite:** 1.3 Schreibweisen und Konventionen

1.2 Aufbau des Buchs

Der Aufbau dieses Buchs unterscheidet sich bewußt von dem der meisten anderen C++-Lehrbücher. Durch seine Konzeption als Lehrbuch für C++-Einsteiger ist es unabhängig von den verschiedenen Entwicklungssystemen und -plattformen. Im Vordergrund steht nicht das Entwicklungssystem XY der Firma Z, sondern vielmehr die *Sprache* C++.

Dabei wurde versucht, das Buch nach einem alten österreichischen Schilehrer-Grundsatz („Vom Einfachen zum Komplexen``) zu gliedern. Die ersten Kapitel beschäftigen sich daher mit den „einfachen`` Aspekten der Sprache: Sprachelemente & Grammatik, Basisdatentypen, Ausdrücke und so weiter bis hin zu C++-Modulen. Diese Grundlagen mögen zwar (vor allem für Leser mit C-Erfahrung) langatmig erscheinen, doch eine gründliche und für den Leser nutzbringende Abhandlung der wesentlichen Konzepte ist nur möglich, wenn die „einfachen`` Sprachelemente klar erläutert sind.

Leser, die schon über entsprechende Erfahrungen verfügen, können sich bei den einführenden Kapiteln (Kapitel 2 bis einschließlich 6) auf eine auszugsweise Lektüre beschränken.

Im Detail ist das Buch wie folgt aufgebaut:

- Kapitel 2 bringt eine kurze Einführung in die Sprache C++ anhand von einigen kleinen Programmen. Im Vordergrund steht dabei nicht die Erläuterung von Sprachkonzepten, sondern vielmehr die Vorstellung des Schemas von einfachen C++-Programmen.

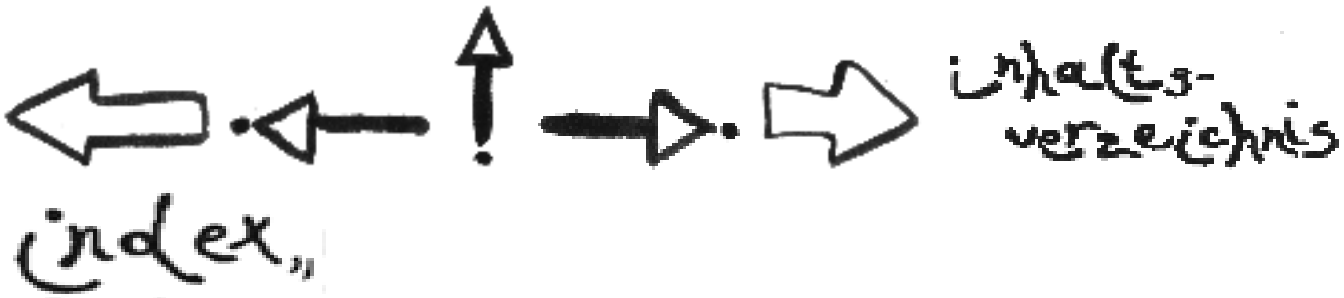
  Diese einfachen Programme können vom Leser abgeändert und getestet werden. So ist er in der Lage, von Anfang an Codebeispiele besser verstehen und eigene kleine Programme erstellen zu können.
- Kapitel 3 beschreibt, aus welchen Zeichen und Zeichenfolgen (lexikalischen Elementen) C++-Programme aufgebaut sind.
- Kapitel 4 stellt die grundlegenden Datentypen von C++ vor. Auf diesen einfachen Datentypen bauen die höheren und strukturierten Datentypen auf.
- Datenelemente werden in Ausdrücken, die aus Operanden und Operatoren bestehen, verarbeitet. In

Kapitel 5 werden die verschiedenen Arten von Ausdrücken sowie die einzelnen Operatoren vorgestellt.

- Grundelemente von C++-Programmen sind Anweisungen. Anweisungen enthalten Ausdrücke, deklarieren Objekte und steuern den Ablauf von Programmen. Kapitel 6 beschreibt die verschiedenen Arten von Anweisungen.
- Eine spezielle Anweisung ist die Funktionsanweisung. Sie ist das erste (und einfachste) Strukturierungs- und Abstraktionsmittel von C++ und faßt mehrere Anweisungen zu einer ,,Super-Anweisung`` zusammen. Kapitel 7 beschreibt Funktionen im Allgemeinen sowie verschiedene C++-Spezifika wie Operator-Funktionen, das Konzept des Überladens oder vorbelegte Funktionsparameter.
- Aufbauend auf diesen Kapiteln werden in Kapitel 8 die höheren Datentypen (Referenzen, Zeiger und Vektoren) im Detail besprochen. Neben den eigentlichen Datentypen steht dabei auch ihr Einsatz (zum Beispiel die Verwendung von Referenzen bei der Parameterübergabe) im Vordergrund. Auch die strukturierten Datentypen (Klassen) werden überblicksmäßig besprochen.
- Kapitel 9 vertieft einige bisher nur oberflächlich behandelte Aspekte: Gültigkeitsbereiche, Deklarationen und die verschiedenen Möglichkeiten der Typumwandlung in C++.
- Der ,,traditionelle`` Teil des Buchs wird durch einen Abschnitt über die modulbasierte Programmierung in C++ beschlossen (Kapitel 10). Das Kapitel beschreibt die Realisierung von Modulen in C++ und betont insbesonders die Bedeutung des Geheimnisprinzips.
- Kapitel 11 ist das ,,Kernkapitel`` des Buchs und erläutert das Klassenkonzept. Die Realisierung von Klassen und Themenbereiche wie Speichermanagement, Überladen von Operatoren, kanonische Form von Klassen etc. werden in diesem Kapitel anhand eines durchgängigen Beispiels demonstriert.
- Kapitel 12 beschreibt das Konzept der Generizität in C++. Besprochen werden generische Funktionen (Funktions-Templates) und Klassen (Klassen-Templates).
- Kapitel 13 zeigt die Realisierung von Vererbung in C++. Dabei stehen zunächst nur die ,,statischen`` Aspekte der Vererbung im Vordergrund.
- Darauf aufbauend werden in Kapitel 14 die dynamischen Aspekte (Polymorphismus, Abstrakte Basisklassen) sowie verschiedene andere weiterführende Themenbereiche (Laufzeit-Typinformation, Mehrfachvererbung) besprochen.
- Kapitel 15 beschließt das Buch mit einer Beschreibung des Konzepts der Ausnahmebehandlung.

In den Anhängen sind verschiedene ergänzende Abschnitte angeführt, die vom Leser je nach Bedarf durchgearbeitet werden können:

- Anhang A stellt die Ein-/Ausgabe in C++ kurz und überblicksweise vor.
- Anhang B beschreibt den C++-Präprozessor.
- Anhang C behandelt die ,,Hauptfunktion`` von C++-Programmen sowie die Parameterübergabe an diese.
- Anhang E führt in einer kurzen Übersicht eine Reihe von empfehlenswerten C++-Büchern an.



Vorige Seite: 1.1 Was das Buch nicht **Eine Ebene höher:** 1 Vorbemerkungen **Nächste Seite:** 1.3 Schreibweisen und Konventionen

Vorige Seite: 1.2 Aufbau des Buchs **Eine Ebene höher:** 1 Vorbemerkungen **Nächste Seite:** 1.4 Übungsbeispiele und Entwicklungssysteme

---

1.3 Schreibweisen und Konventionen

Schreibweisen wie zum Beispiel *der* Anwender werden in einer allgemeinen und geschlechtsneutralen Bedeutung verwendet.

Kursive Schreibweisen heben Begriffe hervor.

Englischsprachige Begriffe sind ebenfalls kursiv gedruckt. Setzen sie sich aus mehreren Wörtern zusammen, so werden sie nicht durch Bindestriche verbunden (*Call by Value* anstelle von *Call-by-Value*).

Codestücke und Schlüsselwörter sind im Zeichensatz `Typewriter` gesetzt.

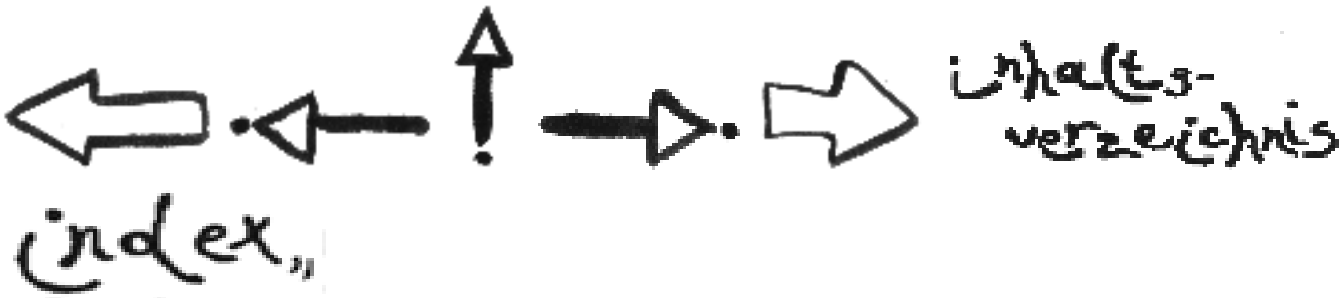
Zitate erfolgen wie üblich unter Angabe der entsprechenden Seitenzahlen. Lediglich in den entsprechend aufbereiteten Quellen werden die Stellen über logische Namen angegeben (zum Beispiel [ISO95, dcl.dcl]), da diese unabhängig von den konkreten Seitenzahlen sind. Eine weitere Bemerkung zu [ISO95]: Diese *Draft*-Version des C++-Standards ist im Gegensatz zur Version vom Dezember 1996 frei verfügbar. Wo immer möglich wird daher diese Version des Standards zitiert.

Zwei besondere Absatzformate sind zu erwähnen. Beide kennzeichnen spezielle Sachverhalte und sind durch eine Grafik am äußeren Rand und durch eine Einrückung nach außen vom restlichen Text abgegrenzt:

Wird im Text auf eine Tatsache besonders hingewiesen, so ist der entsprechende Textabschnitt durch ein Ausrufezeichen am Rand speziell gekennzeichnet.

Spezielle Tips hingegen werden durch seitliche ,,Striche`` markiert. Im Gegensatz zu den Anmerkungen sind diese Tips ,,subjektive`` Hinweise des Autors.

Vorige Seite: 1.2 Aufbau des Buchs **Eine Ebene höher:** 1 Vorbemerkungen **Nächste Seite:** 1.4 Übungsbeispiele und Entwicklungssysteme



Vorige Seite: 1.3 Schreibweisen und Konventionen **Eine Ebene höher:** 1 Vorbemerkungen **Nächste Seite:** 1.5 Die beigelegte CD

1.4 Übungsbeispiele und Entwicklungssysteme

Programmieren erlernt man nicht durch das Studium von Büchern oder Handbüchern, sondern nur durch Übung. Dieses Buch ist lediglich ein „Hilfsmittel`` und bereitet die Programmiersprache C++ für den Leser auf. Die Umsetzung der Konzepte bei konkreten Problemstellungen aber obliegt dem Leser.

Um Möglichkeiten der Umsetzung und die Anwendung der besprochenen Konzepte aufzuzeigen, werden daher Übungsbeispiele verwendet. Dabei handelt es sich weder um theoretische Fragen bezüglich Syntax oder andere Problembereiche von C++ noch um umfangreiche Beispiele, deren Lösungen nur auszugsweise angeführt werden können.

Vielmehr werden Beispiele verwendet, deren konkrete (und vor allem vollständige) Lösungen dem Leser die Anwendung der besprochenen Sprachkonzepte zeigen.

Übungsaufgaben werden ab Kapitel 11 (Klassen) gestellt und sind durch kurze Angaben, die bewußt vage und offen gehalten sind, spezifiziert. Die Angaben geben dem Leser damit großen Freiraum in bezug auf die Realisierung. Jedes einzelne Beispiel ist vollständig in dem Sinne, daß eine Klasse oder eine Reihe von Klassen inklusive Testprogramm zu erstellen ist.

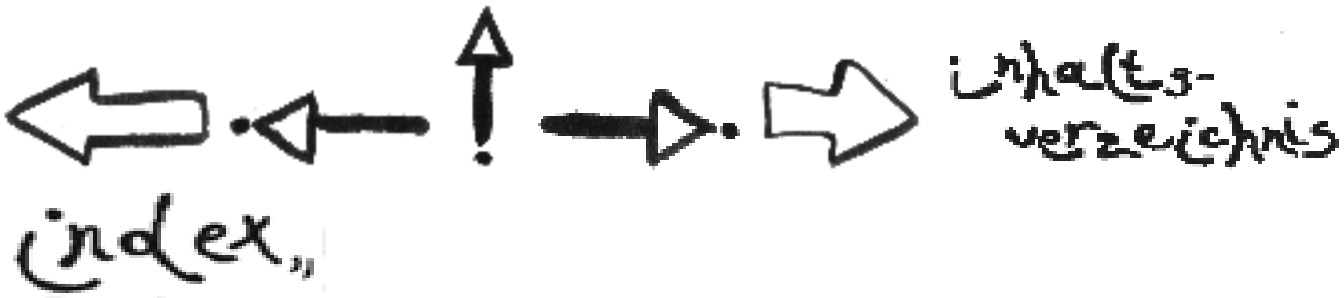
Die Aufgabenstellungen stammen aus Programmierkursen im universitären und außeruniversitären Bereich und wurden in den abgebildeten Formen von Kursteilnehmern gelöst.

In der elektronischen Version des Buchs, die auf der beigelegten CD enthalten ist, werden die Lösungen der Beispiele diskutiert. Dazu werden eine allgemeine Lösungsidee sowie Auszüge aus den wesentlichen Passagen der Lösung angeführt. Die vollständigen Lösungen sind ebenfalls auf der mitgelieferten CD enthalten.

Die einzelnen Aufgaben sind bewußt allgemein gehalten und sollten auf jedem C++-Entwicklungssystem, das dem ANSI/ISO-Standard weitgehend entspricht, übersetzt werden können.

Zu beachten ist, daß die angeführten Lösungen keineswegs *die* idealen Lösungen der Beispiele darstellen. Sie sind vielmehr *mögliche* Lösungen, die dem Leser zeigen sollen, wie bestimmte Konzepte umgesetzt werden können.

Da es sich bei allen Aufgaben um vollständige Beispiele handelt, kann der Leser experimentieren: Er kann die Dateien übersetzen und abändern, er kann die Programme ausführen oder *debuggen* und so den genauen Ablauf verfolgen.



Vorige Seite: 1.3 Schreibweisen und Konventionen **Eine Ebene höher:** 1 Vorbemerkungen **Nächste Seite:** 1.5 Die beigelegte CD

Vorige Seite: 1.4 Übungsbeispiele und Entwicklungssysteme **Eine Ebene höher:** 1 Vorbemerkungen
Nächste Seite: 2 Einführung

1.5 Die beigelegte CD

Dem Buch liegt eine CD bei, die folgende Informationen enthält:

- Alle Übungsbeispiele sind auf der CD vollständig aufgeführt.
- Das gesamte Buch ist auf der CD in Hypertext-Form (HTML) enthalten. Mit Hilfe von WWW-Browsern (zum Beispiel Netscapes *Navigator* oder Microsofts *Explorer*) kann diese Version als eine Art *Online-Manual* benutzt werden.

  Die HTML-Version umfaßt verschiedene Informationen, die im Buch nicht angeführt sind. So sind die Beispielprogramme vollständig (und nicht nur auszugweise) enthalten, verschiedene Referenzen geben interessante Adressen im Internet an etc.
- Auf der CD sind verschiedene Versionen des *Netscape Navigator* (*Windows*, *Unix*, *Macintosh*) enthalten. Der *Navigator* darf *ausschließlich offline* und zum Lesen der beigelegten HTML-Version des Buchs verwendet werden. Jegliche Zuwiderhandlung stellt einen Bruch des Lizenzabkommens dar!

Vorige Seite: 1.5 Die beigelegte CD **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** 2.1 Die Programmiersprache C++

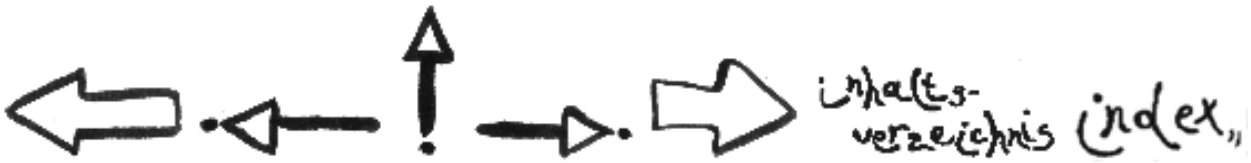
2 Einführung

Dieses Kapitel gibt einen kurzen Überblick über die Entwicklung von C++ und führt anschließend anhand von einigen kleinen Beispielen grob und unvollständig in die allgemeine Struktur von C++-Programmen ein. Dieser Abschnitt soll helfen, die verwendeten Programmbeispiele in den ersten Kapiteln besser verstehen zu können.

Vorige Seite: 2 Einführung **Eine Ebene höher:** 2 Einführung **Nächste Seite:** 2.1.1 Grundlagen von C++

2.1 Die Programmiersprache C++

Die folgenden zwei Abschnitte beschreiben kurz und überblicksmäßig die Programmiersprache C++ sowie ihre Entwicklung.



Vorige Seite: 2.1 Die Programmiersprache C++ **Eine Ebene höher:** 2.1 Die Programmiersprache C++ **Nächste Seite:** 2.1.2 Die Entwicklung von

2.1.1 Grundlagen von C++

C++ wurde von Bjarne Stroustrup [Str92] entwickelt und basiert auf der Programmiersprache C, die Anfang der siebziger Jahre (1971) für das Betriebssystem UNIX entwickelt wurde und Bestandteil dessen Programmierumgebung ist. Die klassische C-Sprachdefinition wurde 1978 von Brian Kernighan und Dennis Ritchie vorgelegt [KR78], fünf Jahre später begannen das *American National Standards Institute* (ANSI) und die *International Standardization Organization* (ISO) einen neuen Standard festzulegen.

Die zweite Grundlage für C++ ist Simula67. Von dieser Sprache wurde das Klassenkonzept mit abgeleiteten Klassen und virtuellen Funktionen entliehen.

Die Gründe, warum Stroustrup C als Basissprache für C++ wählte, sind:

- C ist vielseitig und prägnant.
- C ist effizient.
- C eignet sich für die meisten Aufgaben in der Systemprogrammierung.
- C ist auf praktisch allen Rechnersystemen verfügbar und einfach portierbar.
- C ist Bestandteil der UNIX-Programmierumgebung.
- Es existiert eine Vielzahl an Bibliotheksroutinen und Software-Utilities, die in C geschrieben sind und von C++ genutzt werden können.

C++ ist daher weitgehend kompatibel zu C. Die Kompatibilität zu C hat aber nicht nur Vorteile, wie Programm 2.1 zeigt.

Programm 2.1: Acht-Damen-Problem [Lib93]

```
v,i,j,k,l,s,a[99];
main()
{
  for(scanf("%d",&s);*a-s;v=a[j*=v]-a[i],k=i<s,j+=(v=j<
  s&&(!k&&!!printf(2+"\n\n%c"-(!l<<!j)," #Q"[l^v?(l^j)
  &1:2])&&++l||a[i]<s&&v&&v-i+j&&v+i-j))&&!(l%=s),v||
  (i==j?a[i+=k]=0:++a[i])>=s*k&&++a[--i]);
}
```

Dieses Programm löst das Problem der acht Damen, funktioniert aber auch mit 4-99 Damen. Als ,,Damenproblem`` wird die Aufgabe bezeichnet, acht Damen so auf einem Schachbrett zu plazieren, daß keine der Damen eine der anderen ,,schlagen kann``.

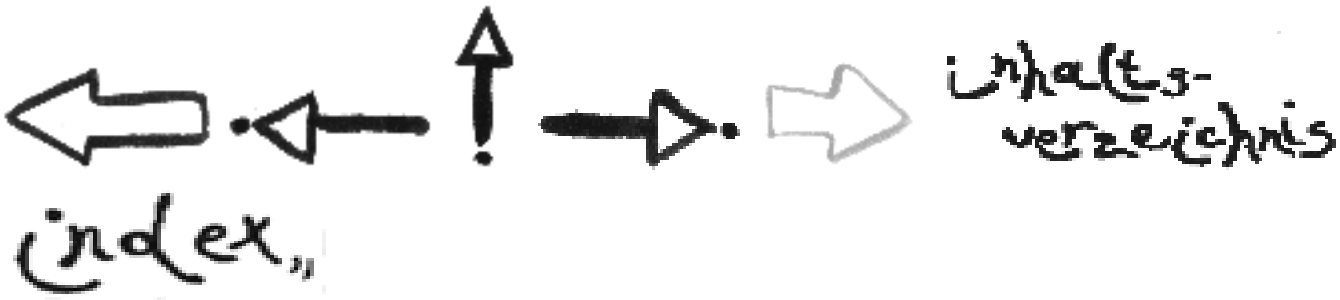
Das Programm ist ein durchaus legales C-Programm:

> ,,The authors note, that they use 'no preprocessor statements and no ifs, breaks, cases, functions, gotos, structures In short, it contains no C language that might confuse the innocenter'.`` [Lib93, S. 301]

Tatsächlich ist das Programm 1990 als Sieger beim jährlichen *Obfuscated C Code Contest* hervorgegangen - Kategorie *Best Small Program*.

Stroustrup ist für die enge Anlehnung von C++ an C oft und hart kritisiert worden. Der Erfolg von C++ aber bestätigt diese Entscheidung.

Vorige Seite: 2.1 Die Programmiersprache C++ **Eine Ebene höher:** 2.1 Die Programmiersprache C++ **Nächste Seite:** 2.1.2 Die Entwicklung von



Vorige Seite: 2.1.1 Grundlagen von C++ **Eine Ebene höher:** 2.1 Die Programmiersprache C++

2.1.2 Die Entwicklung von C++

1979 begann Bjarne Stroustrup seine Arbeit an *C with Classes* als ein Resultat seiner Probleme im Zusammenhang mit einer großen Simulationsaufgabe, die in Simula entwickelt wurde. *C with Classes* war eine erweiterte Version von C, die zusätzlich folgende Sprachelemente umfaßte:

- Klassen
- Vererbung ohne Polymorphismus
- Konstruktoren und Destruktoren
- Funktionen
- `friend`-Deklarationen
- Typprüfung

Die Implementierung von *C with Classes* bestand in einem Präprozessor, der *C with Classes*-Programme in C-Programme übersetzte.

Ab 1982 führte Stroustrup die Entwicklung von *C with Classes* fort. Der Entwicklungsprozeß mündete schließlich in der Version 3.0 von C++, die den im *The Annotated C++ Reference Manual* (*ARM*) [ES90] beschriebenen Sprachstandard implementierte. C++ umfaßte in dieser Version folgende neue Features:

- Virtuelle Funktionen
- Überladen von Funktionen und Operatoren
- Referenzen
- Konstanten
- Speichermanagement
- Templates

Im Jahr 1991 fand das erste Treffen der *ISO Workgroup 21* statt. Die Arbeit des Standardisierungs-Komitees mündete 1995 in einem sogenannten *Draft Standard* [ISO95]. Im Laufe des Jahres 1998 wird der endgültige Standard erwartet. Gegenüber der Version 3.0 wurde C++ im wesentlichen um folgende Features erweitert:

- Ausnahmebehandlung (*Exception Handling*)
- Laufzeit-Typinformationssystem (*Run-Time Typ Information*)

- Namensräume (*Name Spaces*)
- Neue Möglichkeiten der Typumwandlung

Basis für dieses Buch sind der letzte verfügbare *Draft*-C++-Standard [ISO95] sowie verschiedene (nicht „offizielle``) Dokumente, die diesen Standard betreffen.

Vorige Seite: 2.1.1 Grundlagen von C++ **Eine Ebene höher:** 2.1 Die Programmiersprache C++

Vorige Seite: 2.1.2 Die Entwicklung von **Eine Ebene höher:** 2 Einführung **Nächste Seite:** 2.2.1 Das erste Programm: `Hello world!`

2.2 Einfache C++-Programme

Die im folgenden angeführten einfachen C++-Programme sollen dem Leser über die ,,Tristheit`` der Grundlagen in den ersten Kapiteln hinweghelfen. Sie sollen ihm ermöglichen, kleine und einfache Testprogramme zu erstellen. Dabei ist es nicht notwendig, daß jede Anweisung der Beispielprogramme bis ins letzte Detail verstanden wird.

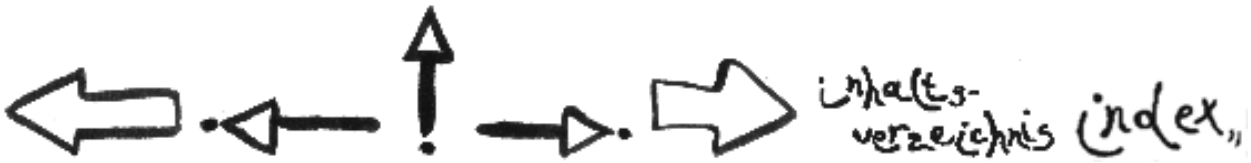
Im einfachsten Fall besteht ein C++-Programm aus einer Textdatei, die den Programmcode (*Source*) enthält. Diese Datei muß vom Compiler übersetzt werden. Der Compiler erzeugt dabei ein sogenanntes *Object File*. Ein *Object File* enthält die für den Computer ,,übersetzten`` Informationen der Programmdatei. In der Regel ist der Name eines *Object Files* gleich dem Namen des *Source Files* mit der Endung `.obj` (MSDOS, Windows oder OS/2) beziehungsweise `.o` (Unix).

Object Files können noch nicht ausgeführt werden. Jedes Programm benötigt bestimmte spezielle Codeteile für die Ausführung, die nicht extra vom Programmierer erstellt werden müssen. Außerdem wird in Programmen im Allgemeinen eine Reihe von bereits vordefinierten Funktionen und anderen Objekten verwendet, deren Definitionen nicht im *Source File* enthalten sind (zum Beispiel die Ein-/Ausgaben). Diese sind in bereits fertig übersetzten Teilen, den sogenannten ,,Bibliotheken`` (*Libaries*) abgelegt.

Anmerkung: Der Begriff ,,Objekt`` wird in den ersten Kapiteln *nicht* im objektorientierten Sinne verwendet. Vielmehr bezeichnet ,,Objekt`` vorerst ganz allgemein ein ,,Ding`` im Speicher.

Um aus einem *Object File* ein ausführbares Programm zu machen, ist es daher nötig, dieses mit den benötigten Bibliotheken zu *linken* (Deutsch: *binden*). Erst durch diesen Schritt wird ein ausführbares Programm erzeugt.

Vorige Seite: 2.1.2 Die Entwicklung von **Eine Ebene höher:** 2 Einführung **Nächste Seite:** 2.2.1 Das erste Programm: `Hello world!`



Vorige Seite: 2.2 Einfache C++-Programme **Eine Ebene höher:** 2.2 Einfache C++-Programme **Nächste Seite:** 2.2.2 Variablen und Kontrollstrukturen

---

2.2.1 Das erste Programm: `Hello world!`

Wie bereits oben erwähnt, kann ein C++-Programm aus einer einzelnen Datei bestehen, die die „Hauptfunktion`` `main` und gegebenenfalls weitere Funktionen enthält. `main` wird beim Start eines C++-Programms automatisch aktiviert.

Das traditionell erste zu erstellende C++-Programm (und zugleich eines der einfachsten) gibt den Text „Hello world¡` aus:

Programm 2.2: `Hello world!`

```
  //File: hello.cpp
  // Hello World

  // Schnittstelle von iostream inkludieren
  // iostream ist ein vordefiniertes Modul, daher wird
  // der Name mit den Zeichen <> begrenzt
#include <iostream>

  // "Hauptprogramm"
void main()
{
  cout << "Hello world!";
}
```

Einige Bemerkungen zum Programm:

- Die Ein-/Ausgabe ist in C++ in einer eigenen „Bibliothek`` (einer Sammlung von bereits vorgefertigten Programmteilen) vereinbart. Um Daten einlesen beziehungsweise ausgeben zu können, muß dem Compiler mitgeteilt werden, daß diese Bibliothek verwendet wird. Dies erfolgt durch die Anweisung `#include <iostream>`.
- `main` ist die Hauptfunktion jedes C/C++-Programms und wird vom C++-Laufzeitsystem beim Programmstart automatisch aktiviert.
- Hello World! benutzt das in allen C++-Entwicklungssystemen vorhandene Modul `iostream`, um den Text auszugeben. `iostream` realisiert eine einfache textuelle Ein-/Ausgabe.

  Grundsätzlich erfolgen alle Eingaben im Programm über Anweisungen der folgenden Art:

  ```
  cin >> Argument;
  ```

  Dabei wird der Wert des Arguments unter Verwendung des Operators >> („lesen von``, *get from*) vom Standard-Eingabestrom `cin`, also von der Tastatur, eingelesen.

  Die Ausgabe erfolgt mit Anweisungen der Art

  ```
  cout << Argument;
  ```

  Der Wert des Arguments wird mit Hilfe des Ausgabeoperators << („schreiben nach``, *put to*) auf den Standard-Ausgabestrom `cout` (und damit auf den Bildschirm) geschrieben.

Sowohl Eingabeoperator >> wie auch Ausgabeoperator << können ,,kaskadiert`` werden:

```
cin >> "Hello" >> " " >> "world!";
```

Vorige Seite: 2.2 Einfache C++-Programme **Eine Ebene höher:** 2.2 Einfache C++-Programme **Nächste Seite:** 2.2.2 Variablen und Kontrollstrukturen

Vorige Seite: 2.2.1 Das erste Programm: `Hello world!` **Eine Ebene höher:** 2.2 Einfache C++-Programme **Nächste Seite:** 2.2.3 Funktionen

2.2.2 Variablen und Kontrollstrukturen

Das folgende Programm löst eine Gleichung der Form $ax^2+bx+c = 0$. Das Programm liest die Koeffizienten *a*, *b* und *c* ein und ermittelt anschließend das Ergebnis.

Die Lösung einer derartigen Gleichung kann nach der Formel $x_{1,2} = -0.5*p \pm sqrt((0.5*p)^2 - q)$ erfolgen, wobei sich *p* und *q* aus der umgeformten Gleichung ergeben ($p = b/a$, $q = c/a$) und *sqrt* die Quadratwurzelfunktion darstellt:

Programm 2.3: `Quadratische Gleichung`

```
#include <iostream>
#include <cmath>

void main()
{
  float a, b, c;
  float help, diskr;

  cout << "Loesen einer Gleichung der Form"
       << " a*x*x+b*x+c=0" << endl;
  cout << "a="; cin >> a;
  cout << "b="; cin >> b;
  cout << "c="; cin >> c;

    // Loesung wenn a!=0
  if (a!=0) {
    help=-0.5*b/a;
    diskr=help*help-c/a;
    if (diskr==0) {        // Eine (reelle) Loesung
      cout << "x = ";
      cout << help;
    } else if (diskr>0) {  // Zwei reelle Loesungen
        cout << "\nx1=" << help+sqrt(diskr);
        cout << "\nx2=" << help-sqrt(diskr);
    } else {               // Zwei komplexe Loesungen
        cout << "\nx1=" << help << '+' << sqrt(-diskr);
        cout << "\nx2=" << help << '-' << sqrt(-diskr);
    }
  } else if (b!=0) {       // b=0: Division nicht erlaubt
    cout << "x = " << -c/b << endl;
  } else if (c!=0) {
    cout << "Keine Loesung!" << endl;
  } else {
```

```
    cout << "Unendlich viele Loesungen!" << endl;
  }
}
```

Einige Erklärungen zum Programm:

- `cmath` ist ein Modul, das verschiedene mathematische Operationen bereitstellt. Im Programm wird `sqrt` (Quadratwurzel) verwendet.
- Die Koeffizienten werden im Programm durch Variablen repräsentiert. Variablen sind - so wie in der Mathematik - Größen, deren Wert ,,variabel`` ist und sich im Laufe des Programms verändern kann. Jeder Variable ist ein ,,Typ`` zugeordnet, der den Wertebereich und die möglichen Operationen bestimmt. Die Variablen im Beispiel sind Fließkommazahlen und vom Typ `float`.

  Beispiele für andere mögliche Typen sind etwa `char` (Zeichen) oder `int` (Ganze Zahlen):

  ```
  int    tag, monat, jahr;
  char   trennZeichen;

  cin >> tag >> monat >> jahr;
  cin >> trennZeichen;

  cout << "Datum:" << tag << trennZeichen
       << monat << trennZeichen << jahr;

  cout << "\n";  // Zeilenvorschub ausgeben
  ```

  Bei der Eingabe von `17`, `4`, `1991` und `/` entsteht folgende Ausgabe:

  `17/4/1991`

  Anmerkung: `cin` und `cout` sind für alle vordefinierten Datentypen (`int`, `char`, `float`) definiert.
- Die `if`-Abfrage `if (a!=0) { ... }` bewirkt, daß die zwischen den geschwungenen Klammern angeführten Anweisungen dann ausgeführt werden, wenn die zugehörige Bedingung (`a!=0`) wahr ist, `a` also ungleich 0 ist.
- Ist nach der `if`-Anweisung ein `else` angeführt, so bedeutet dies, daß die dahinter angeführte Anweisung ausgeführt wird, wenn die Bedingung der `if`-Anweisung falsch war.

  Die Anweisung `if (a!=0) {` *Anweisungsfolge1* `} else if (b!=0) {` *Anweisungsfolge2* `} ...` bewirkt, daß *Anweisungsfolge1* ausgeführt wird, wenn `a` ungleich 0 ist. Andernfalls (`else`) wird geprüft, ob `b` ungleich 0 ist. Wenn ja, wird *Anweisungsfolge2* ausgeführt, ansonsten

---

Vorige Seite: 2.2.1 Das erste Programm: `Hello world!` **Eine Ebene höher:** 2.2 Einfache C++-Programme **Nächste Seite:** 2.2.3 Funktionen

Vorige Seite: 2.2.2 Variablen und Kontrollstrukturen **Eine Ebene höher:** 2.2 Einfache C++-Programme

2.2.3 Funktionen

Programm 2.4 berechnet einen ,,Turm`` und gibt ihn aus. Dabei wird eine Zahl eingelesen und anschließend mit zwei multipliziert. Das Ergebnis wird ausgeben und mit drei multipliziert. Dies wird bis zur Zahl 9 wiederholt, anschließend erfolgen Divisionen durch 2 bis 9. Das Ergebnis ist wiederum die Ausgangszahl. Der Turm ist ,,formatiert`` auszugeben, die Eingabe der Zahl 0 soll das Programm beenden. Negative Eingaben sind nicht zulässig.

In diesem Programm werden bereits die wichtigsten Kontrollstrukturen verwendet: Bedingungsanweisungen (`if`) und Schleifen (`for`, `while`, `do`).

Die formatierte Ein-/Ausgabe erfolgt über die Module `iostream` und die Funktion `setw` des Moduls `iomanip`.

Programm 2.4: Programm `Turm`

```
#include <iostream>

  // iomanip-Modul erlaubt das Festsetzen der
  // Feldlaenge bei der Ausgabe
#include <iomanip>

  // Funktion berechneTurm, berechnet u. gibt einen
  // "Turm" basierend auf der Zahl num aus
void berechneTurm(int num)
{
    // Vereinbarung der lokalen Variablen
  long  help = num;

    // Multiplikation der Zahlen, for-Schleife:
    // die Variable opnd wird mit 2 initialisiert.
    // Dann wird der Schleifenrumpf durchlaufen (alle
    // Anweisungen zwischen { und }) und opnd erhoeht.
    // Anschliessend wird geprueft, ob opnd < 10 ist.
    // Falls ja, dann wird der Schleifenrumpf abermals
    // durchlaufen ...
  for (int opnd=2; opnd<10; opnd++) {

    cout << setw(10); // Feldlaenge f. Ausgabe (rechtsbuendig)
    cout << help;     // Ausgabe von help
    cout << setw(0);  // Loeschen d. Feldes f. die Ausgabe

      // Ausgabeanweisungen koennen auch kaskadiert werden ...
      // '\n' entspricht einem Zeilenvorschub
    cout << " * " << opnd << '\n';
    help *= opnd;   // entspricht "help = help*opnd"
  }
```

```
    // Division der Zahlen, for-Schleife
  for (int opnd=2; opnd<10; opnd++) {
      // Feldlaenge kann direkt im cout-Statement gesetzt werden
    cout << setw(9) << help << setw(0) << " / " << opnd << '\n';
    help /= opnd;   // entspricht "help = help/opnd"
  }
    // statt '\n' kann auch die Konstante endl benutzt werden ...
  cout << help << endl << endl;
}

void main()    // Hauptfunktion
{
  int   num;

    // C++-Repeat-Schleife, fuehrt Schleifenrumpf (Anweisungen
    // zwischen "{ // do" und "} while (num>0); // do" aus, solange
    // num > 0 ist. Die Pruefung der Bedingung erfolgt *nach* dem
    // Schleifenrumpf.
  do { // do
    cout << "Bitte geben Sie eine Zahl ein (>= 0)!" << "\n";
    cout << "0 = Ende!!" << "\n\n";
    cin >> num;

      // Solange num < 0: fuehre Schleifenrumpf (Anweisungen
      // zwischen { und }) solange aus, als num<0 (die Pruefung der
      // Bedingung erfolgt vor dem Ausfuehren des Schleifenrumpfs)
    while (num<0) {
      cout << " Bitte geben Sie eine Zahl >= 0 ein!" << "\n";
      cin >> num; cout << "\n";
    }
      // num > 0: Fuehre Rumpf (Anweisungen zwischen { und }) aus
    if (num > 0) {
      berechneTurm (num);
    }
    cout << "\n\n";
  } while (num > 0); // do
}
```

`berechneTurm` ist eine ,,Funktion``. Eine Funktion ist vereinfacht gesagt eine Zusammenfassung von Anweisungen zu einer Einheit. Die zusammengefaßten Anweisungen können dann wie eine einzelne Anweisung behandelt werden. Im Detail weist `berechneTurm` folgende Besonderheiten auf:

- Die `for`-Schleife ist eine ,,Zählschleife``, die die angegebene Variable `opnd` von 2 bis 9 zählt und für jeden Wert die enthaltenen Anweisungen ausführt.
- Die Anweisung `cout << setw(10)` setzt die Länge des Ausgabefeldes mit 10 fest. Das bedeutet, daß alle Ausgaben rechtsbündig in einem Feld von zehn Zeichen erfolgen (sofern die Feldlänge ausreicht). Beispiele für derartige Ausgaben:

```
0123456789
    Hallo!
       Wau
Test123456
Donaudampfschiffahrtsgesellschaft
```

Das Hauptprogramm `main` besteht aus einer Schleife (`do { ... } while (num > 0);`, die wiederum eine Schleife (`while (num<0) { }` sowie eine Bedingungsanweisung (`if (num>0) { }` enthält:

- Die erste Schleife (die „äußere`` Schleife) enthält eine zweite Schleife (die „innere``) sowie eine `if`-Anweisung. Die äußere Schleife ist eine sogenannte `do`-Schleife, die alle Anweisungen zwischen dem folgenden `{` und dem abschließenden `}` wiederholt, solange die dahinter angeführte Bedingung `(num > 0)` gilt. Die Prüfung der Bedingung erfolgt jeweils *nach* der Ausführung dieser Anweisungen.
- Die innere `while`-Schleife wiederholt die enthaltenen Anweisungen solange die gelesene Zahl kleiner 0 ist.
- Die Bedingungsanweisung `if (num > 0) { }` führt die enthaltenen Anweisungen dann aus, wenn `num` größer 0 ist.

Vorige Seite: 2.2.2 Variablen und Kontrollstrukturen **Eine Ebene höher:** 2.2 Einfache C++-Programme

Vorige Seite: 2.2.3 Funktionen **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** 3.1 Sprachbeschreibung mit Grammatik

---

3 Lexikalische Elemente von C++

Dieses Kapitel befaßt sich mit den Symbolen, aus denen (gültige) C++-Programme bestehen. Um den konkreten Aufbau dieser Symbole exakt zu beschreiben, wird, ähnlich wie in natürlichen Sprachen, eine Grammatik verwendet.

Diese Grammatik wird zunächst dargelegt, dann erfolgt die Beschreibung der verschiedenen Symbole. C++ kennt folgende Arten von Symbolen:

- Bezeichner
- Schlüsselwörter
- Literale
  Ganze Zahlen, Fließkommazahlen, Zeichen, Zeichenketten, Wahrheitswerte
- Operatoren und Begrenzer
- Kommentare

Vorige Seite: 3 Lexikalische Elemente von **Eine Ebene höher:** 3 Lexikalische Elemente von **Nächste Seite:** 3.2 Bezeichner

---

3.1 Sprachbeschreibung mit Grammatik

Es gibt verschiedene Möglichkeiten, zu beschreiben, wie ein ,,richtiges`` Programm aussieht und aus welchen Symbolen es zusammengesetzt ist. Eine Variante ist zum Beispiel die Aufzählung aller korrekten Programme, eine zweite die verbale Erklärung von Regeln , nach denen Programme aufgebaut sind.

Die erste Möglichkeit ist nicht umsetzbar, da die Menge aller möglichen Programme unendlich groß ist. Auch die zweite ist nicht zufriedenstellend, da sie zu umständlichen und oft nicht eindeutigen Erklärungen führt.

In der Informatik bedient man sich daher einer dritten Möglichkeit, der Beschreibung einer Programmiersprache durch eine formale Grammatik. Die Grammatik einer Programmiersprache besteht (analog der Grammatik von natürlichen Sprachen) aus einer Menge von *Regeln*, die angibt, wie die einzelnen *Sätze* (Anweisungen), aus denen sich ein Programm zusammensetzt, aufgebaut sein müssen. Eine Regel besteht aus einem zu definierenden *Symbol*, gefolgt von einem Doppelpunkt und der Definition des Symbols. Alle Symbole einer Grammatik, die auf der linken Seite einer Regel (= vor dem Doppelpunkt) erscheinen, werden als *Non-Terminalsymbole* bezeichnet, Symbole, die ausschließlich auf der rechten Seite vorkommen, als *Terminalsymbole* .

Dazu ein kurzes und unvollständiges Beispiel:

```
Name:
                Buchstabe
                Name Buchstabe
Buchstabe: one of
                a b c d e f g h i j k l m
                  n o p q r s t u v w x y z
Zahl:
                Vorzeichenopt Ziffernfolge
```

```
Vorzeichen: one of
                + -

Ziffer:  one of
                0 1 2 3 4 5 6 7 8 9

Ziffernfolge:
                Ziffer
                Ziffernfolge Ziffer
```

Die Erklärung zur oben angeführten Grammatik:

- Ein *Name* ist entweder ein *Buchstabe* oder eine Aneinanderreihung (Sequenz) von einem *Namen* gefolgt von einem *Buchstaben*. Die ,,Entweder-Oder``-Beziehung ergibt sich aus der Anführung der beiden um eine Ebene eingerückten Vereinbarungen. Ist eine Vereinbarung länger als eine Zeile, so werden die folgenden Zeilen weiter eingerückt (wie im Fall von *Buchstabe*).
- Ein *Buchstabe* ist eines der dahinter angeführten Symbole (`one of`, *einer von*): `a`, `b` `z`.
- Eine *Zahl* ist ein optionales *Vorzeichen* gefolgt von einer *Ziffernfolge*. Optional bedeutet, daß sich vor einer *Ziffernfolge* genau ein oder kein Vorzeichen befinden kann.
- Ein Vorzeichen ist entweder ein `+` oder ein `-`.
- Eine *Ziffer* ist eines der dahinter angeführten Symbole (`0` `9`).
- Eine *Ziffernfolge* ist entweder eine *Ziffer* oder aber eine *Ziffernfolge* gefolgt von einer *Ziffer*.

Gültige Sätze lassen sich durch Anwendung der Regeln der Grammatik bilden beziehungsweise auf das ,,Wurzelsymbol`` zurückführen. Ein Beispiel für einen gültigen (= der Grammatik entsprechenden) Namen ist `abcdef`:

- `f` ist ein *Buchstabe*, der vordere Teil des Satzes (`abcde`) ist ein *Name* (*Name Buchstabe*).
- Der *Name* `abcde` kann wiederum in einen *Buchstaben* (`e`) und einen *Namen* (`abcd`) zerlegt werden.
- Der *Name* `abcd` wird so lange auf die Folge von *Name Buchstabe* zurückgeführt, bis schlußendlich der *Buchstabe* `a` übrigbleibt, der auf *Name* zurückgeführt werden kann.

Ebenso sind `a` und `abba` gültige *Namen* und `+1234`, `98` oder `-98` gültige *Zahlen*. `1a`, `_ABC` oder `12` sind hingegen keine gültigen *Namen*.

Zwei Anmerkungen zur verwendeten Grammatik:

- Die hier verwendete Beschreibungsform einer Grammatik geht auf [ISO95] beziehungsweise [Str92] zurück und stellt an und für sich keine der ,,üblichen`` wissenschaftlichen Beschreibungsformen dar. Sie wird aber im Umfeld von C++ und damit auch in diesem Buch verwendet.
- Die Grammatik von C++ ist zum Teil derart komplex, daß sie schwer verständlich ist und keinesfalls Klarheit schafft. Dieses Buch ist kein Sprachhandbuch zur Programmiersprache C++ und erhebt daher nicht den Anspruch, C++ in seiner Gesamtheit zu beschreiben. Formale

Beschreibungen der Syntax der Programmiersprache C++ werden ausschließlich dort verwendet, wo sie Klarheit schaffen und Zweideutigkeiten vermeiden helfen. Beschreibungen, die keine Klarheit bringen, werden nicht verwendet.

Vorige Seite: 3 Lexikalische Elemente von **Eine Ebene höher:** 3 Lexikalische Elemente von **Nächste Seite:** 3.2 Bezeichner

Vorige Seite: 3.1 Sprachbeschreibung mit Grammatik **Eine Ebene höher:** 3 Lexikalische Elemente von
Nächste Seite: 3.3 Schlüsselwörter

3.2 Bezeichner

Bezeichner sind Namen für beliebige ,,Objekte`` im Programm: Variablen, Funktionen, selbstdefinierte Datentypen, Klassen, Objekte etc. Für Bezeichner gilt, daß sie aus beliebigen Buchstaben, Ziffern und dem Zeichen _ (*Underscore*) zusammengesetzt sind, mit der Einschränkung, daß das erste Zeichen keine Ziffer sein darf.

Sie sind wie folgt aufgebaut:

```
identifier:
                nondigit
                identifier nondigit
                identifier digit
nondigit: one of
                _ a b c d e f g h i j k l m
                  n o p q r s t u v w x y z
                  A B C D E F G H I J K L M
                  N O P Q R S T U V W X Y Z
digit: one of
                0 1 2 3 4 5 6 7 8 9
```

Beispiele für gültige Bezeichner sind `hallo`, `_hallo`, _ (*Underscore*) oder `Test123`. Dagegen sind `1Hallo`, `+abcd` oder etwa `äbba` ungültige Bezeichner.

Zwei Dinge sind in bezug auf Bezeichner wesentlich:

- C++ ist *Case Sensitive* , das heißt, es unterscheidet zwischen Groß- und Kleinschreibung. `sum` und `Sum` sind daher zwei verschiedene Bezeichner.
- Wörter werden durch Zeichen, die keine Buchstaben, Ziffern und *Underscores* sind, begrenzt.

Daher ist zum Beispiel `elseif` ein einzelner Bezeichner und nicht die Aufeinanderfolge von `else` und `if`.

Wichtig ist, bei der Wahl von Bezeichnernamen konsistent vorzugehen. Bezeichnernamen sollten nach bestimmten (frei wählbaren) Regeln aufgebaut sein. So wird erreicht, daß bereits aus dem Namen hervorgeht, um welche Art von Bezeichnern es sich handelt. Im folgenden werden einige Richtlinien zur Wahl der Bezeichner vorgeschlagen (basierend auf [HN93]):

- Variablen, Konstanten, Funktionen und Methoden beginnen mit einem Kleinbuchstaben, Typen mit einem Großbuchstaben.
- Namen, die aus mehr als einem Wort zusammengesetzt sind, werden in *Mixed Case* notiert: `endOfLineReached`.
- Bezeichner sollten ,,sinnvolle`` Namen erhalten:
  - Variablennamen sollten Rückschlüsse auf ihre Art erlauben und entsprechend gewählt werden: `isDone` kann wahr oder falsch sein, eine Variable `numberOfLines` wird eine Zahl sein und eine Variable `firstName` eine Zeichenkette.
  - Funktionen und Methoden werden mit ,,Tätigkeiten`` umschrieben: `terminatePrg`, `insertBlanks` oder `draw`.
- Bezeichner, die mit zwei *Underscores* (_), denen ein Großbuchstabe folgt, beginnen, sind zwar syntaktisch richtig, allerdings oft für Compiler-interne Bezeichner reserviert und sollten daher vermieden werden.

Vorige Seite: 3.1 Sprachbeschreibung mit Grammatik **Eine Ebene höher:** 3 Lexikalische Elemente von
Nächste Seite: 3.3 Schlüsselwörter

Vorige Seite: 3.2 Bezeichner **Eine Ebene höher:** 3 Lexikalische Elemente von **Nächste Seite:** 3.4 Literale

3.3 Schlüsselwörter

Schlüsselwörter sind reservierte Bezeichner mit einer vorgegebenen Bedeutung und dienen zur Beschreibung von Aktionen und Objekten in C++-Programmen. Sie dürfen daher nicht anderweitig verwendet werden. Tabelle 3.1 zeigt die Schlüsselwörter von C++.

Kategorie	Schlüsselwörter
Basisdatentypen	`bool true false char wchar_t short signed`
	`unsigned int long float double void`
Datentypen	`enum class struct union typedef`
Speicherklassen	`auto extern register static mutable`
Qualifizierer	`const volatile`
Kontrollstrukturen	`if while else case switch default do for`
Sprunganweisungen	`break continue return goto`
Typumwandlungen	`const_cast dynamic_cast reinterpret_cast`
	`static_cast`
Typsystem	`typeid typename`
Namensräume	`namespace using`
Operatoren	`new delete sizeof operator this`
Zugriffsschutz	`friend private public protected`
Funktionsattribute	`inline virtual explicit`
Generizität	`template`
Ausnahmebehandlung	`catch throw try`
Assemblercode	`asm`

Tabelle: Schlüsselwörter in C++

Neben diesen Standard-Schlüsselwörtern existieren noch andere reservierte Wörter, die alternativ zu den Symbolen für Operatoren (Operator-Symbole) verwendet werden können und in Abschnitt 3.5.3 angeführt sind.

Vorige Seite: 3.2 Bezeichner **Eine Ebene höher:** 3 Lexikalische Elemente von **Nächste Seite:** 3.4 Literale

Vorige Seite: 3.3 Schlüsselwörter **Eine Ebene höher:** 3 Lexikalische Elemente von **Nächste Seite:** 3.4.1 Ganze Zahlen

---

3.4 Literale

Literale sind Zahlen, Wahrheitswerte oder Zeichenketten im Programmtext. So wie alle anderen Symbole eines Programms müssen auch sie nach bestimmten Regeln aufgebaut sein.

Vorige Seite: 3.4 Literale **Eine Ebene höher:** 3.4 Literale **Nächste Seite:** 3.4.2 Fließkommazahlen

3.4.1 Ganze Zahlen

Eine ganze Zahl besteht aus einer Folge von Ziffern. Es wird zwischen Dezimal-, Oktal- und Hexadezimalzahlen unterschieden, wobei jede der Zahlen durch die (optionale) Angabe eines sogenannten Suffixes noch genauer spezifiziert werden kann:

```
integer-literal:
                decimal-literal integer-suffixopt
                octal-literal integer-suffixopt
                hexadecimal-literal integer-suffixopt
integer-suffix:
                unsigned-suffix long-suffixopt
                long-suffix unsigned-suffixopt
unsigned-suffix: one of
                u  U
long-suffix: one of
                l  L
```

Integer-Suffix bestimmt den Wertebereich einer Zahl genauer: Ein Suffix `u` beziehungsweise `U` bedeutet, daß die Zahl vorzeichenlos ist, `l` beziehungsweise `L` gibt an, daß es sich um eine sehr große Zahl (eine sogenannte `long`-Zahl) handelt.

Die einzelnen Zahlen im Detail:

- Eine Dezimalzahl beginnt mit einer von 0 verschiedenen Ziffer, der eine beliebig lange Reihe von Ziffern (auch 0) folgen kann:

```
decimal-literal:
                nonzero-digit
```

decimal-literal digit

digit: one of

0 1 2 3 4 5 6 7 8 9

nonzero-digit: one of

1 2 3 4 5 6 7 8 9

- Eine Oktalzahl beginnt mit der Ziffer 0, der eine Sequenz von Oktalziffern folgen kann:

octal-literal:

0

octal-literal octal-digit

octal-digit: one of

0 1 2 3 4 5 6 7

- Eine Hexadezimalzahl besteht aus einer Reihe von Hexadezimalziffern, denen die Sequenz `0x` beziehungsweise `0X` vorangestellt ist:

hexadecimal-literal:

0x *hexadecimal-digit*

0X *hexadecimal-digit*

hexadecimal-literal hexadecimal-digit

hexadecimal-digit: one of

0 1 2 3 4 5 6 7 8 9

a b c d e f A B C D E F

Einige Beispiele für gültige Zahlen:

Ganze Dezimalzahlen	`65` `123` `50000` `1234567L` `987UL` `45ul`
Oktalzahlen	`024` `01` `077UL`
Hexadezimalzahlen	`0x1A` `0XFFFF` `0x1234567L`
Ungültige Zahlen	`08` `1A` `0xG` `1234ulL` `00xa`

- Die Verwendung von `l` zur Spezifikation von `long`-Zahlen sollte aus Gründen der Lesbarkeit (leicht zu verwechseln mit `1`) unterbleiben. Es ist besser, `L` zu verwenden.

Vorige Seite: 3.4 Literale **Eine Ebene höher:** 3.4 Literale **Nächste Seite:** 3.4.2 Fließkommazahlen

Vorige Seite: 3.4.1 Ganze Zahlen **Eine Ebene höher:** 3.4 Literale **Nächste Seite:** 3.4.3 Zeichen

3.4.2 Fließkommazahlen

Eine Fließkommazahl besteht aus einer gebrochenen Zahl optional gefolgt von einem Exponenten (E). Der genaue Aufbau ist wie folgt angegeben:

```
floating-literal:
                fractional-constant exponent-part floating-suffix
                digit-sequence exponent-part floating-suffix
fractional-constant:
                digit-sequenceopt . digit-sequence
                digit-sequence .
exponent-part:
                e signopt digit-sequence
                E signopt digit-sequence
sign: one of
                +  -
digit-sequence:
                digit
                digit-sequence digit
floating-suffix: one of
                f  l  F  L
```

Ein Suffix `f` oder `F` bezeichnet - ebenso wie eine Zahl ohne Suffix - eine Fließkommazahl mit einfacher Genauigkeit. Ein Suffix `l` oder `L` gibt an, daß die Fließkommazahl eine erhöhte Genauigkeit aufweist.

Beispiele für gültige Fließkommazahlen sind `3.141592`, `10000.`, `3.0856782E16F`, `1234567.89E-20L`. Beispiele für ungültige Fließkommazahlen sind `0x123.56`, `45E+-23` oder

```
12.345FL.
```

Vorige Seite: 3.4.2 Fließkommazahlen **Eine Ebene höher:** 3.4 Literale **Nächste Seite:** 3.4.4 Zeichenketten

3.4.3 Zeichen

Ein Zeichen-Literal wird in einfache Hochkommas eingeschlossen angegeben. Zeichen-Literale umfassen neben den druckbaren Zeichen auch Steuerzeichen. Um diese (nicht druckbaren) Zeichen darzustellen, wird eine sogenannte *Escape*-Sequenz verwendet. Eine *Escape*-Sequenz wird mit dem Zeichen \ (*Backslash*) eingeleitet und bestimmt ein Zeichen aus dem ASCII mittels einer oktalen oder hexadezimalen Zahl. Um das Zeichen \ selbst darzustellen, wird \\ verwendet.

```
character-literal:
                'c-char-sequence'
                L'c-char-sequence'
c-char-sequence:
                c-char
                c-char-sequence c-char
c-char:
                any member of the source character set except
                  the single-quote ', backslash \, or new-line character
                escape-sequence
escape-sequence:
                simple-escape-sequence
                octal-escape-sequence
                hexadecimal-escape-sequence
simple-escape-sequence: one of
                \'  \"  \?  \\
                  \a  \b  \f  \n  \r  \t  \v
octal-escape-sequence:
```

```
                \ octal-digit
                octal-escape-sequence octal-digit
hexadecimal-escape-sequence:
                \x hexadecimal-digit
                hexadecimal-escape-sequence hexadecimal-digit
```

Beispiele für Zeichen-Literale sind:

`'A'`	Zeichen A	`'0'`	Zeichen 0
`'\"'`	einfaches Hochkomma	`'"'`	doppeltes Hochkomma
`'\\'`	Backslash	`'\a'`	Alarmton
`'\b'`	Backspace	`'\f'`	Seitenvorschub
`'\n'`	Neue Zeile	`'\r'`	Wagenrücklauf
`'\t'`	Tabulator	`'\v'`	Vertikaltabulator
`'\x12'`	Zeichen mit ASCII-Wert 18	`'\12'`	Zeichen mit ASCII-Wert 10
`'\xFE'`	Zeichen mit ASCII-Wert 254	`'\062'`	Zeichen mit ASCII-Wert 50

C++ unterstützt auch multilinguale Zeichensätze mit mehr als 256 Zeichen. Derartige Zeichen beginnen mit `L` und sind Teil eines sogenannten *Wide Character Sets* (Typ `wchar_t`).

Vorige Seite: 3.4.2 Fließkommazahlen **Eine Ebene höher:** 3.4 Literale **Nächste Seite:** 3.4.4 Zeichenketten

Vorige Seite: 3.4.3 Zeichen **Eine Ebene höher:** 3.4 Literale **Nächste Seite:** 3.4.5 Wahrheitswerte

3.4.4 Zeichenketten

Ein Zeichenketten-Literal ist eine (möglicherweise auch leere) Sequenz von Zeichen, die in doppelte Hochkommas eingeschlossen ist. Der Zeichenkette kann optional ein `L` vorangestellt werden, um eine Zeichenkette mit Multilingual-Zeichen zu kennzeichnen.

string-literal:

" *s-char-sequence*$_{opt}$ "

`L` " *s-char-sequence*$_{opt}$ "

s-char-sequence:

s-char

s-char-sequence s-char

s-char:

any member of the source character set except
the double-quote ", \\, or new-line character
escape-sequence

Zwei Dinge sind im Zusammenhang mit Zeichenketten zu beachten:

- Zeichenketten-Literale werden automatisch vom System um ein Zeichen ,,verlängert``: Sie werden durch das Zeichen mit dem ASCII-Wert 0 abgeschlossen, das das Ende der Zeichenkette kennzeichnet. Die oben angeführte Zeichenkette `Hallo` wird also vom Compiler als Folge von sechs Zeichen abgelegt, von denen das letzte das Zeichen `0x00` ist.
- Ferner ist zu beachten, daß aufeinanderfolgende Zeichenketten-Literale automatisch zu einem einzelnen Literal verkettet werden. So wird die Sequenz
  `"Hello" " " "world!"`
  automatisch zu
  `"Hello world!"`
  verkettet.

Beispiele für Zeichenketten-Literale sind: `"Hallo"`, `""` (Leere Zeichenkette) oder `"Ha \x41"` (Ha

A). Ein Beispiel für eine `wchar_t`-Zeichenkette ist etwa `L"Hallo"`, die durch die Zeichenfolge `L'H'`, `L'a'`, `L'l'`, `L'l'`, `L'o'`, `0x00` repräsentiert wird.

Vorige Seite: 3.4.4 Zeichenketten **Eine Ebene höher:** 3.4 Literale

3.4.5 Wahrheitswerte

Wahrheitswerte (Boolesche Werte, *Boolean*-Werte) umfassen die Werte „Wahr`` (`true` ) und „Falsch``(`false`).
`boolean-literal:`

`false`

`true`

Mit *Boolean*-Werten werden logische Sachverhalte dargestellt.

Vorige Seite: 3.4.5 Wahrheitswerte **Eine Ebene höher:** 3 Lexikalische Elemente von **Nächste Seite:** 3.5.1 Operatoren

---

3.5 Operatoren und Begrenzer

Operatoren und Begrenzer sind einzelne Sonderzeichen beziehungsweise Sequenzen von Sonderzeichen oder reservierte Wörter mit vordefinierter Bedeutung. Operatoren bestimmen Aktionen, die auf Programmobjekte ausgeführt werden können. Begrenzer wiederum trennen Symbole des Programmtexts voneinander.

Vorige Seite: 3.5 Operatoren und Begrenzer **Eine Ebene höher:** 3.5 Operatoren und Begrenzer **Nächste Seite:** 3.5.2 Begrenzer

3.5.1 Operatoren

Die Tabellen 3.3 und 3.2 zeigen die arithmetischen und logischen Operatoren von C++. Tabelle 3.4 führt weitere Operatoren mit Namen und Verwendung an. In den Tabellen steht *expr* für einen beliebigen Ausdruck (*Expression*), *lvalue* für einen beliebigen Ausdruck, der ein veränderbares Objekt repräsentiert, *className* für einen Klassen- oder Strukturnamen, *member* für ein Klassen- oder Strukturelement, *pointer* für einen beliebigen Zeiger, *type* für einen Typnamen, *exception* für eine Ausnahme und *preprocText* für einen beliebigen Präprozessortext.

Anmerkungen zu den beiden Tabellen:

- Die Schiebeoperatoren << und >> werden auch für die Ausgabe und Eingabe verwendet.
- Weitere Operatoren ergeben sich aus einer Kombination von arithmetischen beziehungsweise logischen Operatoren mit dem Zuweisungsoperator. Die Anweisung `lvalue += expr;` zum Beispiel entspricht der Anweisungskombination `lvalue = lvalue + expr;`. Es gibt folgende kombinierte Operatoren: *=, /=, %=, +=, -=, <<=, >>=, &=, |=, ^=.

Op	Name	Beispiel	Op	Name	Beispiel
++	Postinkrement	`lvalue++`	*	Multiplikation	`expr*expr`
++	Preinkrement	`++lvalue`	/	Division	`expr/expr`
--	Postdekrement	`lvalue--`	%	Modulo	`expr%expr`
--	Predekrement	`--lvalue`	+	Binäre Addition	`expr+expr`
+	Unäres Plus	`+expr`	-	Binäre Subtraktion	`expr-expr`
-	Unäres Minus	`-expr`			

Tabelle 3.2: Arithmetische Operatoren [Str92, S. 99f]

Op	Name	Beispiel	Op	Name	Beispiel
!	Logisches Nicht	`!expr`	&	Bitweises Und	`expr & expr`
&&	Logisches Und	`expr && expr`	|	Bitweises Oder (inkl.)	`expr | expr`
||	Logisches Oder (inkl.)	`expr || expr`	^	Bitweises Oder (exkl.)	`expr ^ expr`
~	Bit-Komplement	`~expr`	<<	Links-Schiebeoperation	`expr << expr`
<	Kleiner	`expr < expr`	>>	Rechts-Schiebeoperation	`expr >> expr`

>	Größer	`expr > expr`	<=	Kleiner-Gleich	`expr <= expr`
==	Gleichheitstest	`expr == expr`	>=	Größer-Gleich	`expr >= expr`
!=	Ungleichheitstest	`expr != expr`			

Tabelle: Logische Operatoren [Str92, S. 99f]

Op	Name	Beispiel
`::`	*Scope*-Operator	`scopeName::member`
`::`	Globaler *Scope*-Operator	`::name`
`.`	Mitgliedsoperator	`object.member`
`->`	Mitgliedsoperator (Zeiger)	`pointer->member`
`.*`	Element-Zeiger-Operator	`object.*pointerToMember`
`->*`	Element-Zeiger-Operator (Zeiger)	`pointer->*pointerToMember`
`new`	Allokierung von Objekten	`new type`
`new[]`	Allokierung von Objekten, Vektorform	`new type[expr]`
`delete`	Zerstören von Objekten	`delete pointer`
`delete[]`	Zerstören von Objekten, Vektorform	`delete[] pointer`
`[]`	Indexoperator	`pointer[expr]`
`()`	Funktionsaufrufoperator	`expr(exression_list);`
`()`	Konstruktionsoperator	`new type(expr_list);`
`sizeof`	Größenoperator	`sizeof(expr);`
`?:`	Konditionaler Audrucksoperator	`expr ? expr : expr;`
`&`	Adreßoperator	`&lvalue;`
`*`	Indirektionsoperator	`*expr;`
`()`	Typumwandlung	`(type)expr;` bzw. `type(expr);`
`=`	Zuweisung	`lvalue = expr`
`throw`	Auslöseoperator für Ausnahmen	`throw exception;`
`,`	Kommaoperator	`expr, expr`
#	Präprozessoroperator	`#preprocText`
##	Konkatenationsoperator	`preprocText ## preprocText`

Tabelle: Andere Operatoren [Str92, S. 99f], [ISO95]

Vorige Seite: 3.5 Operatoren und Begrenzer **Eine Ebene höher:** 3.5 Operatoren und Begrenzer **Nächste Seite:** 3.5.2 Begrenzer

Vorige Seite: 3.5.1 Operatoren **Eine Ebene höher:** 3.5 Operatoren und Begrenzer **Nächste Seite:** 3.5.3 Alternative Operatoren und

3.5.2 Begrenzer

Begrenzer trennen Symbole voneinander. C++ kennt die im folgenden angeführten Begrenzer, wobei zu beachten ist, daß einige Symbole mehrfach belegt sind. So existieren zum Beispiel sowohl ein Operator `*` als auch ein Begrenzer `*`.

Der *Strichpunkt* `;` dient als Terminator einer Anweisung. Jeder gültige C++-Ausdruck (auch der leere), der mit einem Strichpunkt endet, wird als Anweisung interpretiert.

Das *Komma* `,` trennt die Argumente in einer Funktionsparameterliste.

```
y = power(x, 12);
```

Runde Klammern `()` fassen Ausdrücke zusammen, isolieren konditionale Ausdrücke und begrenzen Funktionsparameterlisten:

```
a = 3 * (b + c);   // Umgehen der ueblichen Prioritaet
if (a < b)         // Konditionaler Ausdruck
   d = square(a);  // Aufruf der Funktion "square"
```

Eckige Klammern `[]` dienen zum Deklarieren von ein- und mehrdimensionalen Vektoren. Die eckigen Klammern sind in einem anderen Zusammenhang auch ein Operator (vergleiche Abschnitt 8.3), mit dem Vektoren indiziert werden.

```
char str[30];  // ein Vektor von 30 Zeichen
str[5] = 'a';  // Indizierung von str, Operator []
```

Der *Doppelpunkt* `:` kennzeichnet eine mit einer Sprungmarke (einem *Label*) versehene Anweisung.

```
...
start: x = 0;    // Sprungmarke start
...
goto start;      // Verzweigung zur Sprungmarke start
...
switch(a) {
   case 1:       // Sprungmarke 1 in switch
      a++;
      break;
   case 7:       // Sprungmarke 7 in switch
      a = b;
```

```
      break;
   case 39:     // Sprungmarke 39 in switch
      --a;
      break;
   default:     // Sprungmarke default in switch
      break;
   }
...
```

Geschweifte Klammern {} markieren Beginn und Ende eines Anweisungsblocks.

```
int main()
{
  ...
  if (x==0) {
    ..
  }
}
...
```

Ellipsen sind drei unmittelbar aufeinanderfolgende Punkte ohne *Whitespaces* dazwischen. Sie kennzeichnen in der formalen Parameterliste einer Funktion eine variable Anzahl von Argumenten oder Argumente mit beliebigen Typen.

Anmerkung: Als *Whitespaces* werden alle ,,Zwischenzeichen`` bezeichnet: Leerzeichen (*Blank*, *Space*), Tabulator (horizontal und vertikal), Zeilenvorschub und Seitenvorschub.

```
/* Die Funktion "printf" hat eine beliebige Anzahl von
 * Argumenten. Dies ist ihr Prototyp:
 */
int printf (const char* __format, ...);
```

Der *Stern* * in einer Variablendeklaration zeigt an, daß ein Zeiger auf ein Objekt deklariert wird.

```
char* string;  // Zeiger auf Variable des Typs char
int** a;       // Zeiger auf Zeiger auf int-Variable
```

Das *Referenzzeichen* & in einer Variablendeklaration zeigt an, daß ein Referenz-Objekt deklariert wird.

```
char& ch;  // char-Referenz
int&  a;   // int-Referenz
```

Das *Gleichheitszeichen* = trennt Variablendeklarationen von Initialisierungslisten. In der Parameterliste einer Funktion wiederum kennzeichnet das Gleichheitszeichen den Vorgabewert eines Parameters.

```
float pi=3.14159265; // Initialisierung von pi
int foo(int a=1);    // a hat Vorgabewert 1
```

- Eine Initialisierung einer Variablen ist *keine* Zuweisung! Zuweisung und Initialisierung erfolgen zwar mit dem Symbol =, im ersten Fall handelt es sich aber um den Begrenzer =, im zweiten Fall um den Operator =. Eine Initialisierung erfolgt beim Anlegen einer Variable, während sich eine Zuweisung immer auf ein bereits existierendes Objekt bezieht (mehr dazu in Abschnitt 11.7.1).

Vorige Seite: 3.5.1 Operatoren **Eine Ebene höher:** 3.5 Operatoren und Begrenzer **Nächste Seite:** 3.5.3 Alternative Operatoren und

Vorige Seite: 3.5.2 Begrenzer **Eine Ebene höher:** 3.5 Operatoren und Begrenzer **Nächste Seite:** 3.5.4 Kommentare

3.5.3 Alternative Operatoren und Begrenzer

Für verschiedene Operatoren und Begrenzer existieren alternative Symbole beziehungsweise sogenannte Trigraph-Sequenzen . Tabelle 3.5 zeigt die alternativen Schlüsselwörter, Tabelle 3.6 die Trigraph-Sequenzen, die bei der Übersetzung *vor* jedem anderen Symbol durch ihr eigentliches Symbol ersetzt werden.

Anmerkung: Die Trigraph-Sequenzen sowie die alternativen Symbole sind Bestandteil des Präprozessors (Anhang B) und nicht Teil des Sprachumfangs der eigentlichen Sprache C++. Sie werden aber im Zusammenhang mit den lexikalischen Elementen behandelt.

Symbol	Alternative	Symbol	Alternative	Symbol	Alternative
{	<%	}	%>	&=	and_eq
[	<:	]	:>	|=	or_eq
#	%:	##	%:%:	^=	xor_eq
&&	and	&	bitand	!=	not_eq
||	or	|	bitor	!	not
~	compl	^	xor		

Tabelle 3.5: Alternative Symbole für Operatoren und Begrenzer [ISO95, lex.key]

Symbol	Trigraph	Symbol	Trigraph	Symbol	Trigraph
#	??=	[	??(	{	??<
\	??/	]	??)	}	??>
^	??'	|	??!	~	??-

Tabelle 3.6: Trigraph-Sequenzen [ISO95, lex.digraph]

Vorige Seite: 3.5.2 Begrenzer **Eine Ebene höher:** 3.5 Operatoren und Begrenzer **Nächste Seite:** 3.5.4 Kommentare

Vorige Seite: 3.5.3 Alternative Operatoren und **Eine Ebene höher:** 3.5 Operatoren und Begrenzer

3.5.4 Kommentare

Kommentare sind Anmerkungen im Programmtext, die für den Leser des Programms bestimmt sind. Der Compiler ,,ignoriert`` sie und entfernt sie vor dem Übersetzen des Programms in Maschinencode aus dem Quelltext. In C++ gibt es zwei Möglichkeiten, Kommentare in den Quelltext einzufügen:

1. Einzeilige Kommentare: Die Zeichenfolge `//` leitet einen Kommentar ein, der sich bis zum Ende der Zeile, in der er auftritt, erstreckt. Der Kommentar kann an beliebiger Stelle beginnen.
2. Kommentare über mehrere Zeilen: Mit der Zeichenfolge `/*` beginnt ein Kommentar, der mit dem ersten darauffolgenden Zeichenpaar `*/` endet. Kommentare dieser Art können mehrere Zeilen lang sein. Es ist allerdings nicht möglich, derartige Kommentare zu ,,schachteln`` (ein geschachtelter Kommentar ist ein Kommentar im Kommentar).

```
foo(c, d);  // Ein einfacher C++-Kommentar
foo(c, d);  /* Ein Kommentar im C-Stil */

/*                // Auskommentieren von Codeteilen
foo(c,d);         // Die C++-Kommentare werden
++a;              // mit auskommentiert
*/

/* xy /* ab */*/  // Fehler! Geschachtelter Kommentar
                  // in C++ nicht moeglich, der
                  // Kommentar ist nach "ab */" zu Ende.
```

Grundsätzlich sind beide Arten von Kommentaren zulässig. Jedoch ist es üblich, einzeilige Kommentare zu verwenden. Mehrzeilige Kommentare werden zum Auskommentieren von Codeteilen benutzt.

Vorige Seite: 3.5.4 Kommentare **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** 4.1 Die Begriffe Definition

4 Einfache Deklarationen und Basisdatentypen

Dieses Kapitel zeigt, wie in C++ auf einfache Art Variablen und Konstanten vereinbart werden. Ferner werden die grundlegenden Datentypen von C++ (,,Basisdatentypen``) erläutert.

Vorige Seite: 4 Einfache Deklarationen und **Eine Ebene höher:** 4 Einfache Deklarationen und **Nächste Seite:** 4.2 Basisdatentypen

4.1 Die Begriffe Definition und Deklaration

In C++ müssen alle Objekte vor ihrer Verwendung *deklariert* werden. Eine Deklaration gibt dem Compiler einen Namen (Bezeichner) bekannt und verbindet diesen Namen mit einem Typ. Der Typ gibt Auskunft über die „Art`` des Bezeichners und bestimmt damit implizit die Operationen, die für das deklarierte Objekt zulässig sind. Eine Deklaration eines Objekts als Zahl zum Beispiel bedeutet, daß unter anderem die Operationen `+`, `-`, `/` etc. auf dieses Objekt angewandt werden können. Einfache Deklarationen sind von der Art *T identifier*, wobei *T* ein Datentyp und *identifier* ein Bezeichner ist. Auch Deklarationen der Art *T identifierList* sind möglich, wobei *identifierList* eine Liste von durch Beistriche getrennten Bezeichnern ist:

```
int  i;        // i ist eine ganze Zahl
char ch;       // ch ist ein Zeichen
int  j, k, l;  // j, k und l sind ganze Zahlen
```

Die Begriffe Deklaration und Definition werden oft synonym verwendet. Sie bezeichnen aber verschiedene Dinge: Eine Deklaration führt einen oder mehrere Namen in einem Programm ein [ES90, S. 13]. Dem Compiler werden zwar mit dem Namen Informationen über einen Typ oder eine Funktion bekanntgegeben, es wird aber kein Programmcode erzeugt oder Speicherplatz für ein Objekt angelegt.

Eine Definition wiederum vereinbart konkrete Objekte im Programm, also Variablen (inklusive deren Speicherplatz) oder ausführbaren Code. Jede Definition ist damit zugleich eine Deklaration. Ebenso sind sehr viele Deklarationen zugleich Definitionen: Die Deklaration einer Variablen vom Typ `int` in der oben angeführten Art gibt den Namen des Objekts bekannt und vereinbart auch Speicherplatz für das Objekt. Andererseits kann der Name einer Variablen auch vereinbart werden, ohne daß ein konkretes Objekt angelegt wird (siehe Abschnitt 9.2.1, Speicherklassenattribut `extern`). Ähnlich ist die Situation in bezug auf Klassen und Funktionen: Die Deklaration gibt Auskunft über Name und Art eines Bezeichners, aber erst die Definition vereinbart ein konkretes Objekt.

Daraus folgt, daß es für jedes Objekt eines Programms beliebig viele Deklarationen (die sich nicht widersprechen dürfen) geben kann, aber jeweils nur eine Definition.

C++ unterscheidet sich in bezug auf Deklarationen von C: Im Unterschied zu C sind Deklarationen in C++ Anweisungen und können daher überall im Programmtext stehen.

Deklarationen können überall im Programmtext stehen, sollten allerdings (bis auf wenige Ausnahmen) mit Rücksicht auf die Lesbarkeit und Übersichtlichkeit vor den anderen Anweisungen codiert werden.

Vorige Seite: 4 Einfache Deklarationen und **Eine Ebene höher:** 4 Einfache Deklarationen und **Nächste Seite:** 4.2 Basisdatentypen

Vorige Seite: 4.1 Die Begriffe Definition **Eine Ebene höher:** 4 Einfache Deklarationen und **Nächste Seite:** 4.2.1 Der Datentyp `bool`

4.2 Basisdatentypen

Basisdatentypen sind vordefinierte einfache Datentypen. Sie umfassen Wahrheitswerte (`bool`), Zahlen (`int`, `short int`, `long int`, `float`, `double` und `long double`), Zeichen (`char`, `wchar_t`) und den Typ ,,nichts`` (`void`). Daneben gibt [ISO95] auch den Typ `enum` als einen *fundamental type* an. Ein `enum` ist zwar kein vordefinierter Typ, wird aber aus Gründen der Konformität ebenfalls hier angeführt.

Einige dieser Typen können auch über andere Namen, sogenannte Typspezifikatoren, angegeben werden. Tabelle 4.1 zeigt die Basisdatentypen, die auch über alternative Typnamen spezifiziert werden können.

Typ	Spezifikatoren
int	`int`, `signed`, `signed int`
unsigned int	`unsigned int`, `unsigned`
short int	`short int`, `short`,
	`signed short`, `signed short int`
unsigned short int	`unsigned short int`, `unsigned short`
long int	`long int`, `long`, `signed long`, `signed long int`
unsigned long int	`unsigned long int`, `unsigned long`

Tabelle 4.1: Typen und ihre alternativen Spezifikatoren [ISO95, dcl-type.simple]

Im folgenden werden die (üblichen) einfachen Namen `short` und `long` anstelle ihrer ,,korrekten`` Typnamen `short int` und `long int` verwendet.

Die Typen `bool`, `char`, `wchar_t` und alle `int`-Typen werden auch als *Integrale Typen* oder Integer-Typen bezeichnet.

Vorige Seite: 4.1 Die Begriffe Definition **Eine Ebene höher:** 4 Einfache Deklarationen und **Nächste Seite:** 4.2.1 Der Datentyp `bool`

Vorige Seite: 4.2 Basisdatentypen **Eine Ebene höher:** 4.2 Basisdatentypen **Nächste Seite:** 4.2.2 Die Datentypen `char` und `wchar_t`

4.2.1 Der Datentyp `bool`

Mit dem Datentyp `bool` werden Wahrheitswerte beschrieben. Sein Wertebereich umfaßt die beiden Werte `false` und `true`, die die Werte ,,Falsch`` und ,,Wahr`` ausdrücken.

Tabelle 4.2 zeigt die Ergebnisse der logischen Operationen &&, `||` und `!`. && repräsentiert die logische Und-Verknüpfung, `||` die logische inklusive Oder-Verküpfung und `!` die Negation.

X	Y	X && Y	X || Y	!X
`true`	`true`	`true`	`true`	`false`
`true`	`false`	`false`	`true`	`false`
`false`	`true`	`false`	`true`	`true`
`false`	`false`	`false`	`false`	`true`

Tabelle 4.2: Logische Operationen

Vorige Seite: 4.2 Basisdatentypen **Eine Ebene höher:** 4.2 Basisdatentypen **Nächste Seite:** 4.2.2 Die Datentypen `char` und `wchar_t`

Vorige Seite: 4.2.1 Der Datentyp `bool` **Eine Ebene höher:** 4.2 Basisdatentypen **Nächste Seite:** 4.2.3 Die `int`-Datentypen

---

4.2.2 Die Datentypen `char` und `wchar_t`

Der grundlegende Zeichentyp ist `char`. Er belegt ein Byte Speicherplatz und besitzt somit 256 Ausprägungen. Im Zeichendatentyp findet der gesamte erweiterte ASCII-Zeichensatz Platz. Da im ASCII-Zeichensatz alle Zeichen geordnet sind, lassen sich auch Vergleiche zwischen einzelnen Zeichen durchführen. Anhand des ASCII-Zeichensatzes können auch Umwandlungen von Zeichen in Zahlen und umgekehrt vorgenommen werden. Das Zeichen `'A'` zum Beispiel entspricht der Zahl 65.

Der Zeichendatentyp kann explizit als `signed` oder `unsigned` spezifiziert werden. Zu beachten ist, daß `char`, `unsigned char` und `signed char` drei verschiedene Typen sind. Ob ein `char` negative Zahlen aufnehmen kann (Wertebereich von -128 bis +127), hängt von der jeweiligen Compiler-Implementierung ab.

```
char           ch;   // Wertebereich zw. -128 und 127
unsigned char  uch;  // Wertebereich zw. 0 und 255
wchar_t        mch;
ch = 'A';
uch = 'a';
mch = L'X';
```

Vorige Seite: 4.2.2 Die Datentypen `char` und `wchar_t` **Eine Ebene höher:** 4.2 Basisdatentypen
Nächste Seite: 4.2.4 Fließkommadatentypen

4.2.3 Die `int`-Datentypen

Der grundlegende Ganzzahldatentyp ist `int`. Der Wertebereich von `int`-Typen ist durch die Größe eines Maschinenwortes des jeweiligen Rechnersystems bestimmt. Auf 16-Bit-Systemen (zum Beispiel DOS) weisen `int`-Variablen 2 Bytes auf, auf 32-Bit-Systemen 4. Integervariablen besitzen demnach 65536 beziehungsweise 4294967296 (=2 hoch 32) Ausprägungen.

Die Familie der `int`-Datentypen umfaßt neben `int` noch die Datentypen `short` und `long`:

- `short`
  Der Typ `short` umfaßt *mindestens* den Wertebereich des Typs `char` und *maximal* den des Typs `int`. In der Regel weisen Variablen vom Typ `short` zwei Bytes Ausdehnung auf und haben damit einen Wertebereich von -32768 bis +32767 (beziehungsweise von 0 bis 65535).
- `long`
  In vielen Fällen ist der Wertebereich von `int` zu gering. `long` verfügt über einen *mindestens* so großen Wertebereich wie `int`. Auf 16- und 32-Bit-Systemen belegen `long`-Variablen in der Regel vier Bytes Speicherplatz, was 4294967296 (=2 hoch 32) Ausprägungen entspricht.

Jeder der drei `int`-Typen kann zusätzlich als `signed` oder `unsigned` spezifiziert werden. Die `signed`-Datentypen weisen dieselbe Ausdehnung wie die `unsigned`-Typen auf und verfügen über einen entsprechend veränderten Wertebereich. Im folgenden sind (übliche) Beispiele für ganzzahlige Variablen auf 16-Bit-Systemen angeführt:

```
int x;             // -32768 <= x <= 32767
signed int x;      // -32768 <= x <= 32767
unsigned x;        // 0 <= x <= 65535
short x;           // -32768 <= x <= 32767
signed short x;    // -32768 <= x <= 32767
long x;            // -2147483648 <= x <= 2147483647
signed long x;     // -2147483648 <= x <= 2147483647
unsigned long x;   // 0 <= x <= 4294967296
```

- Die Größe der Basisdatentypen `int`, `short` und `long` ist *nicht* festgelegt. `int` weist die Größe eines Maschinenworts auf und ist mindestens so groß wie `short`, `long` hat mindestens die Ausdehnung von `int`.

Vorige Seite: 4.2.2 Die Datentypen `char` und `wchar_t` **Eine Ebene höher:** 4.2 Basisdatentypen
Nächste Seite: 4.2.4 Fließkommadatentypen

Vorige Seite: 4.2.3 Die `int`-Datentypen **Eine Ebene höher:** 4.2 Basisdatentypen **Nächste Seite:** 4.2.5 Der Datentyp `void`

4.2.4 Fließkommadatentypen

Der Basis-Fließkommadatentyp ist `float` und nimmt reelle Zahlen auf. Reelle Zahlen werden (vereinfacht gesagt) durch eine ganze Zahl, die sogenannte *Mantisse* , und die Lage des Dezimalpunkts, bestimmt durch den Exponenten, dargestellt.

Die Genauigkeit von Fließkommadatentypen ergibt sich naturgemäß aus der Größe dieser Mantisse und ist compilerabhängig. Reicht die Genauigkeit von `float` nicht aus, so kann auf zwei andere Datentypen zurückgegriffen werden: `double`, der mindestens die Genauigkeit von `float` aufweist, und `long double`, der mindestens die Genauigkeit von `double` aufweist.

Genaue Angaben bezüglich der Größe und Genauigkeit von Datentypen sind in der Datei `limits` des jeweiligen Entwicklungssystems angeführt.

```
float pi;
double erdanziehung;
long double x;
long double y;

pi = 3.141592;
erdanziehung = 0.980665e1;
x = -123.456789012345678E-1234;
y = 2.0E500;
```

Vorige Seite: 4.2.4 Fließkommadatentypen **Eine Ebene höher:** 4.2 Basisdatentypen

4.2.5 Der Datentyp `void`

`void` ist ein spezieller Typ, der anzeigt, daß kein Wert vorhanden ist. Es ist nicht möglich, ein Objekt vom Typ `void` anzulegen. Vielmehr findet der Datentyp Anwendung bei der Deklaration von speziellen Zeigern, von denen nicht bekannt ist, auf welchen Typ sie verweisen (Abschnitt 8.2.7), oder bei Funktionen, die keinen Rückgabewert liefern (Abschnitt 7.4.1).

```
void   initData();  // Funktion ohne Rueckgabewert
void*  ptr;         // Zeiger ohne konkrete Typangabe
```

Vorige Seite: 4.2.5 Der Datentyp `void` **Eine Ebene höher:** 4 Einfache Deklarationen und **Nächste Seite:** 4.4 Deklaration von Konstanten

---

4.3 Aufzählungstypen

Enumerationstypen werden auch als Aufzählungstypen bezeichnet. Ein Aufzählungstyp beschreibt seine Wertemenge durch Aufzählung all seiner Werte. Die Grammatik gibt den Typ `enum` wie folgt an:

```
enum-specifier:
                enum identifieropt  enumerator-listopt
enumerator-list:
                enumerator-definition
                enumerator-list , enumerator-definition
enumerator-definition:
                enumerator
                enumerator = constant-expression
enumerator:
                identifier
```

Alle in `enumerator-list` angegebenen Namen sind Bezeichner für die Werte des Enumerationstyps. Folgendes Beispiel vereinbart einen Aufzählungsdatentyp für Wochentage:

```
enum Weekday {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
Weekday  e;
e = Sat;
```

Den einzelnen Werten eines Enumerationstyps werden der Reihe nach Zahlen zugeordnet. Standardmäßig ist der erste Wert 0. Die darauffolgenden Werte sind (aufsteigend) 1, 2, 3 Durch Zuweisung einer Konstanten ergibt sich die Möglichkeit, diese Standardnumerierung zu ändern:

```
enum Color {Red=1, Blue, White=29, Black};
```

`Red` belegt hier den Wert 1, `Blue` den Wert 1+1=2, `White` den Wert 29 und `black` den Wert 29+1=30.

Jeder Enumerationseintrag kann durch integrale Promotion (siehe Abschnitt 9.4.1) in den zugehörigen `int`-Wert umgewandelt werden. Umgekehrt ist dies nicht möglich - hier muß eine explizite Typumwandlung (siehe Abschnitt 9.4.2) stattfinden:

```
Color col = 2;    // Fehler, keine automatische
                  // Konvertierung int->Color
int i = col;      // i=2, ok, integrale Promotion
```

Vorige Seite: 4.3 Aufzählungstypen **Eine Ebene höher:** 4 Einfache Deklarationen und **Nächste Seite:** 5 Ausdrücke

4.4 Deklaration von Konstanten

Eine Konstante ist ein Objekt, das seinen Wert während des gesamten Programmablaufes nicht ändert. Ein Beispiel für einen derartigen Wert ist etwa in einem Programm zur Adreßverwaltung die maximale Anzahl der Adressen. Die Konstante wird bei der Deklaration der Variablen, den verschiedenen Schleifendurchläufen u.ä. anstelle eines Literals verwendet.

Wird keine Konstante, sondern ein Literal verwendet, so zieht jede Änderung der konstanten Größen eine Änderung aller derartigen Literale im Programm nach.

Wird hingegen eine Konstante verwendet, so kann der Wert an einer Stelle im Source-Code gesetzt werden und es besteht die Sicherheit, daß der Wert im Laufe des Programms nicht verändert wird.

Konstante werden deklariert, indem vor dem eigentlichen Typ das Schlüsselwort `const` notiert wird:

```
const int val = 12;
```

Wird versucht, während der Programmausführung der Konstante `val` einen Wert zuzuweisen, so führt dies zu einem Übersetzungsfehler.

`const` führt sowohl zu einer erhöhten Lesbarkeit und Wartbarkeit von Programmen als auch zu einer wesentlich höheren Programmsicherheit: `const`-Werte können nicht (oder nur umständlich) vom Programmierer verändert werden, sie können innerhalb von Funktionen auf den Rückgabewert verweisen oder auf eine Funktion als Ganzes.

Eine Zeichenkonstante besitzt den Integer-Wert des in Hochkommas eingeschlossenen Zeichens. Zeichenkonstante können außerdem Steuerzeichen aufnehmen, die sogenannten *Escape*-Sequenzen. Diese werden mit dem Zeichen \ eingeleitet. Um das Zeichen \ selbst darzustellen, wird \\ verwendet.

```
const char ersterBuchstabe = 'A';      // Zeichen A
const char anfuehrungszeichen = '"';   // Anf.Zeichen
const char backslash = '\\';           // Zeichen \
const char alert = '\a';               // Alarmton
const char my = '\230';                // Zeichen mit
                                       // ASCII-Wert 230
const char nullZeichen = 0x00;         // Nullzeichen
```

Beispiele für Zahlenkonstanten sind etwa:

```
const short pensionsalter = 65;
const int   nrOfCycles = 16384;
const long  lichtgeschwindigkeit = 299792458;
```

Beispiele für Fließkommakonstanten sind:

```
const float pi = 3.141592;
const double stilb = 10000.;
const double parsec = 3.0856782E16;
```

- `const` sollte in C++-Programmen verwendet werden, wo immer dies möglich ist. Kandidaten für die Verwendung von `const` sind Variablen oder Objekte, deren Wert nicht geändert werden darf, konstante Zeiger, Eingangsparameter usw.

Eine Anmerkung zu den von C bekannten `#define`-Konstanten: Durch die Angabe des Präprozessor-Schlüsselworts `#define` (siehe Anhang B.1) können sogenannte „Präprozessor-Konstante`` vereinbart werden. Diese sind nichts anderes als ein Text, der *vor* dem eigentlichen Übersetzungsvorgang durch einen anderen ersetzt wird. Dieser Ersetzungsvorgang findet nicht im Rahmen der C++-Übersetzung statt und umgeht damit Syntaxprüfung und Typprüfungen. In C ist die Verwendung von `#define` üblich, weil es lange Zeit keine andere Möglichkeit zur Vereinbarung von Konstanten gab. In C++ ist dies nicht der Fall, daher sollte die Verwendung von `#define` zur Vereinbarung von Konstanten unbedingt unterbleiben.

- Bei der Vereinbarung von Konstanten durch `#define` erfolgt nur eine textuelle Ersetzung. Dies ist bedenklich und sollte in C++ keinesfalls verwendet werden. In C++ werden Konstante ausschließlich durch die Verwendung von `const` vereinbart.

Vorige Seite: 4.3 Aufzählungstypen **Eine Ebene höher:** 4 Einfache Deklarationen und **Nächste Seite:** 5 Ausdrücke

Vorige Seite: 4.4 Deklaration von Konstanten **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** 5.1 Auswertungsreihenfolge

5 Ausdrücke

Ähnlich wie mathematische Ausdrücke stellen auch Ausdrücke in C++ Berechnungen dar und bestehen aus Operanden und Operatoren. Operanden können Variablen, Konstanten oder wiederum Ausdrücke sein. Die Auswertung jedes Ausdrucks liefert einen Wert, der sich aus der Verknüpfung von Operanden durch Operatoren ergibt.

Grundsätzlich kann man zwischen folgenden Klassen von Ausdrücken unterscheiden:

- Arithmetische Ausdrücke
  Arithmetische Ausdrücke sind alle Ausdrücke, deren Ergebnis als Skalar geschrieben werden kann (`char`-, `int`- oder `float`-Typen). Ein arithmetischer Ausdruck, der ein Ergebnis von integralem Typ liefert, wird auch als Integer-Ausdruck bezeichnet.
- Logische Ausdrücke
  Logische Ausdrücke sind Ausdrücke, die Wahrheitswerte beschreiben. Sie entstehen durch Vergleiche oder logische Verknüpfungen.
- Andere Ausdrücke
  Die restlichen Ausdrücke werden unter dem Sammelbegriff ,,Andere Ausdrücke`` zusammengefaßt. Darunter fallen Typumwandlungen (*Cast*-Ausdrücke) ebenso wie `typeid`-Ausdrücke.

Im folgenden werden die Begriffe *Auswertungsreihenfolge* (Priorität und Assoziativität), *LValue* und *RValue* erläutert. Anschließend werden die verschiedenen Arten von Ausdrücken besprochen: Syntaktisch wird zwischen vier verschiedenen Arten unterschieden: *Primäre Ausdrücke*, *Postfix-Ausdrücke*, *Unäre Ausdrücke* und *andere Ausdrücke*.

Vorige Seite: 5 Ausdrücke **Eine Ebene höher:** 5 Ausdrücke **Nächste Seite:** 5.2 *LValues* und *RValues*

5.1 Auswertungsreihenfolge

Alle Ausdrücke werden nach bestimmten Regeln ausgewertet. Maßgeblich für die Art der Auswertung sind dabei *Assoziativität* und *Priorität* der Operatoren.

Die Assoziativität gibt Auskunft über die Auswertungsreihenfolge der Operanden eines Ausdrucks. So wird zum Beispiel im Ausdruck `p++` zuerst `p` ausgewertet und dann die linke Seite des Operators `++` (`p`) erhöht, während der Ausdruck `++p` zuerst `p` erhöht und dann den Ausdruck auswertet. Das folgende Codestück gibt `6, 8` aus.

```
int i = 6;
cout << i++;  // gibt i (= 6) aus und erhoeht danach i
              // (= 6+1)
cout << ++i;  // erhoeht i (= 7+1) und gibt i dann aus
```

Die Priorität von Operatoren wiederum gibt an, in welcher Reihenfolge die verschiedenen Operanden eines Ausdrucks ausgewertet werden. In C++ weisen zum Beispiel die multiplikativen Operatoren `*` und `/` (so wie in der Mathematik) eine höhere Priorität als die additiven Operatoren `+` und `-` auf (3+4*5 versus (3+4)*5).

In Tabelle 5.1 werden die Regeln für Priorität und Assoziativität der Operatoren zusammengefaßt. Die Operatoren in einer Zeile besitzen die gleiche Priorität, die insgesamt von oben nach unten abnimmt. So haben `*`, `/` und `%` zum Beispiel die gleiche Priorität, die höher ist als die der Operatoren `+` und `-`. Der Operator `this` ist nicht angeführt.

Priorität	Assoziativität	Operatoren
17	rechts	`::`(global)
	links	`::`(Klassen-Scope)
16	links	`[] -> . ()`(Konstruktions-
		und Funktionsaufrufoperator)
	rechts	`dynamic_cast static_cast const_cast`
		`typeid reinterpret_cast`
	-	`++ --` (jeweils Postfix)
15	rechts	`! ~ ++ -- - sizeof ()`(*Cast*-Operator)
		&(Adreßoperator) *(Indirektionsoperator)

		`new new[] delete delete[]`
14	links	`->* .*`
13	links	* (Multiplikation) / %
12	links	+ -
11	links	<< >>
10	links	< <= > >=
9	links	== !=
8	links	& (Bitweises Und)
7	links	^
6	links	|
5	links	&&
4	links	||
3	links	?:
2	rechts	= += -= *= /= %= <<= >>= &= |=^=
1	links	,

Tabelle 5.1: Operatoren: Priorität und Assoziativität (nach [Lip91, S. 87])

Einige ,,Besonderheiten`` sind im folgenden angeführt:

- Die Priorität des Vergleichsoperators == ist höher als die des Bit-orientierten Operators &, woraus die Notwendigkeit folgt, den &-Ausdruck im Konstrukt

  ```
  if((statusReg & display) == Bitpos0)
  ```

  zu klammern.
- Zuweisungsoperatoren (=, +=, /= etc.) sind rechtsassoziativ, das heißt der Ausdruck

  ```
  x = y = z = 0;
  ```

  wird behandelt, als wäre er folgendermaßen geklammert:

  ```
  (x = (y = (z = 0)));
  ```
- Ein Ausdruck mit den linksassoziativen Operatoren && und || unterschiedlicher Priorität

  ```
  x = x && y || z;
  ```

  entspricht folgendem geklammerten Ausdruck:

  ```
  (x = ((x && y) || z));
  ```

Um sicherzustellen, daß die Auswertung des Ausdrucks in der gewünschten Form durchgeführt wird und um das Lesen eines Programms zu erleichtern, sollten bei komplexeren Ausdrücken Klammern verwendet werden.

Sieht man von der Priorität der Operatoren ab, so ist es vom jeweiligen Compiler abhängig, in welcher Reihenfolge die Auswertung der Operanden eines Operators erfolgt. Das kann insbesondere dann zu Problemen führen, wenn diese Ausdrücke Nebeneffekte enthalten wie sie zum Beispiel durch Inkrement- oder Dekrement-Operatoren verursacht werden können. Ein typisches Beispiel hierfür ist:

```
x = f(y) + g(y++);
```

Es ist nicht definiert, ob zuerst die Funktion `f` oder die Funktion `g` ausgeführt wird. Demzufolge ist nicht klar, ob der Aufruf von `f` mit dem inkrementierten Argumentwert `y` oder dem ursprünglichen Wert von `y` erfolgt.

Analog ist auch die Reihenfolge, in der die Argumente für einen Funktionsaufruf bewertet werden, nicht definiert. Daher kann die Anweisung

```
printf("%d %d\n", ++n, pot(2,n));
```

verschiedene Resultate liefern, abhängig davon, ob `n` inkrementiert wird, bevor `pot` aufgerufen wird. Der Programmierer sollte es daher grundsätzlich unterlassen, Ausdrücke zu verwenden, die von der Reihenfolge der Auswertung abhängen.

Vorige Seite: 5 Ausdrücke **Eine Ebene höher:** 5 Ausdrücke **Nächste Seite:** 5.2 *LValues* und *RValues*

Vorige Seite: 5.1 Auswertungsreihenfolge **Eine Ebene höher:** 5 Ausdrücke **Nächste Seite:** 5.3 Primäre Ausdrücke

---

5.2 *LValues* und *RValues*

Jeder Ausdruck ist entweder ein sogenannter *Left Value* (kurz *LValue*) oder ein *Right Value* (kurz *RValue*). Vereinfacht gesagt, ist ein *LValue* ein Ausdruck, der auf der linken Seite einer Zuweisung stehen kann, also das Ziel einer Zuweisung oder ,,Wertveränderung`` sein kann. Ein *LValue* ist entweder ein Objekt oder eine Funktion (mehr dazu im Zusammenhang mit Referenzen in Abschnitt 8.1).

Alle Ausdrücke, die nicht als *LValue* bezeichnet werden, sind folglich *RValues*. Verschiedene Operanden verlangen einen *LValue* (zum Beispiel der Zuweisungsoperator auf seiner linken Seite), andere wiederum verlangen einen *RValue* (zum Beispiel +). Jeder *LValue* kann aber - falls dies nötig ist - auch als *RValue* betrachtet werden (siehe Abschnitt 9.4.1).

Vorige Seite: 5.2 *LValues* und *RValues* **Eine Ebene höher:** 5 Ausdrücke **Nächste Seite:** 5.4 Postfix-Ausdrücke

5.3 Primäre Ausdrücke

Primäre Ausdrücke haben bei der Auswertung die höchste Priorität. Sie sind wie folgt aufgebaut [ISO95, expr.prim]:

```
primary-expression:
                literal
                this
                :: identifier
                :: operator-function-id
                :: qualified-id
                ( expression )
                id-expression
```

Literale wurden bereits dargelegt. `this` ist ein Schlüsselwort, das einen Zeiger auf das aktuelle Objekt liefert und wird im Zusammenhang mit Klassen (Abschnitt 11) erläutert. `::`-Ausdrücke erlauben den Zugriff auf Objekte eines bestimmten Gültigkeitsbereichs und werden ebenfalls später (Abschnitt 11.9.3) näher erläutert. Geklammerte Ausdrücke sind Ausdrücke, deren Werte denen der ,,enthaltenen`` Ausdrücke entspricht. Die Klammerung legt lediglich die Auswertungsreihenfolge fest.

Die letzte Form von primären Ausdrücken (`Id-Ausdruck`) wird im Zusammenhang mit Klassen nach den Operatoren `.*` und `->*` verwendet (siehe Abschnitt 11.8.1).

Vorige Seite: 5.3 Primäre Ausdrücke **Eine Ebene höher:** 5 Ausdrücke **Nächste Seite:** 5.5 Unäre Ausdrücke

5.4 Postfix-Ausdrücke

Postfix-Ausdrücke werden, wie in Tabelle 5.1 angeführt, von rechts nach links ausgewertet.

```
postfix-expression:
                primary-expression
                postfix-expression [ expression ]
                postfix-expression ( expression-listopt )
                simple-type-specifier ( expression-listopt )
                postfix-expression . templateopt id-expression
                postfix-expression -> templateopt id-expression
                postfix-expression ++
                postfix-expression --
                dynamic_cast < type-id > ( expression )
                static_cast < type-id > ( expression )
                reinterpret_cast < type-id > ( expression )
                const_cast < type-id > ( expression )
                typeid ( expression )
                typeid ( type-id )
expression-list:
                assignment-expression
                expression-list , assignment-expression
```

Die eckigen Klammern werden in C++ zur Indizierung von Vektoren verwendet und bilden den sogenannten Indexoperator. Indexoperatoren werden in Abschnitt 8.3 im Zusammenhang mit Vektoren besprochen.

Runde Klammern dienen neben dem Zusammenfassen von Ausdrücken und dem Isolieren konditionaler Ausdrücke auch als Funktionsaufrufoperator (siehe Abschnitt 7.3). Funktionen sind Zusammenfassungen von Anweisungen zu einer Art „Verbundanweisung``, die über den Funktionsaufrufoperator `()` aufgerufen wird. Sie werden im Zusammenhang mit Funktionen in Kapitel 7 besprochen.

Neben den bereits oben beschriebenen Bedeutungen dienen runde Klammern, die einer Typangabe folgen, auch als Typumwandlungsoperatoren. Mit ihrer Hilfe können Ausdrücke eines Typs in Ausdrücke eines anderen Typs umgewandelt werden. Typumwandlungsoperatoren werden in Abschnitt 9.4 im Zusammenhang mit den verschiedenen anderen Arten der Typumwandlung erläutert.

Für den direkten Zugriff auf Elemente von Klassen und Objekten verwendet man den Operator `.` beziehungsweise den Operator `->`. Beide werden im Zusammenhang mit Klassen in Abschnitt 8.6 besprochen.

Die Postfix-Operatoren `++` beziehungsweise `--` dienen zur Inkrementierung beziehungsweise Dekrementierung eines Operanden um den Wert eins. Beide Operatoren können nur auf einen Operanden angewandt werden, der sich auf einen modifizierbaren Speicherbereich (*LValue*) bezieht und von skalarem Typ (arithmetischer Typ oder Zeiger) ist.

Zu beachten ist, daß neben den hier besprochenen Postfix-Versionen der beiden Operatoren auch Präfix-Versionen existieren (siehe Abschnitt 5.5). Obwohl der Postfix-Inkrement-Operator den Wert eines Ausdrucks um eins erhöht, ist der Wert des Ausdrucks *postfix-expression* `++` gleich dem Wert des Ausdrucks *vor* dem Erhöhen (tatsächlich ist der Wert sogar eine Kopie des ursprünglichen Ausdrucks). Die Semantik kann damit auch wie folgt umschrieben werden: *postfix-expression* wird um eins erhöht (beziehungsweise erniedrigt), *nachdem* der aktuelle Ausdruck ausgewertet wurde.

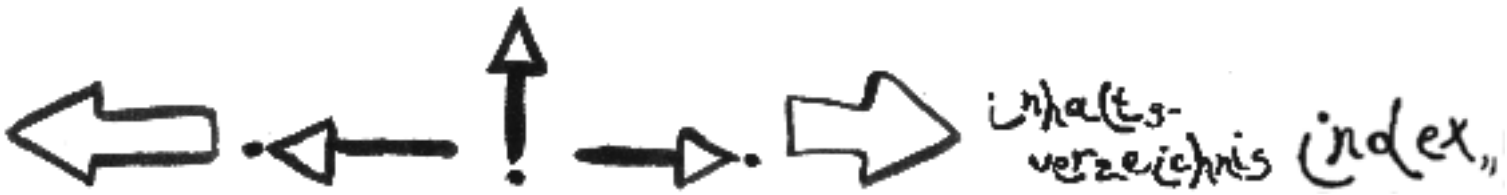
Dasselbe gilt entsprechend für den Postfix-Dekrement-Operator.

Beispiele:

```
int n, x, y;
...
n = 5;    // n wird mit 5 initialisiert
x = n++;  // x erhaelt den Wert 5, n den Wert 6 (= 5+1)
z = x--;  // z erhaelt den Wert von x (= 5), x wird
          // anschliessend dekrementiert (x = 4)
```

Mit dem `typeid`-Operator kann der Typ eines Ausdrucks festgestellt werden. `typeid` wird im Zusammenhang mit Objekten benutzt, um deren Typ zur Laufzeit zu erfahren. Der Operator wird im Zusammenhang mit dem Laufzeit-Typinformationssystem (*Run-Time Type Infomation System*) besprochen (Abschnitt 14.6).

Neben dem Postfix-Operator `()` existieren noch andere Möglichkeiten zur Typumwandlung von Ausdrücken mit den Operatoren `static_cast`, `dynamic_cast`, `reinterpret_cast` und `const_cast`. Sie werden, wie der Typumwandlungsoperator `()`, in Abschnitt 9.4.2 besprochen.



Vorige Seite: 5.3 Primäre Ausdrücke **Eine Ebene höher:** 5 Ausdrücke **Nächste Seite:** 5.5 Unäre Ausdrücke

Vorige Seite: 5.4 Postfix-Ausdrücke **Eine Ebene höher:** 5 Ausdrücke **Nächste Seite:** 5.6 Andere Ausdrücke

5.5 Unäre Ausdrücke

Unäre Ausdrücke werden von links nach rechts ausgewertet.

```
unary-expression:
                postfix-expression
                ++  unary-expression
                --  unary-expression
                unary-operator cast-expression
                sizeof unary-expression
                sizeof ( type-id )
                new-expression
                delete-expression
unary-operator: one of
                *  &  +  -  !  ~
```

Unäre Ausdrücke können neben den bereits besprochenen Postfix-Ausdrücken auch aus Präfix-Inkrement- beziehungsweise Präfix-Dekrement-Operatoren gefolgt von unären Ausdrücken bestehen.

Die Präfix-Operatoren `++` beziehungsweise `--` dienen zur Inkrementierung beziehungsweise Dekrementierung eines Operanden um den Wert eins. Beide Operatoren können nur auf Operanden angewandt werden, die sich auf modifizierbare Speicherbereiche beziehen und von skalarem Typ (arithmetischer Typ oder Zeiger) sind.

Im Unterschied zu den oben angeführten Postfix-Inkrement- beziehungsweise Postfix-Dekrement-Operatoren gilt hier allerdings, daß der Wert von *unary-expression* gleich dem Wert des Ausdrucks *nach* der Inkrementierung beziehungsweise Dekrementierung ist.

```
int n, x, y;
...
n = 5;          // n wird mit 5 initialisiert
x = n++;        // x = 5, n = 6 (=5+1)
y = ++n;        // n und y sind 7 (=6+1)
z = --x + ++y;  // z = x (=4) + y (=7+1),
```

Mit dem unären Indirektionsoperator * kann eine Adreßbezugnahme aufgelöst werden, das heißt ein Zeiger wird ,,dereferenziert``.

Das ,,Gegenstück`` zum Indirektionsoperator stellt der Adreßoperator & dar. Er liefert die Speicheradresse eines Objekts (Variable oder Funktion). Beide Operatoren werden im Zusammenhang mit dynamischen Datenstrukturen näher erläutert (Abschnitt 8.2).

Die beiden unären Plus-/Minus-Operatoren + und - können ausschließlich auf arithmetische Typen angewandt werden.

Der unäre --Operator liefert den negativen Wert seines Operanden. Dabei finden die üblichen arithmetischen Umwandlungen statt. Der negative Wert eines `unsigned`-Werts wird bestimmt, indem der Wert von 2 hoch *n* subtrahiert wird, wobei *n* die Anzahl der Bits darstellt, die zur Repräsentation des ursprünglichen Ausdrucks nach einer eventuellen Typumwandlung nötig ist.

```
const int EOF = -1;  // Konstante EndOfFile
...
int i, j;
float x;
...
i = -j;             // auf eine Variable angewandt
x = -(3.5 * -i);    // auf einen Ausdruck angewandt
```

Das Ergebnis des unären +-Operators ist der Wert seines Operanden, nachdem die üblichen arithmetischen Umwandlungen durchgeführt worden sind.

Der logische Negationsoperator !, der wie ein Vorzeichen verwendet wird, liefert einen Wert vom Typ `bool`. Das Ergebnis ist `true`, wenn der Operand nach seiner Konvertierung nach `bool` (siehe Abschnitt 9.4.1) den Wert `false` hat, ansonsten `false`.

```
int  n;
bool isNeg, isZero;
char ch;

isZero = !n;      // Unschoen, besser:
isZero = (n==0);
...
isNoLetter = !(((ch>='a')&&(ch<='z'))||
               ((ch>='A')&&(ch<='Z')));
```

Der Operator ~ (Tilde), der vor einem Ausdruck steht, bildet das binäre Einser-Komplement seines Operanden. Dieser Operand muß ein integraler Typ beziehungsweise ein Enumerationstyp sein.

```
unsigned int wort, x;
...
wort = ~wort;  // Bildung des Einser-Komplements
f = ~0;        // Setzen aller Bits von f
```

Der Ausdruck

```
x & ~077;              // Konstante 077 ist oktal
```

zeigt eine typische Verwendung des Operators ~ mit dem Operator zum Maskieren einzelner Bits . Dieser Ausdruck ist im Gegensatz zum Ausdruck

```
x & 017777777777700  // speziell fuer 32 Bit Wortlaenge
```

von der Wortlänge des Rechners unabhängig.

Der Operator `sizeof` kann sowohl auf einen Ausdruck als auch auf einen Typ (`Typ-Id`) angewandt werden. Er liefert die Größe des Ausdrucks beziehungsweise des Typs, das heißt den Speicherbereich, den eine Variable des Typs `Typ-Id` verbraucht, in Bytes. Sein Ergebnis ist vom Typ `size_t`, ein Typ der implementierungsabhängig ist. Er ist in den Standard-Header-Dateien eines C++-Entwicklungssystems definiert.

Wird `sizeof` auf einen Vektortyp (beziehungsweise eine Variable vom Typ Vektor) angewandt, so liefert `sizeof` die Größe des gesamten Objekts.

```
int a[10];
int n;
cout << sizeof n;  // liefert die Groesse v. n in Bytes
n = sizeof(a);     // n = (2 oder 4)*10

struct Date {
   short century;
   short year;
   char month;
   char day;
} today;

n = sizeof(today);  // n = Groesse d. Struktur
```

Die Operatoren `new` und `delete` werden dazu verwendet, vom System dynamisch Speicher anzufordern und wieder freizugeben (vergleiche `malloc` und `free` in C). `new[]` und `delete[]` sind die entsprechenden Pendants zur Allokierung und Zerstörung von Vektoren. Die vier Operatoren werden in Abschnitt 8.2 im Zusammenhang mit Zeigern erläutert.

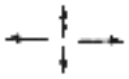
Vorige Seite: 5.4 Postfix-Ausdrücke **Eine Ebene höher:** 5 Ausdrücke **Nächste Seite:** 5.6 Andere

Ausdrücke

Vorige Seite: 5.5 Unäre Ausdrücke **Eine Ebene höher:** 5 Ausdrücke **Nächste Seite:** 6 Anweisungen

5.6 Andere Ausdrücke

Alle anderen Ausdrücke bestehen aus den noch nicht erwähnten Operatoren in Verbindung mit den zugehörigen Operanden:

Der *Cast*-Operator stellt die letzte der zahlreichen Möglichkeiten zur Typkonvertierung (vergleiche Seite ) dar. Auch diese Möglichkeit wird - wie die bereits angeführten - im Detail in Abschnitt 9.4.2 besprochen.

Die sogenannten Element-Zeiger-Operatoren `.*` beziehungsweise `->*` erlauben den Zugriff auf „Zeiger auf Klassen-Elemente`` über einen Zeiger auf ein Objekt. Sie werden im Zusammenhang mit Klassen und Zeigern auf Klassen-Elemente in Abschnitt 8.6 im Detail besprochen. Element-Zeiger-Operatoren sind linksassoziativ.

In C++ gibt es die drei multiplikativen Operatoren `*`, `/` und `%`, die alle drei linksassoziativ sind. Die Operanden für `*` und `/` müssen arithmetische Typen sein, der Operator `%` kann nur auf ganzzahlige Operanden angewandt werden.

Der Operator `*` bezeichnet eine Multiplikation. Dieser Operator ist kommutativ und assoziativ, so daß der Compiler Ausdrücke mit mehreren Multiplikationen auf der gleichen Vorrangebene in beliebiger Reihenfolge anordnen kann. Der Operator `/` bezeichnet eine Division. Der Modulo-Operator `%` liefert den Rest, der bei der Division seiner ganzzahligen Operanden entsteht. Ist einer der Operanden negativ, so ist bei vielen Compilern das Vorzeichen des Rests gleich mit dem des Dividenden.

Der Gebrauch von `/` oder `%` mit null als zweiten Operanden ergibt einen Fehler. Wenn der Divisor ungleich null ist, dann gilt immer: *E_1 = (E_1/E_2) * E_2 + E_1 % E_2*.

Das unten angeführte Beispiel zeigt eine typische Schaltjahr-Abfrage in C++. Ein Jahr ist dann ein Schaltjahr, wenn die Jahreszahl durch vier, aber nicht durch hundert teilbar ist, wobei noch die Ausnahme gilt, daß Jahre, die durch vierhundert teilbar sind, wiederum Schaltjahre sind. In C++ formuliert:

```
if ((year % 4 == 0) && (year % 100 != 0)
    || (year % 400 == 0)) {
                              // Schaltjahr-Behandlung
} else {
                              // Kein Schaltjahr
```

```
}
```

Die binären Operatoren + und - für Addition und Subtraktion sind linksassoziativ und können neben der numerischen Addition auch zur Addition und Subtraktion von Zeigerwerten verwendet werden.

Sind die beiden Operanden von arithmetischem Typ, so liefert der Operator + die Summe und der Operator - die Differenz seiner Operanden:

```
float a, b, c;
int x, y, z;
...
x = y + z;
a = b + c;
cout << a - (b + y);
```

Auch Zeigerwerte können addiert beziehungsweise subtrahiert werden (Zeigerarithmetik) - mehr dazu in Abschnitt 8.3.1.

Die Operatoren << und >> sind linksassoziativ und verschieben ihre linken Operanden (*expr1*) um die durch ihren rechten Operanden (*expr2*) angegebenen Bitpositionen nach links oder rechts.

Beide Operanden müssen ganzzahlig sein. Das Ergebnis der Operation hat den Typ des ersten Operanden und ist nicht definiert, wenn der rechte Operand negativ oder nicht kleiner als die Länge des linken Operanden in Bits ist.

```
unsigned int a, b, c;
...
a = 237;     // a binaer: 00000000 11101101
b = a >> 2;  // b binaer: 00000000 00111011 (237/4=59)
c = a << 1;  // c binaer: 00000001 11011010 (237*2=474)
```

Der Wert von `a << b` ist das Bitmuster von `a` verschoben um `b` Bits nach links. Dabei werden Bits mit Wert 0 nachgeschoben. Der Wert von `a >> b` ist das Bitmuster von `a` um `b` Bits nach rechts verschoben. Ist `a` vom Typ `unsigned`, so werden Bits mit Wert 0 nachgeschoben. Wenn `a` jedoch ein Integer-Wert mit Vorzeichen ist, kann keine arithmetische Verschiebung stattfinden, bei der das Vorzeichen-Bit nachgeschoben wird (compiler- beziehungsweise maschinenabhängig).

Die Operatoren < (kleiner), > (größer), <= (kleiner oder gleich) und >= (größer oder gleich) liefern den `bool`-Wert `false`, wenn die angegebene Relation falsch ist, und `true`, wenn die Relation vorliegt. Diese Operatoren sind linksassoziativ, was zu Problemen führen kann: `a < b < c` liefert nicht, was es zu bedeuten scheint. Es wird zuerst ermittelt, ob `a<b` gilt und anschließend, ob dieses Resultat (`false` oder `true`) < `c` ist. Richtig wäre: `(a < b) && (b < c)`.

Beispiele:

```
int a, b;
bool compRes;
float x, y;
...
compRes = a < b;
...
```

```
if (x >= y) cout << "x >= y";
```

Mit den Vergleichsoperatoren können auch Zeigerwerte verglichen werden. Das Resultat wird durch die relative Position der Objekte im Adreßraum bestimmt, auf welche die Zeigerwerte verweisen. Vergleiche sollten aus Gründen der Portabilität nur auf Zeiger, die auf Objekte im gleichen Vektor verweisen, angewandt werden (siehe Abschnitt 8.3.1).

Der Operator `==` prüft auf Gleichheit, der Operator `!=` auf Ungleichheit seiner beiden Operanden. Im Gegensatz zu anderen Programmiersprachen stellt das einfache Gleichheitssymbol `=` in C++ keinen Vergleich sondern die Wertzuweisung dar.

Die beiden Operatoren liefern den `bool`-Wert `false`, wenn die angegebene Relation falsch ist, und `true`, wenn die Relation vorliegt. Sie sind - wie die relationalen Operatoren - linksassoziativ, weisen aber eine niedrigere Priorität auf. Der Ausdruck

```
e1 < e2 == e3 < e4
```

ist gleichbedeutend mit `((e1 < e2) == (e3 < e4))` und liefert genau dann den Wert `true`, wenn beide Vergleiche `e1 < e2` und `e3 < e4` das gleiche Resultat liefern.

```
int zeilen;                     // Zeilenzaehler
char ch;
...
zeilen = 0;                     // Zeilenanzahl = 0
while ((c = getchar()) != EOF) // Solange nicht EOF
   if (c == '\n')               // Zaehler erhoehen wenn
      ++zeilen;                 // Zeilenvorschub
   ...
```

Die Gleichheitsoperatoren verhalten sich analog zu den relationalen Operatoren und können auch auf Zeiger angewandt werden. Sinnvoll ist auch ein Vergleich eines Zeigers mit dem Wert Null (0), der als Nullzeiger, also ein Zeiger, der auf kein Objekt verweist, aufgefaßt wird.

Die bitweisen Operatoren `&`, `^` und `|` sind linksassoziativ und ermöglichen die direkte Manipulation einzelner Bits. Die Operanden dieser Operatoren müssen ganzzahlige Typen sein und werden in jeder Bitposition nach den in Tabelle 4.2 angegebenen Regeln miteinander verknüpft.

Die Und-Verknüpfung zum Beispiel wird oft benutzt, um einzelne oder mehrere Bits eines Wortes zu ,,löschen``, also auf 0 zu setzen. Beispielsweise löscht

```
c = n & 0177;
```

in `n` alle Bits mit Ausnahme der letzten sieben. Die Oder-Verknüpfung wiederum kann zum Setzen einzelner Bits verwendet werden:

```
x = x | Maske;
```

setzt genau die Bits (zusätzlich zu den in x bereits gesetzten), die in `Maske` gesetzt sind.

Die beiden binären logischen Operatoren `&&` (bedingtes logisches Und) und `||` (bedingtes logisches Oder) können auf alle Ausdrücke angewandt werden, die auf `bool` konvertierbar sind. Das Resultat ist immer ein `bool`-Wert.

Ausdrücke , die mit diesen Operatoren verknüpft sind, werden immer strikt von links nach rechts ausgewertet und zwar nur so lange, bis das Resultat der logischen Verknüpfung feststeht (*Short-Circuit*, Kurzschlußauswertung): Ein logischer Und-Ausdruck erhält genau dann den Wert `true`, wenn die Werte beider Operanden `true` sind. Ist einer der Operandenwerte `false`, so hat auch der Und-Ausdruck diesen Wert, wobei die Auswertung des Und-Ausdrucks sofort abgebrochen wird, *sobald* die Auswertung eines Operanden `false` ergibt.

Die Kurzschlußregel für die Auswertung eines Oder-Ausdrucks erfolgt demnach so, daß der gesamte Ausdruck `true` ergibt, *sobald* die Auswertung eines der Operanden `true` ergibt.

Die Auswertung des Vergleichsausdrucks

```
if (((c >= 'a') && (c <= 'z')) ||
    ((c >= 'A') && (c >= 'Z'))) ...
```

zum Beispiel erfolgt von links nach rechts. Ist `c` ein Kleinbuchstabe, so hat der Gesamtausdruck nach der Auswertung `((c>='a') && (c<='z'))` bereits den Wert `true`, und die Auswertung wird abgebrochen. Die Priorität von `&&` ist größer als die Priorität von `||`, und beide haben geringere Priorität als die Vergleichsoperatoren. Folglich können in obigem Beispiel auch sämtliche Klammern innerhalb des Ausdrucks weggelassen werden, was sich aber auf die Lesbarkeit negativ auswirkt.

Folgendes Beispiel ist *nur* aufgrund der Kurzschlußauswertung richtig:

```
if ((x!=0) && (a/x == 17)) { ...
```

Bei einer Kurzschlußauswertung wird die `if`-Abfrage sofort abgebrochen. Ohne Kurzschlußauswertung wird auch der zweite Ausdruck (`a/x`) in jedem Fall ausgewertet, was zu einem Fehler führt, wenn `x` gleich 0 ist.

Der Bedingungsoperator `?:` ist linksassoziativ und stellt eine Art verkürzte Selektion dar. Ein Ausdruck der Art `expr1 ? expr2 : expr3` liefert den Wert von `expr2` falls `expr1` wahr ist, sonst den Wert von `expr3`. Zu beachten ist, daß das Ergebnis einer bedingten Bewertung wieder ein Ausdruck ist (entweder `expr1` oder `expr2`) und folglich so wie jeder andere Ausdruck verwendet werden kann.

Um beispielsweise der Variablen `max` den größeren der Werte `x` und `y` zuzuweisen, ist es möglich, anstelle von

```
if (x > y) {
   max = x;
} else {
   max = y;
}
```

einfach zu schreiben:

```
max =  (x > y) ? x : y;
```

Bedingte Bewertungen erweisen sich oft als schwer lesbar. Ein Beispiel für den ,,schlechten`` Einsatz des Bedingungsoperators ist unten angegeben: Bei der Ausgabe von *N* Elementen eines Vektors soll zwischen den Elementen ein Leerzeichen und nach zehn Elementen sowie nach dem letzten Zeichen ein Zeilentrenner ausgegeben werden:

```
for (i = 0; i < N; i++)
  cout << a[i] <<
       (i % 10 == 9 || i == N-1) ? '\n' : ' ');
```

- Bedingte Bewertungen führen oft zu knappen und unleserlichen Formulierungen und sollten daher nicht um jeden Preis verwendet werden. Oft erweist sich eine (etwas längere) `if`-Abfrage als besser lesbar.

Es gibt elf Zuweisungsoperatoren in C++: `=`, `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=` und `|=`. Der Operator `=` wird als einfacher Zuweisungsoperator, die anderen zehn werden als zusammengesetzte oder kombinierte Zuweisungsoperatoren bezeichnet.

Der linke Operand muß immer ein *LValue* sein, der Typ des Resultats entspricht immer dem Typ dieses linken Operanden. Eine Zuweisungsoperation liefert als Resultat immer den Wert, der sich nach der Zuweisung im linken Operanden befindet, daher sind auch mehrfache Zuweisungen wie

```
x = y = z = 0;  // x, y und z wird 0 zugewiesen
```

möglich.

Bei der einfachen Zuweisung mit `=` ersetzt der Wert des rechten Operanden den Wert des Objekts, das der linke Operand bezeichnet.

```
a = b;
a = pot(x, 2);
```

Das Resultat einer zusammengesetzten Zuweisung der Form `expr1` *op*`= expr2` kann aus der Zuweisung `expr1 = expr1` *op* `expr2` geschlossen werden. So entspricht etwa der Ausdruck `i *= 3` dem Ausdruck `i = i*3`. Der wesentliche Unterschied zwischen der verkürzten Form mittels kombinierten Zuweisungsoperator und der ,,normalen`` Form besteht darin, daß bei der Kurzform `expr1` nur einmal bewertet wird. Dadurch kann der Compiler unter Umständen einen effizienteren Zielcode erzeugen.

Bei den `+=` und `-=` Operationen kann der linke Operand ein Zeiger und der rechte ein `int`-Wert sein. Von diesen Fällen abgesehen, müssen alle Operanden arithmetischen Typen angehören.

```
i += j;         // erhoeht i um Wert j
wort >>= 1;     // verschiebt Bits v. wort um 1
                // nach rechts
x1 *= x2 - 1;  // anstelle von x1 = x1 * (x2 - 1)
```

Neben der Verwendung als Begrenzer in Aufzählungen (zum Beispiel in Listen von Funktionsargumenten oder bei Deklarationen) wird das Komma als Operator verwendet.

Der Operator `,` ist linksassoziativ. Die zwei durch Komma getrennten Ausdrücke werden von links nach rechts ausgewertet. Typ und Wert des rechten Ausdrucks bestimmen Typ und Wert des Gesamtausdrucks, das heißt der Wert des linken Ausdrucks wird zwar berechnet, aber nicht weiter verwendet. Zum Beispiel wird bei einem Ausdruck

```
x = 1, y += x;
```

zunächst `x` der Wert `1` zugewiesen und anschließend `y` um diesen Wert von `x` erhöht. Der Ausdruckswert ergibt sich aus dem letzten „Teil-Ausdruck`` und damit aus dem neuen Wert von `y`.

Der Operator `,` darf aber nicht mit dem Trennzeichen `,` in Funktionsargument- und Initialisierungslisten verwechselt werden. Zur Vermeidung von Mehrdeutigkeiten wie etwa im folgenden Beispiel müssen Klammern benutzt werden:

```
foo(i, (j=1, j+4), k);
```

Die Funktion `foo` wird mit drei und nicht mit vier Argumenten aufgerufen, und zwar mit `i`, `5` und `k`. Für ein weiteres Beispiel der Verwendung des Kommaoperators siehe Abschnitt 6.6.3.

Vorige Seite: 5.5 Unäre Ausdrücke **Eine Ebene höher:** 5 Ausdrücke **Nächste Seite:** 6 Anweisungen

Vorige Seite: 5.6 Andere Ausdrücke **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** 6.1 Ausdrucksanweisung

---

6 Anweisungen

Bisher wurden die Basisdatentypen von C++, einfache Deklarationen und Ausdrücke behandelt. Dieser Abschnitt beschreibt die Anweisungen, mit denen die Verarbeitung der Objekte gesteuert wird. Die verschiedenen Anweisungen umfassen Zuweisungen (die genaugenommen unter den Ausdrücken einzuordnen wären), Blockanweisungen, Deklarationen sowie Steueranweisungen. Als Steueranweisungen werden die Selektionsanweisungen `if`, `if-else`, `switch`, die Iterationsanweisungen `for`, `while`, `do-while` und die Sprunganweisungen `break` (zum Aussprung aus Schleifen und `switch`-Konstrukten), `continue` (zur Fortführung des Programmablaufs bei der nächsten Schleifeniteration), `return` (zum Rücksprung aus einer Funktion) und `goto` bezeichnet.

Im Zusammenhang mit Anweisungen sind die Konzepte der *Strukturierten Programmierung* von Bedeutung. Der Begriff Strukturierte Programmierung umschreibt (sehr vereinfacht dargestellt) die These, daß drei Anweisungstypen ausreichen, um jedes algorithmische Problem lösen zu können:

- Sequenz
  Eine Sequenz ist die Aufeinanderfolge von verschiedenen Anweisungen.
- Iteration
  Iterationen sind Wiederholungsanweisungen, die es erlauben, eine oder mehrere Anweisungen 0, 1 oder *n* mal auszuführen.
- Selektion
  Eine Selektion erlaubt es, eine oder mehrere Anweisungen in Abhängigkeit von einer Bedingung auszuführen.

Die Verwendung von Sprunganweisungen (`goto`s) ist unnötig und gilt sogar als schlechter Programmierstil, seit [Dij68] nachgewiesen hat, daß ein direkter Zusammenhang zwischen der Anzahl der Sprunganweisungen und der mangelhaften Qualität von Programmen besteht.

Vorige Seite: 5.6 Andere Ausdrücke **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** 6.1 Ausdrucksanweisung

Vorige Seite: 6 Anweisungen **Eine Ebene höher:** 6 Anweisungen **Nächste Seite:** 6.2 Sprungmarken

6.1 Ausdrucksanweisung

Eine Ausdrucksanweisung ist ein optionaler Ausdruck, der durch einen Strichpunkt abgeschlossen wird:

expression-statement:

*expression*opt ;

Jene Ausdrucksanweisung, die durch einen alleinstehenden Strichpunkt angegeben ist, wird als Nullanweisung bezeichnet. Sie kann in solchen Fällen von Nutzen sein, in denen die Syntax eine Anweisung verlangt, jedoch vom Programm her keine Notwendigkeit dafür besteht. Eine Nullanweisung muß verwendet werden, um zum Beispiel eine Schleifenanweisung wie `for` oder `while` mit einem leeren Schleifenrumpf zu versehen.

Beispiel:

```
while (*p++ = getch());
```

Eine andere spezielle Ausdrucksanweisung ist die Zuweisung . Sie hat die Form *expr1* `=` *expr2*`;`. Der linke Operand (*expr1*) des Zuweisungsausdrucks ist ein *LValue*. Da ein *LValue* sich nicht auf eine Konstante beziehen kann, ist sichergestellt, daß konstante Objekte nicht verändert werden (da sie nie auf der linken Seite einer Zuweisung angeführt sein können).

Wie bereits erwähnt, kann der Zuweisungsoperator ein einfacher (`=`) oder ein kombinierter Zuweisungsoperator (`*=`, `+=`, `/=` etc.) sein. Bei der einfachsten Zuweisung mit `=` ersetzt der Wert des Ausdrucks den Wert des Objekts, auf das sich der linke Operand bezieht.

Beispiele:

```
int d, ch;
d = 3.14;  // d hat nach der Zuweisung d. Wert 3
ch = 'a';  // ch erhaelt den Wert 97 (ASCII: 'a')
d *= 2;    // d wird nur einmal ausgewertet (= 3*2).
```

Der Typ eines Zuweisungsausdrucks ist der Typ des *LValues*. Sein Wert ist der Wert des *LValues* nach erfolgter Zuweisung. Bei geschachtelten Zuweisungsausdrücken wird jeweils der äußerst rechte Ausdruck zuerst ausgewertet:

```
a = b = 0;      // entspricht a = (b=0)
```

Eine dritte Art von Ausdrucksanweisungen sind die sogenannten Funktionsaufrufe. Funktionsaufrufe erfolgen über die Angabe eines Funktionsnamens gefolgt von runden Klammern, in die die sogenannten Parameter eingeschlossen sind. Funktionsaufrufe werden im nächsten Kapitel behandelt.

Vorige Seite: 6 Anweisungen **Eine Ebene höher:** 6 Anweisungen **Nächste Seite:** 6.2 Sprungmarken

Vorige Seite: 6.1 Ausdrucksanweisung **Eine Ebene höher:** 6 Anweisungen **Nächste Seite:** 6.3 Blockanweisungen

6.2 Sprungmarken

Eine Sprungmarke ,,markiert`` eine Anweisung. Eine markierte Anweisung kann von Sprunganweisungen als Ziel angegeben werden. Die Marke (*Label*) dient also als eine Art Verweis, zu dem hinverzweigt werden kann. Es existieren drei verschiedene Arten von Sprungmarken:

```
labeled-statement:
                identifier : statement
                case constant-expression : statement
                default : statement
```

Die erste Art (*labeled-statement*) ist eine frei zu vereinbarende Sprungmarke, zu der mittels `goto` verzweigt wird (siehe Abschnitt 6.7.4). Die beiden anderen Arten von Sprungmarken werden in der `switch`-Anweisung (Abschnitt 6.5.2) verwendet.

Beispiel:

```
  ...
label1: j=7;
        i++;
label2:
```

Vorige Seite: 6.2 Sprungmarken **Eine Ebene höher:** 6 Anweisungen **Nächste Seite:** 6.4 Deklaration

6.3 Blockanweisungen

Die Blockanweisung erlaubt es, mehrere Anweisungen zu einer einzigen zusammenzufassen. Sie wird oft auch als zusammengesetzte Anweisung bezeichnet. Ein Block ist also eine (möglicherweise leere) Liste von Anweisungen, die in geschweifte Klammern eingeschlossen ist.

compound-statement:

{ *statement-seq*$_{opt}$ }

statement-seq:

statement-seq statement

Ein Beispiel für eine Blockanweisung:

```
{
  cout << "Test";
}
```

Blockanweisungen können auch geschachtelt werden. Ein Beispiel dazu:

```
{
   ...
   {
      q = a / b;
      r = a % b;
   }
   ...
}
```

Jeder Block vereinbart einen neuen *Gültigkeitsbereich* (*Scope*). Der Gültigkeitsbereich ist der Bereich, in dem ein deklariertes Objekt verwendet werden kann, also gültig ist. Der Gültigkeitsbereich eines in einem Block deklarierten Bezeichners erstreckt sich von der Position seiner Deklaration bis zum Ende des umgebenden Blocks. In jedem Gültigkeitsbereich darf ein Bezeichner nur einmal deklariert werden. Da ein Block einen neuen Gültigkeitsbereich vereinbart, können in einem Block aber Bezeichner neu deklariert werden, die Namen von bereits existierenden Objekten tragen.

Das folgende Programmstück vereinbart insgesamt vier Variablen `x` und `y`:

```
{
   int x=5, y=10;
   {
      int x=15, y=20;
      cout << "x=" << x << "y=" << y << "\n";
   }
   cout << "x=" << x << "y=" << y << "\n";
}
```

Sowohl im inneren als auch im äußeren Block werden jeweils zwei Objekte `x` und `y` deklariert. Das führt zu keinem Fehler, da die beiden Objekte jeweils in einem eigenen Gültigkeitsbereich vereinbart sind. Zu beachten ist, daß die beiden ,,äußeren`` Variablen `x` und `y` auch im inneren Block *existieren* - sie werden von den inneren Objekten lediglich *überdeckt* und können daher nicht verwendet werden.

Das Programm liefert folgende Ausgabe:

```
x=15   y=20
x=5    y=10
```

Ein anderer Begriff in diesem Zusammenhang ist die *Lebensdauer* von Objekten. Beide Begriffe (Gültigkeitsbereich und Lebensdauer) werden in Kapitel 9 ausführlich behandelt.

Vorige Seite: 6.2 Sprungmarken **Eine Ebene höher:** 6 Anweisungen **Nächste Seite:** 6.4 Deklaration

Vorige Seite: 6.3 Blockanweisungen **Eine Ebene höher:** 6 Anweisungen **Nächste Seite:** 6.5 Selektion

6.4 Deklaration

Die Deklarationsanweisung führt einen neuen Bezeichner im aktuellen Block ein. Sie wurde bereits in den Abschnitten 4.1 und 4.2 im Zusammenhang mit Deklarationen und Basisdatentypen besprochen. Wesentlich ist, daß jeder Bezeichner im aktuellen Block eindeutig sein muß.

Beispiele für Deklarationen von Objekten:

```
{
  int    i;
  float  f;

  {
    int i;   // ueberdeckt aeusseres i
    ...
  }
}
char   ch;  // ch ist ein Zeichen
```

Problematisch im Zusammenhang mit Deklarationen kann die `goto`-Anweisung sein - mehr dazu in Abschnitt 6.7.4.

Vorige Seite: 6.4 Deklaration **Eine Ebene höher:** 6 Anweisungen **Nächste Seite:** Die `if else`-Anweisung

6.5 Selektion

Selektionen erlauben es, den Ablauf eines Programms von bestimmten Zuständen abhängig zu machen. C++ kennt zwei verschiedene Selektions-Anweisungen, die `if`- und die `switch`-Anweisung.

Vorige Seite: 6.5 Selektion **Eine Ebene höher:** 6.5 Selektion **Nächste Seite:** 6.5.2 Die `switch`-Anweisung

---

Die `if else`-Anweisung

`selection-statement:`

`if ( `*`condition`*` ) `*`statement`*

`if ( `*`condition`*` ) `*`statement`*` else `*`statement`*

`condition:`

`expression`

`type-specifier-seq declarator = assignment-expression`

Die `if`-Anweisung ist eine „einfache`` Selektion. Wird eine `if`-Anweisung betreten, so wird zunächst die Bedingung *condition* ausgewertet. Im Fall einer `if`-Anweisung ohne `else`-Zweig wird die der Bedingung folgende Anweisung *statement* genau dann ausgeführt, wenn *condition* `true` ergibt.

Jeder `if`-Anweisung kann optional ein sogenannter `else`-Zweig angefügt werden. Dieser Teil wird ausgeführt, wenn *condition* `false` ergibt.

Beispiele:

```
if (a!=0)    // falls a den Wert 0 hat, wird die
  c = b/a;   // Division durch a nicht ausgefuehrt
...
if (a >= b)  // Maximum zweier Zahlen ermitteln
  max = a;
else
  max = b;
```

`if`-Anweisungen können auch geschachtelt werden. Sind keine geschweiften Klammern gesetzt, so bezieht sich das `else` auf den jeweils vorhergehenden `if`-Befehl ohne `else`-Zweig. Das folgende Programmstück ermittelt von drei Werten `a`, `b`, `c` das Maximum:

```
if (a >= b)
  if (a >= c) max = a;          // a >= b
  else max = c;
```

```
  else if (b >= c) max = b;    // b > a
  else max = c;
```

Das Beispiel zeigt, daß keine beziehungsweise eine inkonsistente Einrückung zu schlecht lesbarem und wartbarem Code führt. Mit entsprechender Einrückung könnte das Codestück etwa wie folgt aussehen:

```
if (a >= b)
  if (a >= c)
    max = a;
  else
    max = c;
else
  if (b >= c)
    max = b;
  else
    max = c;
```

Blockanweisungen können verwendet werden, um `if`- und `else`-Zweig abzugrenzen. Dies ist dann *erforderlich*, wenn in einem Zweig mehr als eine Anweisung notiert werden soll, da die `if`-Anweisung sowohl für den `if`- als auch für den `else`-Zweig jeweils nur *eine* einzelne Anweisung vorsieht!

```
if ((punkte < 15) || (abgegebUebungen < 8)) {
  cout << "Zu wenig Uebungen!";
  note = 5;
} else if (punkte >= 27) {
  cout << "Sehr Gut";
  note = 1;
} else if (punkte >= 23) {
  cout << "Gut";
  note = 2;
...
}
```

Die geschweiften Klammern bedeuten keinen Performance- oder Effizienznachteil und sollten im Zweifelsfall immer verwendet werden. Gerade bei mehreren geschachtelten `if`-Anweisungen mit `else`-Zweigen wird dadurch die Lesbarkeit des Programms deutlich gesteigert. Die Ellemtel-Programmierrichtlinien für C++ zum Beispiel fordern das Verwenden der geschweiften Klammern in jedem Fall - auch bei einzelnen Anweisungen im `if`- beziehungsweise `else`-Zweig [HN93].

`if`-Anweisungen führen so wie die Kontrollanweisungen `for`, `while` und `switch` einen eigenen Gültigkeitsbereich (*Substatement*) ein. Der Gültigkeitsbereich erstreckt sich vom Beginn der `if`-Anweisung bis zu deren Ende. Eine Variable, die innerhalb einer `if`-Anweisung deklariert wird, gilt bis an das Ende der `if`-Anweisung.

```
if ((int i = help/anzahl) >= 27) {
  cout << "Durchschnitt >= 27: " << i;
}
i = 0;   // Fehler! i gilt nur bis ans Ende v. if
```

```
if ((int i = help/anzahl) >= 27)
  cout << "Durchschnitt >= 27: " << i;

i = 0;   // Wie oben, Fehler!
```

Diese Regel führt teilweise auch zu verwirrenden Situationen. So ist folgendes Codestück fehlerhaft:

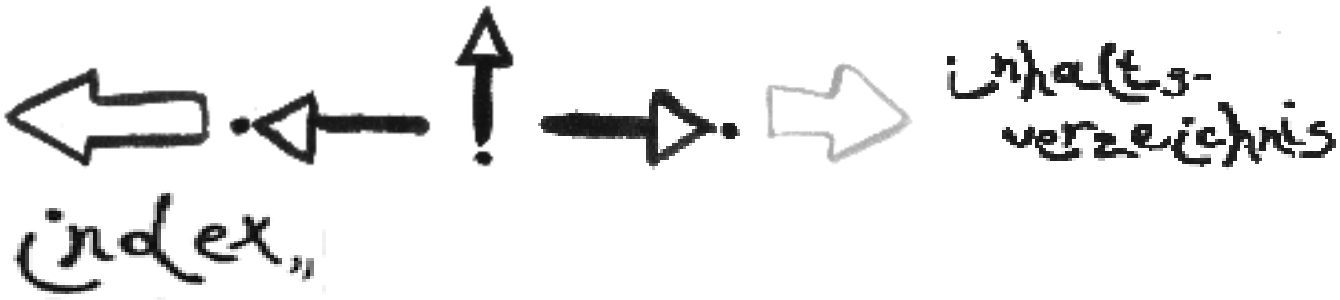
```
if (help > 99)
  int i;
i = 17;
```

`int i` ist eine *lokale* Deklaration, der Bezeichner gilt daher nur innerhalb der `if`-Anweisung. Als Art Eselsbrücke kann angeführt werden, daß obige `if`-Anweisung der folgenden Anweisungsfolge entspricht:

```
if (help > 99) {
  int i;
}
```

`i` ist innerhalb des Gültigkeitsbereichs der `if`-Anweisung deklariert und damit auch nur dort gültig.

Vorige Seite: 6.5 Selektion **Eine Ebene höher:** 6.5 Selektion **Nächste Seite:** 6.5.2 Die `switch`-Anweisung



Vorige Seite: Die `if else`-Anweisung **Eine Ebene höher:** 6.5 Selektion

6.5.2 Die `switch`-Anweisung

Eine `if`-Anweisung ist eine einfache Fallunterscheidung, die in Abhängigkeit von der angegebenen Bedingung den `if`- oder den `else`-Zweig (falls vorhanden) ausführt. Sehr oft aber gibt es Fälle, in denen in Abhängigkeit von einer Art „Schaltervariable`` (in der Regel eine Zahl oder ein Zeichen) verschiedene Codestücke ausgeführt werden sollen.

Eine derartige Abfrage ist die `switch`-Anweisung in C++. Sie weist folgende Syntax auf:
`switch-statement:`

`switch ( condition ) statement`

condition ist ein integraler Ausdruck, der als Schalter für die eigentliche Selektion dient. Die Anweisung *statement* ist im allgemeinen eine Blockanweisung, die aus einer Reihe von Anweisungen besteht, von denen jede mit einer `case`-Marke versehen sein kann. Die `case`-Marke hat (wie bereits angeführt) die Form
`case constant-expression :`

und stellt einen „Einsprungpunkt`` dar. Maximal einer der vorhandenen Einsprungpunkte darf die Sprungmarke `default` aufweisen:
`default :`

Der Ablauf beim Betreten einer `switch`-Anweisung ist wie folgt: Die Bedingung wird ausgewertet und in Abhängigkeit vom Ergebnis wird zu einem der vorhandenen Einsprungpunkte verzweigt. Von diesem Punkt aus werden *alle weiteren* Anweisungen der `switch`-Anweisung bis zum Ende abgearbeitet. Damit unterscheidet sich die `switch`-Anweisung von C++ von den `CASE`-Anweisungen in anderen Programmiersprachen wie Ada, Modula2 oder Oberon.

Ist ein entsprechender Einsprungpunkt nicht vorhanden, so verzweigt die Programmausführung zur Einsprungmarke `default`. Ist dieser nicht vorhanden, wird keine der angeführten Anweisungen ausgeführt.

```
switch (ch) {
```

```
   case '#':
     stat1;
   case '?':
     stat2;
   case '%':
     stat3;
   default :
     stat4;
}
```

Im oben angeführten Beispiel wird die Variable `ch` ausgewertet und - je nach Wert der Variablen - zu einem der Einsprungpunkte verzweigt. Ein Wert von '?' zum Beispiel führt zum Sprung zur Anweisungsfolge `stat2`. In weiterer Folge werden `stat3` und `stat4` ausgeführt.

Sollen die weiteren Anweisungen *nicht* ausgeführt werden, sondern nur die bei der jeweiligen Marke notierten Anweisungen, so muß vor jeder neuen `case`-Marke ein Aussprung aus der `switch`-Abfrage erfolgen (siehe dazu auch Abschnitt 6.7):

```
switch (ch) {
   case '#':
     stat1;
     break;  // Aussprung aus switch
   ...
   case '%':
     stat3;
     break;  // Aussprung aus switch
   default :
     stat4;
}
```

Es ist guter Programmierstil, in `switch`-Anweisungen *immer* einen `default`-Zweig zu codieren. Falls dieser Zweig nicht betreten werden soll oder darf, erfolgt dort eine entsprechende Fehlermeldung. Diese „defensive`` Vorgangsweise ist eine Absicherung gegen ungewollte Fehlerfälle.

Vorige Seite: Die `if else`-Anweisung **Eine Ebene höher:** 6.5 Selektion

Vorige Seite: 6.5.2 Die `switch`-Anweisung **Eine Ebene höher:** 6 Anweisungen **Nächste Seite:** 6.6.1 Die `while`-Anweisung

6.6 Iteration

Iterationen werden auch als *Schleifen* bezeichnet. Eine Iteration ist eine Anweisung, die es erlaubt, eine andere Anweisung beliebig oft auszuführen. Damit unterscheiden sie sich grundsätzlich von den bisher angeführten Anweisungstypen, mit denen es lediglich möglich war, Anweisungen hintereinander auszuführen oder Anweisungen ein- oder keinmal auszuführen.

Wir unterscheiden in C++ - so wie in den meisten modernen Programmiersprachen - zwischen drei verschiedenen Schleifen: der Abweisungs- oder `while`-Schleife, der Durchlauf- oder `do until`-Schleife und der Zähl- oder `for`-Schleife:

iteration-statement:

```
while ( condition ) statement

do statement while ( expression ) ;

for ( for-init-statement conditionopt ;
  expressionopt ) statement
```

for-init-statement:

```
expression-statement

simple-declaration
```

Alle drei Schleifenkonstruktionen weisen den bei der `if`-Anweisung beschriebenen Gültigkeitsbereich *Substatement* auf.

Vorige Seite: 6.6 Iteration **Eine Ebene höher:** 6.6 Iteration **Nächste Seite:** Die `do while`-Anweisung

6.6.1 Die `while`-Anweisung

Die `while`-Schleife führt die Anweisung *statement* so lange aus, wie die Auswertung der Bedingung *condition* `true` ergibt. Im Detail kann der Ablauf wie folgt angegeben werden:

Der Ausdruck *condition* wird *vor* dem Betreten des Schleifenrumpfs (gegeben durch *statement*) ausgewertet. Ergibt der Ausdruck `false`, so wird die Schleife nicht betreten, andernfalls wird der Schleifenrumpf ausgeführt. Danach wird abermals die Bedingung *condition* ausgewertet und der Ablauf wiederholt sich.

Ein typisches Beispiel für eine `while`-Anweisung ist das Lesen aus einer Eingabedatei:

```
ch = getchar();         // Lesen eines Zeichens
while (ch != EOF) {     // Solange nicht EOF:
  ...                   //  Verarbeitung
  ch = getchar();       //  Naechstes Zeichen
}
```

Kürzer (und unleserlicher):

```
while ((ch = getchar()) != EOF);
```

Die `while`-Schleife kann die beiden anderen Schleifenarten (`for`, `do while`) ersetzen und sollte in allen allgemeinen Fällen verwendet werden.

Vorige Seite: 6.6.1 Die `while`-Anweisung **Eine Ebene höher:** 6.6 Iteration **Nächste Seite:** 6.6.3 Die `for`-Anweisung

---

Die do while-Anweisung

Die `do while`-Anweisung führt so wie die `while`-Anweisung eine Anweisung so lange aus, wie die Bedingung *condition* gilt. Allerdings erfolgt die Prüfung der Bedingung erst *nach* der Ausführung des Schleifenrumpfs. Im Detail ist der Ablauf bei der Ausführung einer `do while`-Schleife wie folgt:

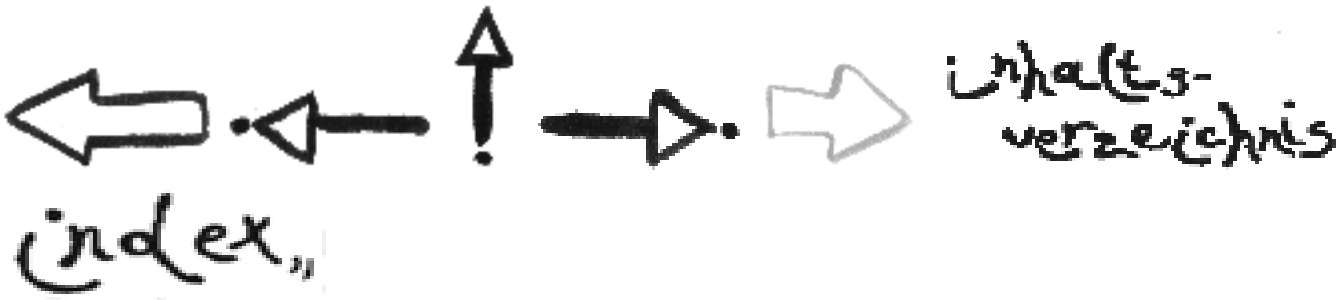
Der Schleifenrumpf wird ausgeführt. Anschließend wird die Bedingung *condition* ausgewertet. Ergibt die Auswertung `true`, so erfolgt ein weiterer Schleifendurchlauf, andernfalls wird die Schleife verlassen.

Damit ergibt sich ein Unterschied zu den bekannten `REPEAT - UNTIL`-Anweisungen, die eine Anweisungsfolge wiederholen *bis* eine Bedingung wahr wird.

Eine `do while`-Schleife wird immer mindestens einmal ausgeführt. Sie wird daher *nur* dann verwendet, wenn der Schleifenrumpf mindestens einmal durchlaufen werden soll.

Im Beispiel unten wird der größte gemeinsame Teiler zweier Zahlen nach dem Algorithmus von Euklid berechnet:

```
  // ggt nach Euklid
do {
  rest = a % b;
  a = b;
  b = rest;
} while ( rest > 0 );
```



Vorige Seite: Die `do` `while`-Anweisung **Eine Ebene höher:** 6.6 Iteration

6.6.3 Die `for`-Anweisung

Die `for`-Schleife ist eine Art „Abkürzung`` für eine spezielle `while`-Schleife. So entspricht eine `for`-Schleife der Art

`for ( `*`for-init-statement condition`*`; `*`expression`*`) `*`statement`*

in etwa der folgenden `while`-Schleife [ISO95, stmt.for]:

```
{

for-init-statement

while ( condition ) {

statement

expression ;

}

}
```

Anmerkung: In der Grammatik ist zwischen *for-init-statement* und *condition* kein Strichpunkt angeführt, da *for-init-statement* als Anweisung ohnehin durch einen Strichpunkt abgeschlossen ist.

Eine `for`-Schleife ist ein Schleifenkonstrukt, das zuerst die in *for-init-statement* angeführte Anweisung ausführt. Dann wird die Bedingung *condition* geprüft. Ergibt die Auswertung `true`, so wird der Schleifenrumpf und dann *expression* ausgeführt. Dann wird die Bedingung *condition* erneut geprüft.

Mit dieser Schleife lassen sich auf einfache Art und Weise Zählschleifen der folgenden Art realisieren:

```
sum = 0;  // Summe der Zahlen von 0 bis 99
for (int i=0; i<100; ++i) {
  sum += i;
```

```
}
```

Die Variable `i` im obigen Beispiel wird *Schleifenvariable* genannt. Es ist guter Programmierstil, eine Schleifenvariable lokal zu deklarieren (vergleiche dazu Schleifenvariablen in Ada): Die Variable wird nur innerhalb des Rumpfs verwendet, sie gilt genau dort, wo sie benötigt wird und belegt keinen überflüssigen Speicherplatz.

Schleifenvariablen werden lokal (das heißt im *for-init-statement*) deklariert und gelten damit ausschließlich innerhalb der `for`-Schleife.

Einige Anmerkungen zu Besonderheiten der `for`-Schleife:

- Ist *condition* nicht angegeben, so entspricht dies einer Endlosschleife (vergleichbar etwa mit `while (true)`).
- Die oben angeführte `while`-Schleife entspricht nicht ganz der angegebenen `for`-Schleife: Wird ein Schleifendurchlauf mittels `continue` abgebrochen (siehe Abschnitt 6.7.2), so wird *expression* ausgeführt und erst dann die Bedingung *condition* neu ausgewertet.
- Eine Endlosschleife kann einfach durch die Anweisung `for(;;)` angegeben werden.

Im Zusammenhang mit der `for`-Schleife wird häufig auch der Kommaoperator eingesetzt. Mit dem Kommaoperator können jeweils mehrere Ausdrücke im Schleifenkopf angegeben werden, um so beispielsweise zwei Indizes gleichzeitig zu manipulieren:

```
  // Umdrehen einer Zeichenkette s ("abcd"->"dcba")
for(i=0, j=strlen(s)-1; i<j; i++, j--) {
   c = s[i];
   s[i] = s[j];
   s[j] = c;
}
```

Das oben angeführte Codebeispiel kann ohne Zweifel als gefährlich bezeichnet werden. Wie so oft in C++ gilt auch in bezug auf den Einsatz des Kommaoperators in `for`-Konstrukten, daß jeder C++-Programmierer derartigen Code zwar verstehen, den eigenen Code aber auf jeden Fall lesbar und leicht verständlich gestalten sollte.

Vorige Seite: Die `do while`-Anweisung **Eine Ebene höher:** 6.6 Iteration

Vorige Seite: 6.6.3 Die `for`-Anweisung **Eine Ebene höher:** 6 Anweisungen **Nächste Seite:** 6.7.1 Die `break`-Anweisung

6.7 Sprunganweisungen

Eine Sprunganweisung ist eine Anweisung, die im Programmcode von einer gegebenen Stelle an das angegebene Ziel (eine Anweisung im Programmtext) ,,springt`` ohne die dazwischenliegenden Anweisungen auszuführen. C++ kennt folgende Sprunganweisungen:

jump-statement:

```
break ;
continue ;
return expressionopt ;
goto identifier ;
```

Die grundsätzliche Problematik von Sprunganweisungen besteht darin, daß der Programmablauf nicht mehr der statischen Notation des Programmtextes folgt. Problematisch sind dabei weniger etwaige Verzweigungen (Aussprünge) als vielmehr die kaum zu überblickenden Einsprungmöglichkeiten.

Dennoch existieren auch sinnvolle Einsatzmöglichkeiten für Sprunganweisungen. Aber noch mehr als bei den verschiedenen anderen Anweisungsarten gilt hier, daß der Einsatz nur in begründeten Fällen erfolgen sollte. Insbesonders die `goto`-Anweisung sollte vermieden werden!

Vorige Seite: 6.7 Sprunganweisungen **Eine Ebene höher:** 6.7 Sprunganweisungen **Nächste Seite:** 6.7.2 Die `continue`-Anweisung

6.7.1 Die `break`-Anweisung

Die `break`-Anweisung führt zu einem Sprung aus der direkt umschließenden Schleife oder `switch`-Anweisung. Der Steuerfluß wird bei jener Anweisung fortgesetzt, die der abgebrochenen `switch`- oder Iterations-Anweisung folgt. Da Sprünge aus einer `switch`-Anweisung schon behandelt wurden, folgt nun ein Beispiel für einen Sprung aus einer Schleife:

```
for (i = 0; i < n; i++) {  // Suche x in einem
                           // eindimensionalen Vektor
   if (a[i] == x) break;   // Abbruch wenn a[i] == x
}
                     // Fortsetzung nach Sprung hier
```

Vorige Seite: 6.7.1 Die `break`-Anweisung **Eine Ebene höher:** 6.7 Sprunganweisungen **Nächste Seite:** 6.7.3 Die `return`-Anweisung

---

6.7.2 Die `continue`-Anweisung

Die `continue`-Anweisung bewirkt einen Abbruch des aktuellen Schleifendurchlaufs in einer `while`-, `do`- oder `for`-Anweisung. Die Anweisungen im Schleifenrumpf, welche dem `continue`-Befehl folgen, werden ignoriert. Der Steuerfluß wird an die direkt umschließende `while`-, `do`- oder `for`-Anweisung weitergegeben.

Im folgenden sind die (gedachten) ,,Sprungziele`` für `continue`-Anweisungen bei den verschiedenen Schleifen angeführt [ISO95, stmt.cont]:

```
while (...) {     do {              for (...) {
 {                                   {
  ...                ...              ...
 }                                   }
 contin: ;          contin: ;        contin: ;
}                  } while (...)     }
```

In bezug auf die `for`-Schleife soll noch einmal angeführt werden, daß die `continue`-Anweisung zur sofortigen Ausführung von `expression` führt, erst dann wird die Schleifenbedingung *condition* ausgewertet:

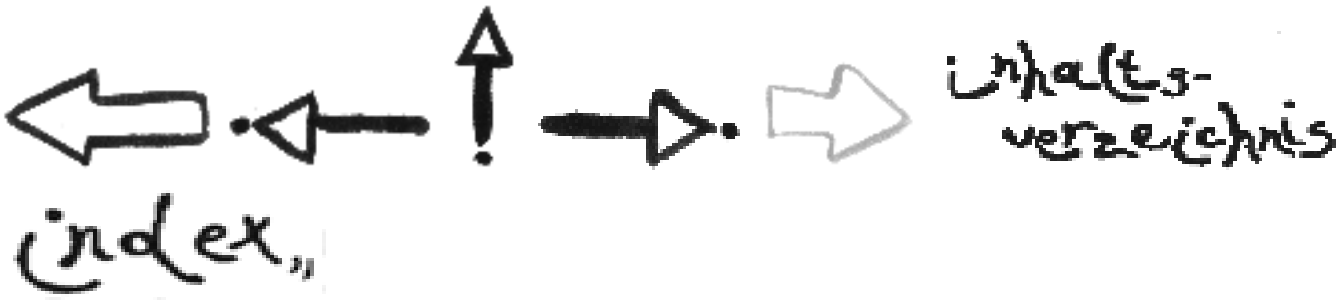
```
    // drucke gerade Zahlen von 1 bis 100
for (i = 1; i <= 100; i++) {
  if (i % 2 != 0) continue;
  cout << i;
}
```

- Die `continue`-Anweisung sollte nur selten verwendet werden, da sie oft zu unübersichtlichen und nur schwer verständlichen Schleifen führt.

Vorige Seite: 6.7.2 Die `continue`-Anweisung **Eine Ebene höher:** 6.7 Sprunganweisungen **Nächste Seite:** 6.7.4 Die `goto`-Anweisung

6.7.3 Die `return`-Anweisung

Die `return`-Anweisung führt zu einem Rücksprung aus einer Funktion (siehe Kapitel 7) an die Stelle, von der die Funktion aufgerufen wurde. Die erste Form wird in Funktionen ohne Rückgabewert verwendet (= Funktionen mit Rückgabewert vom Typ `void`). Die zweite Form liefert den Wert von `expression` als Funktionswert an die aufrufende Stelle zurück.



Vorige Seite: 6.7.3 Die `return`-Anweisung **Eine Ebene höher:** 6.7 Sprunganweisungen

6.7.4 Die `goto`-Anweisung

Eine `goto`-Anweisung bewirkt einen sofortigen Sprung zu der in *identifier* angegebenen Sprungmarke. Dabei ist zu beachten, daß bei einem Sprung zu einem Sprungziel keine Deklaration mit Initialisierung „übergangen`` und direkt in den Gültigkeitsbereich dieser Deklaration gesprungen wird.

Beispiel:

```
  ...
  goto label1;
  int  i = 3;  // Fehler: Deklaration
               // wird uebersprungen!
  ...
label1:
  i++;
label2:
```

Die Verwendung von `goto`s führt zur sogenannten *Spaghettiprogrammierung* . Dem Leser, dem dieser Begriff unklar ist, sei angeraten, die verschiedenen Wege durch ein Programm mit entsprechend vielen `goto`s einzuzeichnen.

`goto`s sollten unbedingt vermieden werden, da ihre Verwendung zu nur schwer nachvollziehbaren und oft fehlerhaften Programmstrukturen führt.

Vorige Seite: 6.7.4 Die `goto`-Anweisung **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** 7.1 Einführung

---

7 Funktionen

Mit den bisher beschriebenen Konzepten können Objekte deklariert und einfache Aktionen mit ihnen ausgeführt werden. In diesem Abschnitt wird der einfachste Abstraktionsmechanismus von C++ behandelt, die Funktion. Funktionen erlauben es, Anweisungen zu größeren Einheiten zusammenzufassen und sie dann wie eine einzelne Anweisung zu behandeln.

Das Kapitel ist wie folgt gegliedert:

- Zunächst wird der Funktionsbegriff in C++ im allgemeinen erläutert.
- Der nächste Abschnitt beschreibt die Deklaration und Definition von Funktionen im Detail.
- Danach werden die Verwendung von Funktionen (der sogenannte ,,Funktionsaufruf``) sowie die Parameterübergabe besprochen.
- Im weiteren werden spezielle Funktionen (Prozeduren und Operator-Funktionen) beschrieben.
- Im Anschluß daran werden drei ,,C++-Spezifika``, `inline`-Funktionen, vorbelegte Funktionsparameter und überladene Funktionen, erläutert.
- Das Konzept der Rekursion beschließt das Kapitel.

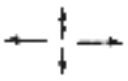
- 7.8 Rekursion

Vorige Seite: 7 Funktionen **Eine Ebene höher:** 7 Funktionen **Nächste Seite:** 7.2 Deklaration und Definition

7.1 Einführung

Funktionen sind eine Art ,,benutzerdefinierte Operationen``, die es erlauben, eine Reihe von Anweisungen zu einer einzelnen zusammenzufassen. Funktionen sind wie folgt aufgebaut:

- Sie haben einen Namen.
- Hinter dem Namen wird die sogenannte *Parameterliste* notiert. In der Parameterliste werden die Funktionsargumente (Parameter) angegeben. Parameter stellen sozusagen die ,,Operanden`` der Funktion dar.
- Vor dem Funktionsnamen wird der *Ergebnistyp* der Funktion notiert. Jede Funktion liefert (so wie ein Operator oder eine mathematische Funktion) einen bestimmten Wert als Ergebnis (= Rückgabewert).
- Die eigentlichen *Aktionen* einer Funktion werden im sogenannten *Funktionsrumpf* angegeben. Der Funktionsrumpf wird hinter der Parameterliste in geschwungenen Klammern angegeben. Die dort notierten Anweisungen führen die ,,komplexe`` Aufgabe der Funktion aus.
- Die Anweisungen im Funktionsrumpf benötigen im allgemeinen (Hilfs-)Variablen zur Ergebnisberechnung. Diese Objekte werden im Funktionsrumpf notiert und gelten nur dort. Sie werden als *lokale Variablen* bezeichnet, weil sie nur innerhalb der Funktion gelten und nur dort verwendet werden können.

Im folgenden ist als Beispiel für eine Funktion die bereits bekannte Ermittlung des größten gemeinsamen Teilers zweier Zahlen (vergleiche Seite ) angeführt:

```
  // Funktionsschnittstelle mit Rueckgabewert,
  // Name & Schnittstelle
int ggt(int a, int b)
{                   // Funktionsrumpf
  int rest;         // lokale Variable, gilt nur in ggt

  do {
    rest = a % b;
    a = b;
    b = rest;
  } while (rest>0);
```

```
  return a;         // Rueckgabe Ergebniswert
}
```

Die Funktion hat den Namen `ggt`, zwei Parameter `a` und `b` und liefert einen `int`-Wert als Ergebnis. Im Funktionsrumpf wird die lokale Variable `rest` deklariert (mehr dazu später). Der dahinter angeführte Anweisungsteil ist bekannt und bedarf mit einer Ausnahme keiner Erläuterung: Die `return`-Anweisung bricht die Funktionsausführung ab und gibt den dahinter angeführten Ausdruck als ,,Ergebnis der Funktion`` an den Aufrufer zurück. Der Typ des Ausdrucks muß mit dem in der Schnittstelle angeführten Typ des Rückgabewerts übereinstimmen beziehungsweise entsprechend umgewandelt werden können.

Vorige Seite: 7 Funktionen **Eine Ebene höher:** 7 Funktionen **Nächste Seite:** 7.2 Deklaration und Definition

Vorige Seite: 7.1 Einführung **Eine Ebene höher:** 7 Funktionen **Nächste Seite:** 7.3 Funktionsaufruf und Parameterübergabe

7.2 Deklaration und Definition von Funktionen

Eine Funktionsdeklaration gibt den Namen der Funktion, den Typ des zurückgelieferten Funktionswerts sowie Anzahl und Typen der Parameter bekannt. Funktionsdeklarationen werden auch als *Prototypen* bezeichnet.

In der Parameterliste werden die einzelnen Parameter getrennt durch Beistriche notiert. Jeder einzelne Parameter ist bei der Deklaration (*nicht* bei der Definition!) durch seinen Typ bestimmt, der Name ist nicht nötig, kann aber angegeben werden. Der Parametername wird zwar vom Compiler ignoriert, ist allerdings von großer Bedeutung für die Lesbarkeit eines Programms. Dazu ein Beispiel:

Die Funktionsdeklaration

```
drawCircle(int xPos, int yPos, int rad);
```

entspricht syntaktisch der Deklaration

```
drawCircle(int, int, int);
```

Die erste Deklaration ist aber wesentlich besser lesbar, da bei einer entsprechenden Namensgebung bereits von der Deklaration der Funktion auf die Bedeutung der einzelnen Parameter geschlossen werden kann.

Jede Funktion muß *vor* ihrer Verwendung deklariert werden. Zu beachten ist, daß im Gegensatz zur einfachen Variablendeklaration eine Funktionsdeklaration noch keine Definition darstellt. Eine Funktionsdeklaration gibt Name, Parameterliste und Rückgabewert bekannt, vereinbart aber keinen Code (vergleiche Abschnitt 4.1)!

Eine weitere Besonderheit von Funktionen ist, daß sie nur auf ,,globalem`` Niveau vereinbart werden können. Das heißt, es ist nicht möglich, Funktionen innerhalb von Blöcken zu vereinbart werden. Damit kann auch keine Funktion eine andere Funktion enthalten.

Eine Funktionsdefinition umfaßt neben der Bekanntgabe der Funktionsschnittstelle auch den eigentlichen Funktionsrumpf, der in geschwungene Klammern (Blockanweisung) eingeschlossen ist.

Im Funktionsrumpf können (wie bereits erwähnt) eigene Variablen, sogenannte lokale Variablen, vereinbaren. Diese gelten ausschließlich innerhalb der Funktion und existieren nur so lange als nötig:

beim Funktionsaufruf werden sie angelegt, beim Verlassen der Funktion werden sie automatisch wieder zerstört.

Die lokalen Objekte existieren damit nur während der Ausführung von Funktionen und belegen auch nur während dieser Zeit Speicherplatz.

Daher sollten Objekte, die nur in einer Funktion von Bedeutung sind, auch lokal in dieser Funktion deklariert werden. Anders gesagt sollen Objekte in dem Gültigkeitsbereich deklariert werden, in dem sie verwendet werden.

Der Gültigkeitsbereich von Objekten sollte so groß als nötig und so klein als möglich sein. Objekte, die nur in Funktionen benötigt werden, sollten daher auch dort vereinbart werden.

```
char foo(int n, char ch);  // Funktionsdeklaration

  // Funktion kann ab hier verwendet werden,
  // da sie bereits deklariert ist.

char foo(int n, char ch)   // Funktionsdefinition
{
  int help1, help2;        // lokale Variablen

  help1 = n*12;
  if ((n<0) || (n>9)) {
    ...
  }
}                          // Ende der Funktion
```

Anmerkung: Eine Funktion kann mehrmals deklariert werden (sofern sich die Deklarationen nicht widersprechen), aber nur einmal definiert werden.

Ergänzend sollen zwei Begriffe im Umfeld von Funktionen erläutert werden:

- Der Typ einer Funktion besteht aus dem Typ des Funktionswerts (= Rückgabewert) und den Parametertypen.

  Beispiele:

  ```
  void funk(char);         // Typ void(char)
  char funk2(char, int);   // Typ char(char,int)
  ```

- Im Gegensatz dazu bezeichnet man die Kombination aus Funktionsnamen und Parameterliste als *Signatur*. Die Signatur einer Funktion ist insbesonders beim Überladen von Funktionen (Abschnitt 7.7) sowie beim Überschreiben von virtuellen Funktionen (Abschnitt 14.2.1) von Bedeutung.

Vorige Seite: 7.1 Einführung **Eine Ebene höher:** 7 Funktionen **Nächste Seite:** 7.3 Funktionsaufruf und Parameterübergabe

Vorige Seite: 7.2 Deklaration und Definition **Eine Ebene höher:** 7 Funktionen **Nächste Seite:** 7.4 Spezielle Funktionen

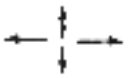
7.3 Funktionsaufruf und Parameterübergabe

Wird eine Funktion definiert, so wird Code für diese Funktion erzeugt. Allerdings wird dieser Code *nicht* automatisch ausgeführt. Um eine Funktion auszuführen, muß ein spezieller Operator verwendet werden, der Funktionsaufrufoperator `( )`. Dieser Operator wird nach dem Funktionsnamen notiert, in den runden Klammern werden die Parameter angegeben:

```
ch = intToChar(a); // ruft intToChar mit Parameter a auf
```

Im folgenden werden die beim Funktionsaufruf angegebenen Parameter als Aktualparameter bezeichnet, die in der Funktionsdeklaration angegebenen Parameter als Formalparameter. Ein Beispiel dazu:

```
int ggt(int a, int b)    // Formale Parameter a und b
{ ... }

void main()
{
  cout << ggt(x, y);          // Aktualparameter x, y
  cout << ggt(f, 1);          // Aktualparameter f, 1
  cout << ggt((3*12+4), 18); // Aktualparameter 40, 18
}
```

Der genaue Ablauf eines Funktionsaufrufs wird anhand des ersten Aufrufs im obigen Beispiel (`ggt(x, y)`) gezeigt (Sourcecode der Funktion siehe Seite ):

1. Die Programmausführung verzweigt zur aufgerufenen Funktion (`ggt`).
2. Die formalen Parameter (`a`, `b`) und die lokalen Variablen (in diesem Fall nur eine, `rest`) der Funktion `ggt` werden neu angelegt.

   Die eigentliche Parameterübergabe erfolgt so, daß von links nach rechts jedem formalen Parameter ein aktueller zugeordnet wird (`x` an `a`, `y` an `b`). Die formalen Parameter werden mit den Werten der korrespondierenden aktuellen Parameter initialisiert. Die formalen Parameter sind also

(zumindest am Beginn der Funktionsausführung) Kopien der aktuellen Parameter.

Die aktuellen Parameter können beliebige Ausdrücke sein, deren Typen mit denen der korrespondierenden formalen Parameter übereinstimmen beziehungsweise so umgewandelt werden können, daß sie übereinstimmen.

Diese Art der Parameterübergabe wird auch als *Call by Value* bezeichnet (es wird der „Wert`` des Aktualparameters übergeben, nicht der Parameter selbst).

Anmerkung: Die Reihenfolge der Auswertung der Argumentliste ist in der Sprachdefinition *nicht* festgelegt.

3.

Anschließend wird die eigentliche Funktion ausgeführt.

4.

Die Funktion wird durch die Anweisung `return a` verlassen. Die Anweisung bewirkt das Anlegen eines sogenannten *temporären Objekts* mit dem Wert von `a`.

Temporäre Objekte sind Objekte ohne Namen, die vom C++-Compiler erzeugt werden, um Werte von Ausdrücken oder ähnliches vorübergehend aufzunehmen. Sie werden vom System automatisch erzeugt und wieder zerstört, sobald sie nicht mehr benötigt werden.

Beim Verlassen der Funktion werden die lokalen Variablen und die Parameter wieder gelöscht.

5.

Dann wird an die Stelle nach dem Funktionsaufruf zurückgesprungen.

6.

Der Rückgabewert der Funktion (das vorher angelegte temporäre Objekt) wird verwendet und anschließend ebenfalls zerstört.

Da die formalen Parameter *Kopien* der aktuellen Parameter darstellen, können die formalen Parameter in der Funktion beliebig verändert werden, ohne daß dies Auswirkungen auf die Aktualparameter hat. So liefert

```
int x, y, erg;
x = 10; y = 22;
erg = ggt(x, y);
cout << x << y;
```

die Werte `10` und `22` als Ausgabe.

Auch wenn beim Funktionsaufruf Variablen verwendet werden, die dieselben Namen haben, wie die entsprechenden formalen Parameter, erfolgt keine Veränderung der Werte der aktuellen Parameter. Unabhängig von ihren Namen sind sie doch verschiedene Objekte! Das Beispiel

```
int a, b, erg;
a = 10; b = 22;
erg = ggt(a, b);
cout "ggt(" << a << ", " << b << ") = " << erg;
```

liefert folgende Ausgabe:

```
ggt(10, 22) = 2
```

Beim Aufruf von `ggt` werden *eigene* lokale Objekte für die formalen Parameter angegeben und die aktuellen Parameter (die ebenfalls `a` und `b` heissen) kopiert. Als eine Eselsbrücke kann hier dienen, daß alle Objekte ihrem Gültigkeitsbereich zugeordnet sind. Für das System existieren daher nicht einfach die Objekte `a` und `b`, sondern a_{ggt}, b_{ggt}, a_{global} und b_{global}. Damit ist auch klar, daß eine Namensgleichheit der Aktual- und Formalparameter keine Rolle spielt - für das C++-System sind dies verschiedene Objekte!

a_{ggt}, b_{ggt} werden in `ggt` verändert und beim Verlassen der Funktion wieder zerstört. Die Ausgabe von `a` und `b` im Beispiel bezieht sich *nicht* auf die formalen Parameter der Funktion `ggt`, die zu diesem Zeitpunkt auch nicht mehr existieren.

Besitzt die aufgerufene Funktion keine Parameter, so muß eine leere Argumentliste angegeben werden.

Achtung: Der Funktionsname ohne `()` stellt einen konstanten Zeiger auf die Anfangsadresse der Funktion dar und bewirkt keinen Aufruf (Siehe dazu Abschnitt 8.2.7).

```
ok = clearScreen(); // Aufruf der Prozedur clearScreen
                    // ohne Parameter
ok = clearScreen;   // Fehler, kein Funktionsaufruf!
```

Vorige Seite: 7.2 Deklaration und Definition **Eine Ebene höher:** 7 Funktionen **Nächste Seite:** 7.4 Spezielle Funktionen

Vorige Seite: 7.3 Funktionsaufruf und Parameterübergabe **Eine Ebene höher:** 7 Funktionen **Nächste Seite:** 7.4.1 Prozeduren

7.4 Spezielle Funktionen

C++ kennt zwei ,,spezielle`` Arten von Funktionen: Prozeduren und Operator-Funktionen.

Vorige Seite: 7.4 Spezielle Funktionen **Eine Ebene höher:** 7.4 Spezielle Funktionen **Nächste Seite:** 7.4.2 Operator-Funktionen

---

7.4.1 Prozeduren

Funktionen sind Einheiten, die Berechnungen durchführen und einen Wert zurückgeben. Funktionen *ohne* Rückgabewert werden oft als Prozeduren bezeichnet. Anders als andere Programmiersprachen (Pascal, Modula2 etc.) kennt C++ keine Prozeduren an sich, vielmehr sind Prozeduren Funktionen, die einen Rückgabewert vom Typ `void` liefern.

```
void printDay(int day, int month, int year)
{
  cout << "Das aktuelle Datum ist " << day << "."
       << month << "." << year << "\n";
}

void printUnderscores(int nr)
{
  for (int i=0; i<nr; i++) cout << "_";
  cout << "\n";
}
```

Der Aufruf einer Prozedur erfolgt ebenso wie der von Funktionen durch Nennung ihres Namens gefolgt von der Argumentliste:

```
prtDate(t, m, j);
prtDate(29, 11, 1967);
printUnderscores(n);
```

Auch Prozeduren können durch die `return`-Anweisung (vorzeitig) verlassen werden. Da Prozeduren aber keinen Ergebniswert liefern, darf hier kein Ausdruck nach dem Schlüsselwort `return` angegeben werden.

Beispiel:

```
void foo()
{
  ...
  if (errOccd) {
    return;
  }
  ...
```

```
}
```

Vorige Seite: 7.4.1 Prozeduren **Eine Ebene höher:** 7.4 Spezielle Funktionen

7.4.2 Operator-Funktionen

Operator-Funktionen sind Funktionen, die Operatoren implementieren. Ihr Name besteht aus dem Schlüsselwort `operator` gefolgt von dem eigentlichen Operator-Symbol. So wird zum Beispiel die Addition von zwei `int`-Werten mit dem Operator `+` durchgeführt, der (falls diese Operation nicht implizit vorhanden wäre) folgende Schnittstelle aufweisen würde:

```
int operator+(int left, int right);
```

Operatoren und vor allem die Definitionen ,,neuer`` Operatoren sind im Zusammenhang mit der Vereinbarung von eigenen Typen (Klassen) von Interesse und werden daher in Abschnitt 11.7.3 näher besprochen.

Vorige Seite: 7.4.2 Operator-Funktionen **Eine Ebene höher:** 7 Funktionen **Nächste Seite:** 7.6 Vorbelegte Parameter

---

7.5 `inline`-Funktionen

Funktionen sind eine Art ,,eigenständige`` Einheiten: Ihre Definition bewirkt die Erzeugung von Code, der über einen eigenen Operator, den Funktionsaufrufoperator, explizit aufgerufen werden muß. Das führt (normalerweise) zu Einsparungen bei der Codegröße, da auch bei mehrfacher Verwendung einer Funktion der Code nur einmal vorhanden ist.

Andererseits stellt der Aufruf einer Funktion einen gewissen *Overhead* dar: Interne Zustände müssen gesichert, Parameter und lokale Variablen neu angelegt und nach der Funktionsausführung wieder zerstört werden.

Für viele sehr einfache Funktionen ist dieser Aufwand größer als die durch die vermiedene Codeverdopplung erreichte Einsparung. Für derartige Fälle kennt C++ die sogenannten `inline`-Funktionen. Das Schlüsselwort `inline` weist den Compiler an, keinen Funktionsaufruf an sich zu erzeugen, sondern den gesamten Code der Funktion an der Stelle des Aufrufs direkt einzufügen. Damit wird der Overhead des Funktionsaufrufs vermieden.

Eine Funktion wird als `inline` spezifiziert, indem vor ihrem Returnwert das Schlüsselwort `inline` notiert wird. Funktionen sollten dann als `inline` spezifiziert werden, wenn sie vom Codeumfang her sehr klein sind und häufig aufgerufen werden (zum Beispiel in Schleifen). Bei richtiger Anwendung kann so eine Verminderung der Laufzeit (und eventuell auch der Größe des Objektcodes) erreicht werden.

Ein derartiges Problem ist zum Beispiel das Feststellen des Maximums zweier Werte.

```
inline int max(int a, int b)
{
  if (a >= b)
    return a;
  else
    return b;
}
```

Eine Anmerkung bezüglich C++-`inline`-Funktionen und C-Makros: In C wird ein derartiges Problem üblicherweise durch die Implementierung eines Makros (Makros werden in Anhang B besprochen) gelöst. Obige `inline`-Funktion kann etwa wie folgt als Makro implementiert werden:

```
#define MAX(a,b)            (((a)>(b)) ? (a) : (b))
```

Makros und `inline`-Funktionen haben (zumindest auf den ersten Blick) dieselbe Wirkung: es wird kein Funktionsaufruf erzeugt, sondern der Code direkt eingefügt. Allerdings werden Makros vom Präprozessor expandiert, das heißt es finden *keinerlei* Typprüfungen statt und die Expandierung liegt nicht im Bereich des eigentlichen C++-Compilers. Das ist auch das größte Problem im Zusammenhang mit Makros.

Der Einsatz von Makros sollte in C++ soweit als irgend möglich unterbleiben. `inline`-Funktionen ersetzen die klassischen C-Makros zum größten Teil und haben den Vorteil, daß sie Teil der Sprache C++ sind.

Zudem muß beachtet werden, daß die meisten der modernen Compiler über entsprechende Optimierungsfunktionen verfügen und ein automatisches *Inlining* vornehmen können.

Optimierungen in bezug auf Codegröße oder Effizienz sollten nur dann vorgenommen werden, wenn ein Effizienz-Problem vorliegt. Allerdings sollten selbst dann nur gezielt jene Stellen des Programms optimiert werden, die den Engpaß verursachen.

Vorige Seite: 7.4.2 Operator-Funktionen **Eine Ebene höher:** 7 Funktionen **Nächste Seite:** 7.6 Vorbelegte Parameter

Vorige Seite: 7.5 `inline`-Funktionen **Eine Ebene höher:** 7 Funktionen **Nächste Seite:** 7.7 Überladen von Funktionen

7.6 Vorbelegte Parameter

Wie bereits erwähnt, erfolgt die Parameterübergabe so, daß von links beginnend die einzelnen Aktualparametern den entsprechenden Formalparametern zugeordnet werden. In C++ können Formalparameter mit Werten vorbelegt werden. Die Standardwerte werden im Funktionsaufruf automatisch verwendet, wenn weniger Aktualparameter als Formalparameter angegeben sind.

Anmerkung: Vorbelegte Parameter werden auch als *Default*-Argumente bezeichnet.

Da aber die Zuordnung der Parameter in jedem Fall von links nach rechts erfolgt, ist zu beachten, daß hinter keinem vorbelegten formalen Parameter ein *nicht vorbelegter* Parameter folgt. In den folgenden Beispielen werden Funktionen mit vorbelegten Parametern deklariert und anschließend aufgerufen:

```
void prtDate0(int day=1, int month=3, int year=1991);
void prtDate1(int day, int month=3, int year=1991);
```

`prtDate0` und `prtDate1` können mit einem, zwei oder drei Argumenten aufgerufen werden, `prtDate0` kann auch ohne Parameter aufgerufen werden, da alle Parameter vorbelegt sind.

```
prtDate0();           // 3 Default-Argumente: 1-3-1991
prtDate1(8);          // 2 Default-Argumente: 8-3-1991
prtDate1(8, 2);       // 1 Default-Argument: 8-2-1991
prtDate1(8, 2, 1990); // 3 Argumente: 8-2-1990
```

Folgende Deklarationen sind fehlerhaft, da nach einem vorbelegten Formalparameter mindestens ein nicht vorbelegter folgt. Damit können keine korrekten Zuordnungen von Aktual- und Formalparametern erfolgen:

```
   // zwei falsche Deklarationen:
void prtDate2(int day=7, int month, int year=1991);
void prtDate3(int day, int month=3, int year);
```

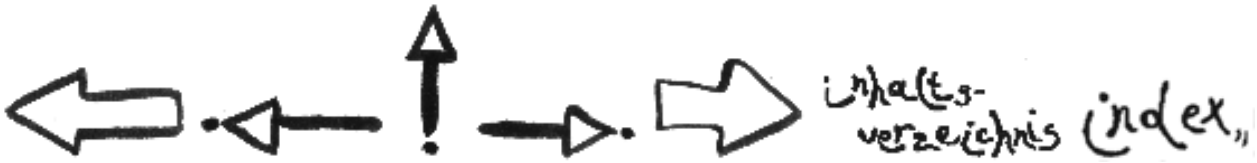
Zu beachten ist auch, daß ein *Default*-Wert für einen Parameter nur einmal angegeben werden kann. Üblich ist, daß *Default*-Parameter ausschließlich im Prototyp spezifiziert werden. Die Funktionsdefinition enthält nur die Parameter ohne vorbelegte Werte. Im folgenden sind Beispiele für richtige und falsche Funktionsdeklarationen (= Prototypen) mit ihren zugehörigen Definitionen angegeben:

```
void foo1(int d, int m=3, int y=1);
void foo1(int d, int m=3, int y=1) { ... }
  // falsch, m,y sind zweimal vorbelegt

void foo2(int d, int m, int y=1);
void foo2(int d, int m, int y) { ... }
  // OK, Default-Werte im Prototyp

void foo3(int d, int m=3, int y=1);
void foo3(int d=1, int m, int y) { ... }
  // syntaktisch richtig, jeder Parameter hat
  // einen Default-Wert: y,m aus dem Prototyp
  // d ist daher der letzte Param. ohne
  // Default-Wert in der Def. und kann einen
```

```
  // Default-Wert erhalten

void foo4(int d, int m, int y=1);
void foo4(int d=1, int m, int y) { ... }
  // falsch, da d in der Definition nicht der
  // letzte Parameter ohne Default-Wert ist.
```

Default-Argumente können unter anderem dann von Nutzen sein, wenn in eine bereits existierende Funktion weitere Argumente aufgenommen werden sollen, ohne daß die Syntax von bestehenden Funktionsaufrufen verändert werden muß. Dazu werden die neuen Argumente als *Default*-Parameter in die Funktionsdeklaration aufgenommen (die korrespondierende Funktionsdefinition muß dann angepaßt werden).

Im Hinblick auf getrennte Übersetzung und die Erstellung größerer Programmsysteme (Zerlegung in Header- und Programmdateien), sollten *Default*-Werte von Parametern *immer* in den entsprechenden Prototypen und *nicht* in den Funktionsdefinitionen angeführt werden.



Vorige Seite: 7.5 `inline`-Funktionen **Eine Ebene höher:** 7 Funktionen **Nächste Seite:** 7.7 Überladen von Funktionen

Vorige Seite: 7.6 Vorbelegte Parameter **Eine Ebene höher:** 7 Funktionen **Nächste Seite:** 7.7.1 Aufruf von überladenen

7.7 Überladen von Funktionen

Bezeichner müssen in ihrem Gültigkeitsbereich eindeutig sein. So ist es zum Beispiel nicht möglich, zwei Variablen oder eine Variable und eine Funktion mit demselben Namen in einem Gültigkeitsbereich zu deklarieren. In bezug auf die Unterscheidung von Funktionen gilt allerdings eine etwas andere Regelung als in Programmiersprachen wie Pascal, Modula2 oder C: Der Compiler identifiziert Funktionen nicht allein über ihren Namen, sondern über ihre gesamte Signatur (Abschnitt 7.2).

Dieser auf den ersten Blick nicht unbedingt einsichtige Mechanismus bringt wesentliche Vorteile. Funktionen, die vergleichbare Aktionen ausführen, aber verschiedene Parameter haben, können in C++ denselben Namen aufweisen. So können zum Beispiel verschiedene Funktionen `print` vereinbart werden:

```
void print(char ch);
void print(int i);
void print(double d);
```

Jede der drei Funktionen hat zwar den Namen `print`, ist aber anders implementiert, da die Ausgabe von `ch`, `i` und `d` aufgrund ihrer Datentypen verschieden ist.

Das Konzept des Überladens bringt eine höhere Verständlichkeit von Programmen, da vergleichbare Aktionen gleich benannt werden können. Zudem können selbst vereinbarte Datentypen mit überladenen Funktionen besser integriert werden. (Kapitel 11 zeigt ein Beispiel für die Integration eines eigenen Datentyps.)

Offensichtlich werden die Vorteile des Mechanismus des Überladens bei Operator-Funktionen: `<<` ist eine überladene Operator-Funktion, die ihr Argument auf dem jeweiligen Zeichenstrom ausgibt. Jedes Objekt wird mit dieser Operator-Funktion ausgegeben. Auch die Ausgabe von Objekten von eigenen Datentypen wird über `<<` realisiert.

Anmerkung: Man vergleiche dies mit der Ausgabe in Oberon, wo für jede Ausgabeprozedur ein anderer Funktionsname verwendet werden muß: `Out.Char`, `Out.String` etc.

Werden in C++ mehrere Funktionen mit dem gleichen Namen deklariert, so gilt:

- Entsprechen Rückgabewert und Parameterliste der zweiten Deklaration denen der ersten, so wird die zweite als gültige Re-Deklaration der ersten aufgefaßt.
- Unterscheiden sich die beiden Deklarationen nur bezüglich ihrer Rückgabewerte, so behandelt der

Compiler die zweite Deklaration als fehlerhafte Re-Deklaration der ersten. Da sich Funktionssignaturen nur auf Name und Parameterliste beziehen, kann der Rückgabewert von Funktionen nicht als Unterscheidungskriterium verwendet werden.

- Nur wenn beide Deklarationen sich in Anzahl oder Typ ihrer Parameter unterscheiden, werden sie als zwei verschiedene Deklarationen mit demselben Funktionsnamen betrachtet (überladene Funktionen).

Für den Compiler sind überladene Funktionen nichts anderes als eine Menge von unabhängigen Funktionen, die sich in ihren Parametern unterscheiden.

Vorige Seite: 7.6 Vorbelegte Parameter **Eine Ebene höher:** 7 Funktionen **Nächste Seite:** 7.7.1 Aufruf von überladenen

Vorige Seite: 7.7 Überladen von Funktionen **Eine Ebene höher:** 7.7 Überladen von Funktionen
Nächste Seite: 7.7.2 Überladen versus vorbelegte

7.7.1 Aufruf von überladenen Funktionen

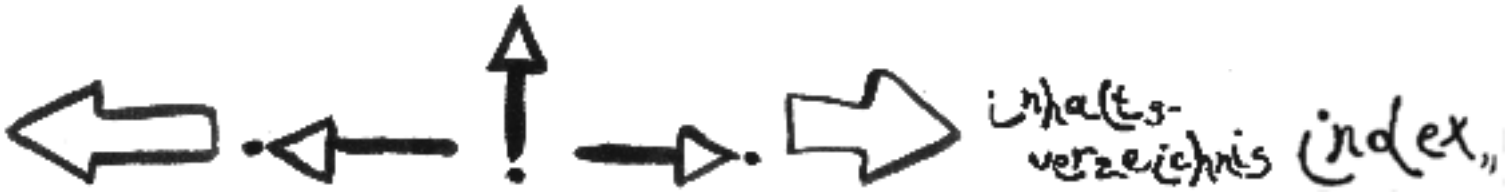
Um unter den vorhandenen Kandidaten für eine Funktion (= Funktionsname stimmt überein) den ,,Richtigen`` zu finden, geht der Compiler vereinfacht gesagt wie folgt vor [Str92, S. 143]:

1. Findet der Compiler eine Funktion, die ,,exakt paßt``, das heißt eine Funktion, deren Signatur ohne jegliche Typumwandlungen mit der des Funktionsaufrufs übereinstimmt, so wird diese verwendet.
2. Als nächstes versucht der Compiler, eine Funktion zu finden, deren Profil nach Anwendung von integraler Promotion (siehe Abschnitt 9.4.1) mit dem geforderten übereinstimmt.
3. Nach der Anwendung von integraler Promotion greift der Compiler auf Standard-Typumwandlungen zurück, um eine entsprechende Funktion zu finden.
4. Wird durch die Anwendung von vordefinierten Typumwandlungen keine passende Funktion gefunden, so sucht der Compiler mittels benutzerdefinierten Typumwandlungen (siehe Abschnitt 11.7.5) nach einer passenden Funktion.
5. Abschließend wird versucht, eine Funktion zu finden, in deren Schnittstelle eine sogenannte Ellipse beliebige Parameter zuläßt. Als Ellipse werden drei Punkte (`...`) bezeichnet. Funktionen mit Ellipsen als Parameter können mit beliebigen Aktualparametern aufgerufen werden. Ihre Auswertung wird hier aber nicht beschrieben, dazu sei auf die Dokumentation des Entwicklungssystems verwiesen.

Beispiele dazu:

```
void print(char ch);     // Fkt1
void print(int i);       // Fkt2
void print(double d);    // Fkt3

int i = 23;
print('x');             // ruft Ftk1 auf
print(1);               // ruft Ftk2 auf
```

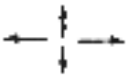
```
print(i);                  // ruft Ftk2 auf
print(1.234);              // ruft Ftk3 auf
print(true);               // ruft Fkt2 auf (integrale
                           // Promotion bool->int)
```



Vorige Seite: 7.7 Überladen von Funktionen **Eine Ebene höher:** 7.7 Überladen von Funktionen
Nächste Seite: 7.7.2 Überladen versus vorbelegte

Vorige Seite: 7.7.1 Aufruf von überladenen **Eine Ebene höher:** 7.7 Überladen von Funktionen

7.7.2 Überladen versus vorbelegte Parameter

Oft kann derselbe Effekt wie beim Überladen von Funktionen auch einfach durch *Default*-Parameter erreicht werden. In solchen Fällen ist der Einsatz von *Default*-Parametern vorzuziehen, da auf diese Art nur eine einzelne Funktion codiert wird (*Single Source*-Prinzip, siehe auch Kapitel 12, Seite ). Keinesfalls sollten aber in überladenen Funktionen *Default*-Parameter verwendet werden.

Als generelle Regel kann angeführt werden, daß *Default*-Argumente verwendet werden, wenn sich die verschiedenen Funktionen nicht grundlegend in ihren Parametertypen unterscheiden. Andernfalls sind überladene Funktionen zu verwenden [Pap95, S. 44].

Funktionen sollten nur dann überladen werden, wenn

- sie eine vergleichbare Operation bezeichnen, die jeweils mit anderen Parametertypen ausgeführt wird,
- dieselbe Wirkung nicht durch *Default*-Parameter erreicht werden kann und
- dadurch keine Information im Programmtext verloren geht.

Die drei überladenen Funktionen

```
void print(int i);
void print(int i, int width);
void print(int i, char fillChar, int width);
```

zum Beispiel können auch durch eine einzige Funktion mit *Default*-Parameter ersetzt werden:

```
void print(int i, int width=0, char fillChar=0x00);
```

Vorige Seite: 7.7.2 Überladen versus vorbelegte **Eine Ebene höher:** 7 Funktionen **Nächste Seite:** 8 Höhere Datentypen und

7.8 Rekursion

Eine Funktion, die sich selbst direkt oder indirekt (über den Umweg anderer Funktionen) aufruft, wird als *rekursiv* bezeichnet. Viele mathematische Funktionen sind rekursiv definiert. Als Beispiel sei die Fakultätsfunktion angegeben:

$$x! = \begin{cases} 1, \text{ wenn } x=0 \\ (x-1)!x, \text{ wenn } x>0 \end{cases}$$

Obige Definition kann einfach in eine C++-Funktion umgesetzt werden:

```
long fak(int n)
{
  if (n == 0) {
    return 1;
  } else {
    return fak(n-1)*n;
  }
}
```

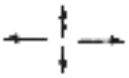
Der Funktionswert `fak(n)` ist damit über den Funktionswert `fak(n-1)` erklärt. Dieses Zurückgreifen von Fall `n` auf den Fall `(n-1)` heißt Rekursionsschritt. Damit sich das rekursive Zurückrechnen nicht unbegrenzt fortsetzt, muß ein Wert (hier der Funktionswert für `n=0`) vorgegeben sein. Dieser Wert wird *Rekursionsbasis* oder auch Anker genannt. Ohne die Rekursionsbasis würde das Verfahren in eine Endlosschleife geraten. Es liegt in der Verantwortung des Programmierers, für den korrekten Abbruch eines rekursiven Verfahrens zu sorgen.

Der genaue Ablauf beim Aufruf `x = fak(4)` ist wie folgt:

1. Die Funktion `fak` wird mit `n==4` betreten. Beim Aufruf werden die lokalen Objekte und die Parameter neu angelegt und initialisiert. Da `n!=0` ist, wird die Funktion `fak` mit `n-1` (also `3`) erneut aufgerufen.

2.

Der Aufruf bewirkt, daß eine *neue* Generation des Parameters `n` angelegt und mit `3` initialisiert wird.

Die auf Seite  angeführte Verbindung von Variablennamen und ihrem Gültigkeitsbereich wird also in bezug auf Funktionen noch um die Anzahl der *aktiven* Instanzen einer Funktion (= der gerade ausgeführten) erweitert. In unserem Fall existieren also die Objekte `n`*fak 1* (= der ,,alte`` Parameter) und `n`*fak 2* (= der ,,neue`` Parameter). `n`*fak 1* wird von `n`*fak 2* überdeckt - der alte Parameter existiert nach wie vor, ist aber nicht mehr sichtbar. Da `n>0` ist, erfolgt wieder ein Aufruf von `fak` mit `n-1`.

3.

Dies wiederholt sich, bis die Prozedur `fak` mit dem Wert `0` aufgerufen wird. In diesem Fall wird der Wert `1` zurückgegeben. Die Rückgabe bewirkt einen Rücksprung an die Stelle, von der `fak(0)` aufgerufen wurde, also zur Anweisung `return(n*fak(n-1))` wobei `n` den Wert `1` hat und der Ausdruck `fak(n-1)` den Wert 1 lieferte. Es wird also der Wert `1*1` zurückgegeben.

4.

Das Ganze wiederholt sich, bis die Ausführung bei der ersten Aufruf-Generation von `fak` (`n==4`) anlangt. Die Rückgabe des Werts `4*3` bewirkt den Rücksprung zum Rufer, also in das Hauptprogramm.

Ein weiteres Beispiel für eine rekursive Prozedur ist das Invertieren (= Umdrehen) einer beliebig langen Zeichenkette:

```
void revertString()
{
  char  ch;

  cin >> ch;
  if (ch != 'X') {
    revertString();
    cout << ch;
  }
}
```

`revertString` hat eine lokale Variable `ch`, in die ein Zeichen eingelesen wird. Ist das Zeichen ungleich `'X'` (= Abbruchzeichen), so ruft sich die Prozedur selbst wieder auf und gibt *anschließend* das gelesene Zeichen aus. Da die Ausgabe nach dem rekursiven Aufruf erfolgt, erscheinen die gelesenen Zeichen in der umgekehrten Reihenfolge am Bildschirm.

Rekursion ist ein sehr mächtiges Konzept. So können etwa alle Probleme der Informatik, die iterativ (also ohne Rekursion) lösbar sind, auch rekursiv gelöst werden. Es sollte aber beachtet werden, daß rekursive Lösungen meist ineffizienter (Speicherverbrauch, Geschwindigkeit) und oft schwer verständlich sind. So kann zum Beispiel das oben angeführte Problem der Fakultätsberechnung iterativ

wesentlich effizienter gelöst werden:

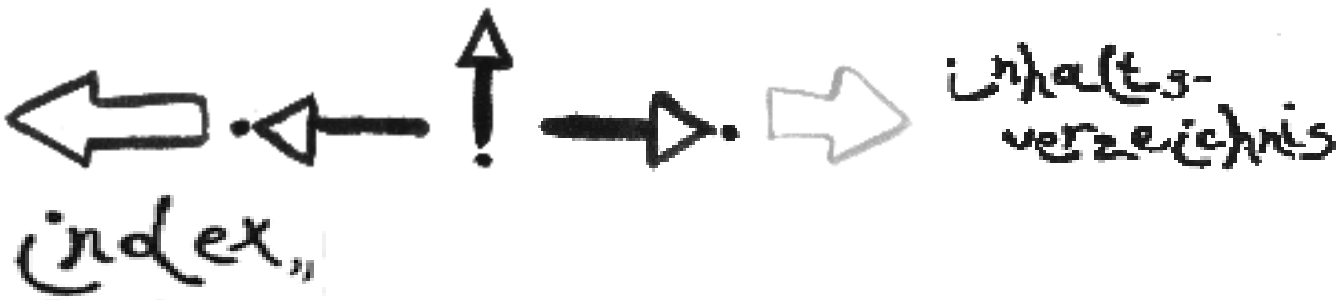
```
long fak(int n)
{
  long r=1;

  for (int i=1; i<n; ++i) {
    r *= i;
  }
  return r;
}
```

Auch viele der anderen oft in Lehrbüchern angeführten Beispiele wie etwa die Berechnung der Potenz einer Zahl sind keine typisch rekursiven Probleme. Andererseits existiert eine nicht unbeträchtliche Anzahl von Problemen, die rekursiv wesentlich einfacher und besser gelöst werden können. Beispiele dafür sind etwa alle Baumstrukturen, rekursive Sortierroutinen (Quicksort), das Spiel ,,Türme von Hanoi`` oder die Berechnung von Binomialkoeffizienten.

Zusammenfassend kann gesagt werden:

Rekursion ist ein mächtiges Konzept, das zur Lösung ,,rekursiver Probleme`` eingesetzt werden sollte. Der Einsatz von Rekursion bei der Lösung von iterativen Problemen sollte unterbleiben.

Für eine genauere Abhandlung der Rekursivität sei auf entsprechende Literatur verwiesen (zum Beispiel [Sed92]).



Vorige Seite: 7.7.2 Überladen versus vorbelegte **Eine Ebene höher:** 7 Funktionen **Nächste Seite:** 8 Höhere Datentypen und

Vorige Seite: 7.8 Rekursion **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** 8.1 Referenzen

---

8 Höhere Datentypen und strukturierte Datentypen

Höhere Datentypen werden auf der Basis von bestehenden Datentypen gebildet. C++ kennt Referenztypen, Zeiger und Vektoren. Als strukturierte Datentypen werden alle Klassen, Strukturen und ihre Unterarten bezeichnet. In diesem Abschnitt werden die höheren Datentypen im Detail besprochen. Über die strukturierten Datentypen wird lediglich ein erster Überblick gegeben, auf sie wird in Kapitel 11 näher eingegangen.

Das Kapitel ist wie folgt gegliedert:

- Zunächst werden Referenzen und Zeiger besprochen.
- Im Anschluß daran werden Vektoren, der Zusammenhang zwischen Vektoren und Zeigern sowie eine spezielle Art von Vektoren (Zeichenketten) dargelegt.
- Danach wird die Verwendung von Zeigern und Referenzen bei der Parameterübergabe gezeigt. Wie bereits in Kapitel 7 angeführt wurde, sind in C++ Aktualparameter grundsätzlich Kopien der Formalparameter. Mit Zeigern und Referenzen kann jedoch auch die von Pascal bekannte *Call by Reference*-Parameterübergabe realisiert werden.
- Den Abschluß bildet eine erste Übersicht über die Klassentypen.

Vorige Seite: 8 Höhere Datentypen und **Eine Ebene höher:** 8 Höhere Datentypen und **Nächste Seite:** 8.2 Grundlegendes zu Zeigern

---

8.1 Referenzen

Unter einer Referenz versteht man einen alternativen Namen für ein Objekt. Anders gesagt bezeichnet eine Referenz kein konkretes Objekt, sondern ein sogenanntes *Alias*. Daher muß ein Referenz-Objekt beim Anlegen mit einem bestehenden Objekt initialisiert werden. Referenz-Objekte werden deklariert, indem der eigentlichen Typangabe der Begrenzer & angefügt wird.

Dabei gibt es zwei ,,verschiedene`` Möglichkeiten, Referenzen zu deklarieren: Das Zeichen & kann der Typangabe folgen oder vor dem Variablennamen stehen. Oft wird die zweite Schreibweise verwendet, [HN93] empfehlen allerdings, das Referenzzeichen direkt hinter die Typangabe zu setzen. Damit wird ausgedrückt, daß es sich nicht um eine Variable vom Typ `T`, sondern vom Typ `T&` handelt.

Es ist aber zu beachten, daß sich & nur auf das erste nachfolgende Objekt bezieht. Daher sollten Referenz-Objekte bei Verwendung der ersten Schreibweise jeweils einzeln vereinbart werden.

Beispiele für die Möglichkeiten der Vereinbarung von Referenzen:

```
int  x = 12;
                    // Schreibweise 1: & nach Typangabe
int& r1=x;
                    // Schreibweise 2: & vor Var-Namen:
int  &r2=x, &r3=x; // zwei int-Referenzen r2, r3

int& r4=x, r5=x;   // r5 ist keine Referenz!
                    // (& bezieht sich nur auf r4)

int& r4=x;          // Besser: zwei Deklarationen
int& r5=x;
```

Referenzen werden deklariert, indem der &-Begrenzer direkt hinter der Typangabe notiert wird. Dadurch wird deutlich angezeigt, daß zum Beispiel kein `int`- sondern ein `int&`-Objekt vereinbart wird.

Folgendes Beispiel zeigt, daß eine Referenz kein eigenes Objekt darstellt:

```
float  x = 12.34;
float& y = x;      // x und y beziehen sich auf
                   // das gleiche int-Objekt
float  z = y;      // z = 12.34;
y = 9.81;          // entspricht: x = 9.81;

cout << x << ", " << y << ", " << z;
```

Die Ausgabe ist wie folgt:

```
9.81, 9.81, 12.34
```

Schon dieses kleine Beispiel illustriert, daß Referenzen sehr oft zu unübersichtlichen und schwer verständlichen Programmen führen. Referenzen werden vor allem bei der Parameterübergabe (Abschnitt 8.5) verwendet - eine andere Verwendung als *Alias*-Objekt sollte weitgehend unterbleiben.

Referenzen können auch als Rückgabewerte von Funktionen verwendet werden. In diesem Fall wird bei der Rückgabe des Werts kein neues Objekt angelegt, sondern nur ein *Alias* für das jeweils bestehende Objekt zurückgegeben:

```
int i=7;

int& getI()
{
  return i;
}

void testFoo()
{
  getI() = 27; // ok, da getI eine Referenz liefert
}
```

- Der Gebrauch von Referenz-Objekten (außer als Parameter und Rückgabewerte) sollte aus Gründen der Verständlichkeit weitgehend unterbleiben.

Vorige Seite: 8 Höhere Datentypen und **Eine Ebene höher:** 8 Höhere Datentypen und **Nächste Seite:** 8.2 Grundlegendes zu Zeigern

Vorige Seite: 8.1 Referenzen **Eine Ebene höher:** 8 Höhere Datentypen und **Nächste Seite:** 8.2.1 Deklaration von Zeigern

---

8.2 Grundlegendes zu Zeigern

Um eine Variable verwalten zu können, muß das System ihren Wert, ihre Größe und den Platz (die Adresse), an der die Variable im Speicher abgelegt ist, kennen. Das Problem dabei ist, daß durch die Deklaration Umfang und Struktur der Daten fix vorgegeben sind. Damit müssen bereits zum Zeitpunkt der Programmerstellung alle Angaben bezüglich Größe und Art der zu verwaltenden Daten festgelegt werden.

Eine Möglichkeit, die sich dadurch ergebenden Einschränkungen zu umgehen, ist es, Variablen über ihre Adresse im Speicher zu verwalten. Dazu werden sogenannte Zeiger (*Pointer*) verwendet, die keinen konkreten Wert, sondern die Adresse einer anderen Variablen enthalten. Eine Zeigervariable ,,verweist`` auf ein anderes Objekt. Mit diesem Konzept lassen sich einerseits Zeitpunkt des Anlegens und Zerstörens von Objekten beliebig kontrollieren, als auch andererseits flexible Datenstrukturen festlegen.

Abbildung 8.1 zeigt einen Zeiger `p`, der auf ein `int`-Objekt verweist: `p` enthält keinen unmittelbar verwendbaren Wert, sondern vielmehr die Adresse des `int`-Objekts. Soll das `int`-Objekt verwendet werden, so erfolgt dies über die in `p` gespeicherte Adresse. Dieser Schritt wird auch als ,,Indirektion`` bezeichnet, da auf die eigentlichen Objekte im Speicher nicht direkt, sondern indirekt über den Zeiger zugegriffen wird.

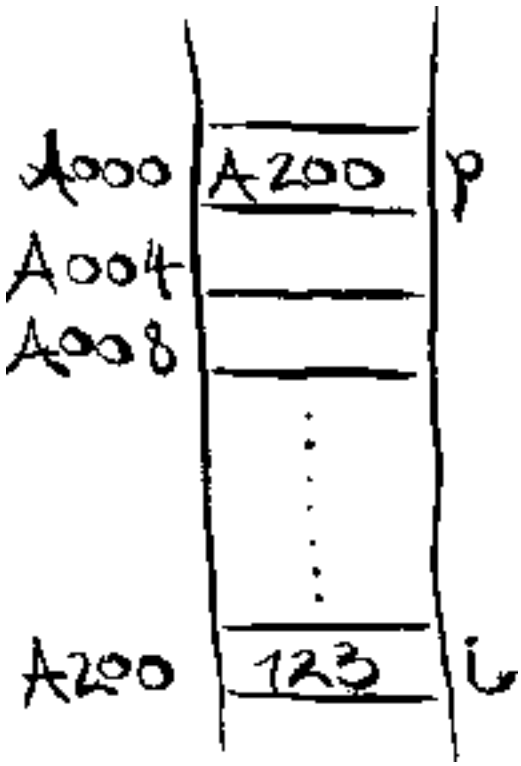


Abbildung 8.1: Zeiger

Folgende Operationen stehen im Zusammenhang mit Zeigern zur Verfügung:

1. Deklaration von Zeiger-Objekten
2. Feststellen der Adressen von Objekten (mittels Adreßoperator)
3. Ansprechen der Objekte, auf die Zeiger verweisen (Indirektionsoperator)
4. Anlegen (Allokieren) von Objekten, auf die Zeiger verweisen (`new`-Operator)
5. Zerstören von dynamisch angelegten Objekten (`delete`-Operator)

Diese Operationen werden im folgenden näher erläutert. Im Anschluß daran werden die verschiedenen Arten von konstanten Zeigern und zwei spezielle Zeigertypen besprochen.

Vorige Seite: 8.1 Referenzen **Eine Ebene höher:** 8 Höhere Datentypen und **Nächste Seite:** 8.2.1 Deklaration von Zeigern

Vorige Seite: 8.2 Grundlegendes zu Zeigern **Eine Ebene höher:** 8.2 Grundlegendes zu Zeigern **Nächste Seite:** 8.2.2 Adreßbildung

8.2.1 Deklaration von Zeigern

Grundsätzlich kann zwischen Zeigern auf (Speicher-)Objekte, `void`-Zeigern, Zeigern auf Funktionen, Zeigern auf Methoden und Zeigern auf Klassen-Elemente unterschieden werden. Hier sind zunächst Zeiger auf Objekte von Interesse, die anderen Arten von Zeigern werden später besprochen.

Zeiger sind in C++ typgebunden, das heißt, jeder Zeiger verweist auf ein Objekt bestimmten Typs. Dazu werden Zeiger mit dem Typnamen gefolgt vom Begrenzer * deklariert:

```
int*    pi;   // Zeiger auf eine int-Variable
char*   pch;  // Zeiger auf eine char-Variable
float*  pf;   // Zeiger auf eine float-Variable
```

Durch die Angabe des Typs werden die Zeigervariablen ,,gebunden``, das heißt, ein `int`-Zeiger kann (im Prinzip) ausschließlich auf `int`-Objekte verweisen. Nur durch Typumwandlungen können typgebundene Zeiger dazu verwendet werden, auf andersartige Objekte zu verweisen. Allerdings liegt es dann in der Verantwortung des Programmierers, die korrekte Verwendung derartiger Zeiger sicherzustellen.

Wie im Falle von Referenzen gibt es auch hier zwei Schreibweisen: Der *-Begrenzer kann der Typangabe folgen oder vor dem Variablennamen stehen.

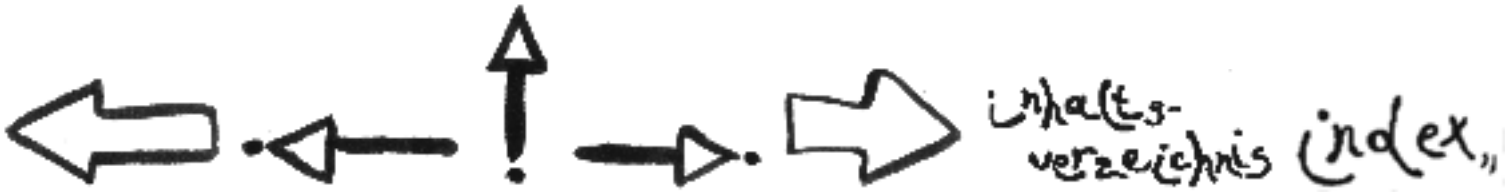
In C ist die zweite Schreibweise üblich, in C++ eher die erste (Sternoperator folgt unmittelbar nach dem Typnamen). Dabei ist zu beachten, daß sich * nur auf das erste nachfolgende Objekt bezieht. Daher sollten Zeiger-Objekte bei Verwendung der zweiten Schreibweise jeweils einzeln vereinbart werden.

Beispiele für die Möglichkeiten der Vereinbarung von Zeigern:

```
int* v1;         // Schreibweise 1: * nach Typangabe
int  *u1, *u2;   // Schreibweise 2: * vor Variable
                 // zwei int-Zeiger u1 und u2

int* w1, w2;     // Nur w1 ist ein Zeiger, w2 ist ein
                 // int, da * sich nur auf w1 bezieht.
int* w1;         // Daher: zwei Deklarationen
int* w2;
```

Zeiger enthalten nach ihrer Vereinbarung keinen speziellen Wert.

Zeiger werden deklariert, indem der `*`-Begrenzer direkt hinter der Typangabe notiert wird. Dadurch wird deutlich angezeigt, daß zum Beispiel kein `int`-, sondern ein `int*`-Objekt vereinbart wird.



Vorige Seite: 8.2 Grundlegendes zu Zeigern **Eine Ebene höher:** 8.2 Grundlegendes zu Zeigern **Nächste Seite:** 8.2.2 Adreßbildung

Vorige Seite: 8.2.1 Deklaration von Zeigern **Eine Ebene höher:** 8.2 Grundlegendes zu Zeigern **Nächste Seite:** 8.2.3 Dereferenzierung von Zeigerwerten

8.2.2 Adreßbildung

Der unäre Adreßoperator & kann auf im Speicher befindliche Operanden oder auf Funktionen angewandt werden und liefert deren Adresse.

In dem folgenden Beispiel ist `x` eine Variable vom Typ `int` und `px` ein Zeiger. Die Anweisung

```
px = &x;
```

weist also die Adresse von `x` der Variablen `px` zu. Bei einem Operandentyp `T` ist das Ergebnis der Operation vom Typ `T*`, also ,,Zeiger auf `T``.

```
float   n;
float*  pfloat;  // pfloat ist ein Zeiger auf float
...
n = 12.5;
pfloat = &n;     // nun zeigt pfloat auf n
```

Zu beachten ist in diesem Zusammenhang, daß temporäre Objekte nur vorübergehend ,,leben`` und ihre Adresse nicht bestimmt werden kann.

- Ein temporäres Objekt, wie es beim Berechnen eines Ausdrucks entsteht, hat keine fixe Adresse, die über den Adreßoperator bestimmt werden kann.

Vorige Seite: 8.2.2 Adreßbildung **Eine Ebene höher:** 8.2 Grundlegendes zu Zeigern **Nächste Seite:** 8.2.4 Anlegen von dynamischen

8.2.3 Dereferenzierung von Zeigerwerten

Mit dem unären Indirektionsoperator * kann eine Adreßbezugnahme aufgelöst werden, das heißt, es wird über einen Zeiger indirekt auf das Objekt zugegriffen, auf das er verweist. Folgendes Beispiel legt einen Zeiger p an, dem die Adresse des int-Objekts x zugewiesen wird.

```
int  x, y;
int* p;     // p ist vom Typ Zeiger auf int
...
x = 10;
p = &x;     // p zeigt nun auf x
```

Nachdem dem Zeiger p die Adresse von x zugeordnet wurde, kann über p auf x zugegriffen werden:

```
y = *p;              // entspricht y = x
```

Nach dieser Operation besitzt y den gleichen Wert wie x, nämlich zehn. Verweist p auf x, so kann der Ausdruck *p überall dort verwendet werden, wo in Ausdrücken x vorkommen kann. Der Indirektionsschritt, das heißt die Auflösung der Adresse einer Zeigervariable, wird auch als ,,Dereferenzierung`` bezeichnet.

```
x = *p - 1;   // x hat den Wert 9
*p = *p - 1;  // x hat den Wert 8 (p verweist auf x)
*p += 2;      // x hat den Wert 10
```

Wichtig ist,daß Zeiger nur dann dereferenziert werden dürfen, wenn sie auf ein gültiges Speicherobjekt verweisen. Beim Anlegen eines Zeigers zum Beispiel erhält dieser keinen speziellen Wert. Wird ein derartiger Zeiger dereferenziert, so wird auf eine ,,zufällige`` Speicheradresse zugegriffen. Abgesehen davon, daß dies nicht sehr sinnvoll ist, wird damit sehr wahrscheinlich eine Speicherschutzverletzung begangen (Zugriff auf Speicher, der nicht dem eigenen Programm gehört).

Mit dem Wert 0 wird angezeigt, daß ein Zeiger auf kein Objekt verweist. Ein Zeiger, der diesen Wert hat, darf auf keinen Fall dereferenziert werden.

```
int* p;
int* q;
...
cout << *p;    // Fehler! p ist nicht initialisiert ->
               //  -> Speicherverletzung!
q = 0;         // q initialisieren
```

```
...
if (q == 0) {  // Abfrage auf 0-Wert
  ...
```

In C wird häufig die Konstante `NULL` anstelle von 0 verwendet, die mit `(void*)0` oder `0` definiert ist. Ist die Konstante mit `(void*)0` vorbelegt, so führt dies in C++ zu unnötigen Problemen, da eine *Cast*-Operation nötig ist, um einen `void*`-Wert einem beliebigen Zeiger zuzuweisen. Ist `NULL` hingegen mit `0` vordefiniert, so ist die Konstante überflüssig.

Anstelle von `NULL` wird der Wert `0` verwendet, um anzuzeigen, daß ein Zeiger auf keine gültige Speicheradresse verweist.

Vorige Seite: 8.2.2 Adreßbildung **Eine Ebene höher:** 8.2 Grundlegendes zu Zeigern **Nächste Seite:** 8.2.4 Anlegen von dynamischen

Vorige Seite: 8.2.3 Dereferenzierung von Zeigerwerten **Eine Ebene höher:** 8.2 Grundlegendes zu Zeigern **Nächste Seite:** 8.2.5 Zerstören von dynamisch

8.2.4 Anlegen von dynamischen Objekten

Zeiger können dazu verwendet werden, auf bestehende Variablen und Funktionen zu verweisen. Zeiger bringen zwar hier gewisse Vorteile, ein wesentlicher Nutzen wurde aber bisher ausgeklammert: das Speichermanagement und die damit mögliche Verwaltung von dynamischen Datenstrukturen.

Das bisher besprochene Speichermanagement zielt auf eine weitgehend automatisierte Objektverwaltung ab: Variablen werden bei der Aktivierung eines Blocks beziehungsweise vor ihrer ersten Verwendung automatisch angelegt und beim Verlassen des Blocks automatisch zerstört. In vielen Fällen ist dieses Verhalten erwünscht, da es den Programmierer von lästigen und oft fehleranfälligen Aufgaben wie Anlegen und Zerstören von Objekten entlastet.

Sehr oft aber stellt dieses Verhalten auch eine große Einschränkung für den Programmierer dar. Immer dann, wenn die Größe von Datenstrukturen frei wählbar, das heißt zur Laufzeit änderbar, sein soll und wenn der Programmierer den Zeitpunkt des Anlegens von Objekten beziehungsweise ihrer Zerstörung kontrollieren möchte, wird eine andere Art des Speichermanagements benötigt.

Für diesen Zweck kann sich ein Programmierer des Zeiger-Konzepts bedienen. Mittels Zeiger können Speicherobjekte explizit (das heißt unter Kontrolle des Programmierers) angelegt und später wieder freigegeben werden. Im Unterschied zu den bisherigen Arten von Variablen unterliegt also das Speichermanagement dem Programmierer - er ist dafür verantwortlich, daß Objekte *vor* ihrer Verwendung allokiert und auch wieder zerstört werden, wenn sie nicht mehr benötigt werden.

Um Speicherobjekte dynamisch anzulegen beziehungsweise zu zerstören, werden die Operatoren `new` und `delete` (beziehungsweise `new[]` und `delete[]`) verwendet. In der einfachsten Form von `new` werden dynamische Objekte erzeugt, indem `new` auf einen Typ-Spezifikator angewandt wird. Eine derartige Anweisung fordert vom System soviel Speicher an, wie für ein Objekt vom angegebenen Typ benötigt wird. Falls diese Anforderung erfolgreich war, gibt der `new`-Operator einen Zeiger (Adresse) auf diesen Speicherbereich zurück. Mit dem `new`-Operator werden einzelne Objekte angelegt, `new[]` wird zum Allokieren von Vektoren (siehe Abschnitt 8.3) verwendet.

Im ANSI/ISO-Standard von C++ existieren drei verschiedene Arten der Operatoren `new` (und `new[]`):

```
void* operator new(size_t) throw(bad_alloc);          (1)
void* operator new(size_t, const nothrow_t&) throw(); (2)
void* operator new(size_t, void*) throw();            (3)
```

- Die erste Variante löst im Fehlerfall (Speicher kann nicht allokiert werden) die Ausnahme

`bad_alloc` aus. Ausnahmen werden im Detail in Kapitel 15 besprochen. Vorerst kann davon ausgegangen werden, daß die Programmausführung abgebrochen wird, wenn kein Speicherplatz angefordert werden konnte.

Diese Variante des `new`-Operators ist die ,,Standard``-Variante, die immer dann verwendet wird, wenn keine spezielle Variante angegeben ist.

- Die zweite Variante des `new`-Operators liefert im Fehlerfall den Wert 0 als Rückgabewert. Dieses Verhalten *war* das Standard-Verhalten von *ARM*-C++-Compilern.
- Die dritte Variante weist einen zusätzlichen Parameter auf und wird auch als *Placement*-Variante bezeichnet. *Placement*-`new`-Operatoren werden zum Beispiel verwendet, um die Allokierung von persistenten Objekten zu realisieren und sind hier nicht weiter besprochen.

Welche Variante verwendet wird, obliegt dem Programmierer. Im folgenden sind einige Beispiele angeführt, die den Aufruf des ersten `new`-Operators bewirken:

```
int*  pInt = new int;     // Allokierung einer int-Zahl
char* pChar1 = new char;  // Allokierung eines Zeichens
char* pChar2 = new char;
```

Anmerkung: Der Ausdruck `new int` wird vom C++-System als ,,verkürzte`` Schreibweise für den Ausdruck `operator new(sizeof(T))` betrachtet.

Soll die zweite Variante des `new`-Operators verwendet werden, so muß die Bibliothek `new` inkludiert und die vordefinierte Konstante `nothrow` angegeben werden. Damit wird dem C++-System angezeigt, daß die Ausnahmen-Version des `new`-Operators *nicht* verwendet wird:

```
#include<new>

int* pInt = new(nothrow) int;
if (pInt) {                    // Allokierung ok?
  ...
}
```

Im Gegensatz zu ,,normalen`` Variablen haben die mit `new` allokierten Speicherobjekte keinen Namen und können ausschließlich über ihre Lage im Speicher (und damit über einen Zeiger) angesprochen werden:

```
*pInt = 7*y +2;
cin >> *pChar1;
*pChar2 = *pChar1;
```

Da die Speicherobjekte ausschließlich über Zeiger referenziert werden können, muß darauf geachtet werden, Objekte nicht zu ,,verlieren``. Im Beispiel unten wird der Zeiger `pChar1` `pChar2` zugewiesen, das heißt `pChar2` verweist jetzt ebenfalls auf das Speicherobjekt, auf das `pChar1` verweist.

```
pChar2 = pChar1;
```

Allerdings geht durch diese Anweisung die einzige Möglichkeit, das Speicherobjekt auf das `pChar2` vorher verwiesen hat, zu referenzieren, verloren. Es existiert kein Zeiger mehr, der auf dieses Objekt verweist und es existiert keine Möglichkeit, dieses Objekt weiter zu verwenden oder zu zerstören - das

Objekt ist verloren. Aber Achtung - auch wenn das Objekt nicht mehr verwendet werden kann, belegt es dennoch weiter Speicher!

Vorige Seite: 8.2.3 Dereferenzierung von Zeigerwerten **Eine Ebene höher:** 8.2 Grundlegendes zu Zeigern **Nächste Seite:** 8.2.5 Zerstören von dynamisch

Vorige Seite: 8.2.4 Anlegen von dynamischen **Eine Ebene höher:** 8.2 Grundlegendes zu Zeigern
Nächste Seite: 8.2.6 Zeigerkonstanten

8.2.5 Zerstören von dynamisch angelegten Speicherobjekten

Wie bereits oben angeführt, müssen die Speicherobjekte nach ihrer Verwendung explizit freigegeben werden. In C++ geschieht dies durch einen Aufruf des Operators `delete` beziehungsweise `delete[]` im Fall von Vektoren (siehe Abschnitt 8.3.1). *Jedes* allokierte Speicherobjekt muß explizit wieder zerstört werden!

```
delete pInt;     // gibt den fuer ein int-Objekt
                 // angeforderten Speicher frei
delete pChar1;   // gibt das char-Objekt frei
```

Anmerkung: Auch von *delete* existieren verschiedene Versionen, die aber hier nicht von Interesse sind. Für genauere Betrachtungen sei auf die Dokumentation des jeweiligen Entwicklungssystems beziehungsweise auf [ISO95] verwiesen.

Der `delete`-Operator gibt den Speicherbereich, auf den seine rechte Seite (ein Zeiger) verweist, frei. Es ist dabei nicht definiert, ob der Zeiger, der das zu zerstörende Speicherobjekt bezeichnet, verändert (null gesetzt) wird. `delete` kann auch mit einem Nullzeiger als Parameter aufgerufen werden. Dies führt zu keinem Fehler.

Wichtig ist in diesem Zusammenhang, daß alle mit `new` angeforderten Speicherobjekte mit `delete` freigegeben werden (und *nicht* mit `free`, der entsprechenden C-Funktion).

```
int*    pInt0;           // (1)
int*    pInt1;           // (2)
int*    pInt2 = new int; // (3)
pInt1 = new int;         // (4)
pInt2 = pInt1;           // (5)

delete pInt0;            // (6)
delete pInt2;            // (7)

*pInt1 = 10;             // (8)
delete pInt1;            // (9)
delete pInt2;            // (10)
*pInt2 = 20;             // (11)
```

Im Beispiel oben werden drei `int`-Zeiger vereinbart. In (3) und (4) werden zwei Zeiger auf dynamische Objekte angelegt. (5) ist ,,fehlerhaft``, da das Speicherobjekt, auf das `pInt2` verweist, nach der Zuweisung nicht mehr referenziert werden kann und verloren ist. (6) ruft den `delete`-Operator mit `pInt0` als Operand auf und ist damit ebenfalls fehlerhaft: `pInt0` ist nicht initialisiert und verweist auf keinen gültigen Speicherplatz, das Ergebnis der `delete`-Anweisung ist nicht definiert.

(7) zerstört das Speicherobjekt auf das `pInt2` verweist. Dieses Speicherobjekt ist identisch mit demjenigen, auf das `pInt1` zeigt. `pInt1` verweist daher jetzt auf einen ungültigen Speicherplatz - es ist ein sogenannter *Dangling Pointer*.

Der erste Fehler in diesem Zusammenhang tritt in (8) auf. `pInt1` verweist auf kein gültiges Objekt und wird dennoch dereferenziert. (9) und (10) sind ebenfalls fehlerhaft. Im Fall von (9) verweist `pInt1` auf den Speicherbereich, der bereits in (7) zerstört wurde.

Im Fall von (10) muß bemerkt werden, daß `delete` zwar problemlos auf Nullzeiger angewandt werden kann, aber nicht davon ausgegangen werden darf, daß die Anweisung `delete pInt2` die Variable `pInt2` auf null setzt. Ist die Variable aber nicht null, so wird `delete` mit einem Operand aufgerufen, der keinen gültigen Speicherplatz referenziert.

Auch (11) führt zu einem Fehler. Der Zeiger `pInt2` darf nicht mehr dereferenziert werden, da er auf kein gültiges Objekt verweist.

C++ verfügt über keine automatisierte Speicherverwaltung, in der nicht mehr benötigte dynamisch angelegte Speicherobjekte automatisch wieder freigegeben werden (*Garbage Collection*) . Das bedeutet, daß alle explizit angeforderten Objekte auch wieder explizit freigegeben werden müssen.

Vorige Seite: 8.2.4 Anlegen von dynamischen **Eine Ebene höher:** 8.2 Grundlegendes zu Zeigern
Nächste Seite: 8.2.6 Zeigerkonstanten

Vorige Seite: 8.2.5 Zerstören von dynamisch **Eine Ebene höher:** 8.2 Grundlegendes zu Zeigern
Nächste Seite: 8.2.7 Spezielle Zeigertypen

8.2.6 Zeigerkonstanten

Der Wert einer Zeigerkonstanten ist nach der Deklaration nicht mehr veränderbar, kann aber wie ein ,,normaler`` Zeiger verwendet werden. Bei der Vereinbarung von konstanten Objekten muß allerdings genau beachtet werden, was als konstant deklariert wird. So bezeichnen `const char*` und `char* const` zwei verschiedene Typen.

Der Unterschied ergibt sich aus der verschiedenen Lage des Schlüsselworts `const`: `const` wird ,,nach rechts gezogen`` und bezeichnet damit im ersten Fall einen Typ ,,Zeiger auf ein konstantes `char`-Speicherobjekt``, während im zweiten Fall der Typ ,,konstanter Zeiger auf `char`-Speicherobjekt`` vorliegt.

In Tabelle 8.1 sind die verschiedenen Möglichkeiten, die sich aus der Kombination von `const` und Zeigern ergeben[Mey92, S. 73].

Objekt	Zeiger	Bemerkung
`char*`	`p1;`	non-`const`-Zeiger auf non-`const`-Objekt
`char*`	`const p2;`	`const`-Zeiger auf non-`const`-Objekt
`const char*`	`p3;`	non-`const`-Zeiger auf `const`-Objekt
`const char*`	`const p4;`	`const`-Zeiger auf `const`-Objekt

Tabelle 8.1: `const`-Zeiger/Zeiger auf `const`-Objekte

`const`-Werte können nicht verändert werden. Ein konstanter Zeiger auf ein beliebiges Objekt bedeutet also, daß der Zeigerwert nicht verändert werden kann, der Wert des Objekts, auf das der Zeiger verweist, hingegen schon. Umgekehrt kann der Wert eines Zeigers auf ein konstantes Objekt verändert werden, allerdings nicht das Objekt, auf das dieser Zeiger verweist. Im Falle eines konstanten Zeigers auf ein konstantes Objekt können weder der Zeigerwert noch das Objekt, auf das der Zeiger verweist, verändert werden. Im folgenden sind Beispiele für die richtige beziehungsweise falsche Verwendung der in Tabelle 8.1 vereinbarten Variablen:

```
p1 = "Miriam";  // Zuweisung an non-const-Zeiger, ok
p2 = "Rimbert;  // Zuweisung an const-Zeiger, Fehler
p3 = p1+1;      // Zuweisung an non-const-Zeiger, ok
++p4;           // Veraenderung eines const-Zeigers,
                // Fehler
p1[0] = "A";    // non-const-Objekt, ok
```

```
*(p2+3) = 0x00; // non-const-Objekt, ok
p3[2] = p4[1];  // const-Objekt, Fehler
p4[1] = p1[0];  // const-Objekt, Fehler
```

Vorige Seite: 8.2.5 Zerstören von dynamisch **Eine Ebene höher:** 8.2 Grundlegendes zu Zeigern
Nächste Seite: 8.2.7 Spezielle Zeigertypen

Vorige Seite: 8.2.6 Zeigerkonstanten **Eine Ebene höher:** 8.2 Grundlegendes zu Zeigern

Teilabschnitte

8.2.7 Spezielle Zeigertypen

Zwei ,,spezielle`` Zeigertypen spielen vor allem in C eine große Rolle, `void`-Zeiger und Zeiger auf Funktionen.

`void`-Zeiger

`void`-Zeiger nehmen eine besondere Stellung unter den verschiedenen Zeigern auf Objekte ein, da `void`-Objekte keine Objekte im herkömmlichen Sinn sind - der Typname `void` zeigt nur das Vorhandensein keines speziellen Typs an. `void`-Zeiger sind Objekte, die eine gültige Adresse darstellen, aber keinen bestimmten Datentyp besitzen. Sie können daher nicht einfach dereferenziert werden. Ein Beispiel für eine `void`-Zeigerdeklaration:

```
int      i = 17;
void*    anyPtr = &i;
```

`anyPtr` ist ein `void`-Zeiger, dem die Adresse des `int`-Objekts `i` zugewiesen wird. Da `anyPtr` ein `void`-Zeiger ist, kann keine einfache Dereferenzierung erfolgen. Die einzige Möglichkeit, den Zeiger zu dereferenzieren, besteht darin, ihn vorher einer Typumwandlung zu unterziehen. Im folgenden Beispiel wird der `void*`-Zeiger auf einen Zeiger vom Typ `int*` konvertiert (der dann dereferenziert werden kann):

```
cout << *(static_cast<int*>anyPtr);
```

Anmerkung: Typumwandlungen dieser Art werden im Detail in Abschnitt 9.4.2 besprochen. Vorerst genügt es zu wissen, daß der Operator `static_cast` den dahinter angeführten Ausdruck auf den in den spitzen Klammern eingeschlossenen Typ umwandelt.

`void`-Zeiger können als Funktionsargumente übergeben, zugewiesen und verglichen werden. In C wurden sie unter anderem dazu verwendet, Funktionen zu implementieren, die mit verschiedenen Arten von Daten arbeiten können.

C++ bietet neben den `void`-Zeigern andere (und bessere) Möglichkeiten, um generische Datenstrukturen zu implementieren (siehe Kapitel 12). Deswegen und weil bei der Verwendung von `void`-Zeigern Typprüfungen durch den Compiler weitgehend unmöglich sind, sollten diese nur sehr selten und mit Vorsicht verwendet werden.

Zeiger auf Funktionen

Zeiger auf Funktionen sind in C/C++ Zeiger, die die Adresse einer Funktion enthalten. Folgende Operationen sind mit derartigen Zeigern sinnvoll:

- Einem Zeiger auf eine Funktion kann eine Adresse zugewiesen werden,
- er kann als Funktionsargument übergeben werden,
- der Zeiger kann mit einem anderen Zeiger verglichen werden, und
- die Funktion, auf die er verweist, kann aufgerufen werden.

Die Deklaration enthält Angaben über die Parameterliste und den Rückgabewert einer Funktion. Um einen Zeiger auf eine Funktion zu vereinbaren, wird (wie bei allen Zeigern) der `*`-Begrenzer verwendet. Allerdings muß die Sequenz von `*` und dem Variablennamen explizit geklammert werden. Der Grund dafür ist, daß die Parameterklammern stärker binden als `*` und eine Deklaration ohne Klammern damit die Vereinbarung eines Funktionsprototyps darstellt.

```
void err1(char*);    // Prototyp
void *err2(char*);   // Prototyp, void*-Rueckgabewert
void (*fErr)(char*); // Zeiger auf Funktion mit einem
                     // char*-Parameter
```

Die Adresse einer Funktion ist durch ihren Namen bestimmt. Eine Adreßbestimmung über den &-Operator ist nicht nötig.

```
fErr = err;  // Initialisiere fErr mit Adresse von err
fErr = &err; // gleich wie oben, & ist ueberfluessig
```

Um die Funktion `err` über einen Zeiger (zum Beispiel `fErr`) aufzurufen, muß der Indirektionsoperator `*` nicht verwendet werden. Wird er verwendet, so ist zu beachten, daß der Operator für den Funktionsaufruf `()` eine höhere Priorität als `*` besitzt. Daher führt `*fErr("fehler")` zu einem Fehler, weil dies vom Compiler als `*(fErr("fehler"))` interpretiert wird.

```
(*fErr)("hallo!");  // Aufruf ueber Zeiger
fErr("hallo!");     // Wie oben, nur ohne explizite
                    // Dereferenzierung
```

Der Typ eines Zeigers auf eine Funktion umfaßt auch die Argumenttypen, die wie in Funktionen deklariert werden müssen. Bei der Zuweisung von Zeigern müssen die Funktionstypen vollständig übereinstimmen.

Ein Beispiel dazu:

```
void (*pf)(char*);   // Zeiger auf void(char*)

  // Prototypen f1, f2, f3
void f1(char*);  // void(char*);
int  f2(char*);  // int(char*);
void f3(int*);   // void(int*);
void f()
{
   pf = &f1;     // ok
   pf = f1;      // ok
   pf = &f2;     // Fehler: falscher Typ fuer Funktionswert
   pf = f3;      // Fehler: falscher Typ fuer Argument
   pf("asdf");   // ok

   (*pf)(1);              // Fehler: falscher Typ fuer Argument
   int i = (*pf)("qwer"); // Fehler: Zuweisung von void an int
}
```

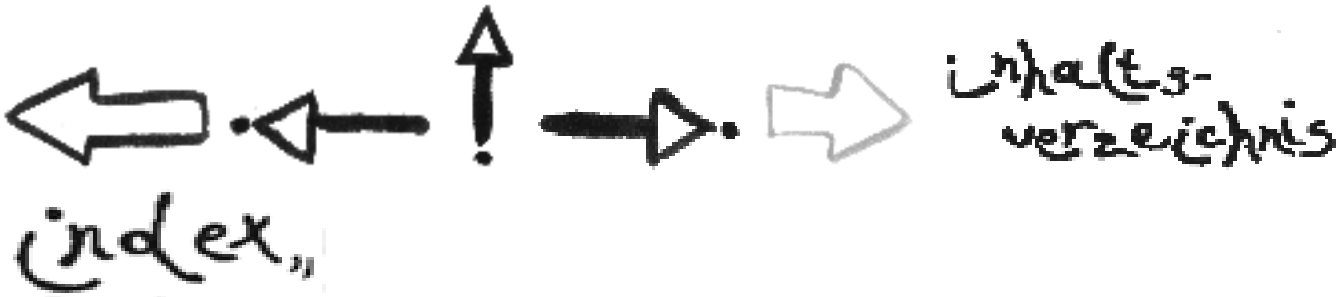
Ein Beispiel zu Zeigern auf Funktionen:

```
void (*fctPointer)(float);

void printStuff(float dataToIgnore)
{
  cout << "This is the print stuff function.\n";
}

void printMessage(float listThisData)
{
  cout << "The data to be listed is " << listThisData << "\n";
}

void printFloat(float dataToPrint)
{
  cout << "The data to be printed is " << dataToPrint << "\n";
}

main()
{
  float pi     = 3.14159;
  float two_pi = 2.0 * pi;
  printStuff(pi);
  fctPointer = printStuff;
  fctPointer(pi);
  fctPointer = printMessage;
  fctPointer(two_pi);
```

```
  fctPointer(13.0);
  fctPointer = printFloat;
  fctPointer(pi);
  printFloat(pi);
}
```

Das oben angeführte Programm liefert folgende Ausgabe:

```
This is the print stuff function.
This is the print stuff function.
The data to be listed is 6.283180
The data to be listed is 13.000000
The data to be printed is 3.141590
The data to be printed is 3.141590
```



Vorige Seite: 8.2.6 Zeigerkonstanten **Eine Ebene höher:** 8.2 Grundlegendes zu Zeigern

Vorige Seite: Zeiger auf Funktionen **Eine Ebene höher:** 8 Höhere Datentypen und **Nächste Seite:** 8.3.1 Vektoren und Zeiger

8.3 Vektoren

Vektoren sind Datentypen, die es erlauben, mehrere gleichartige Objekte zu einer größeren Einheit zusammenzufassen. Die einzelnen Objekte werden sequentiell (also ,,direkt hintereinander``) im Speicher abgelegt. Ein Vektor von 10 `int`-Objekten ist damit zehnmal so groß wie ein einzelnes `int`-Objekt.

Ein Vektor mit *n* Elementen wird durch die Angabe der Dimensionen in eckigen Klammern hinter dem Variablennamen angegeben.

```
int     simpleVektor[9];  // 9 ints
float   x[2];             // 2 floats
double* ptr[256];         // 256 Zeiger auf double
```

Vektoren können auch mehr als eine Dimension aufweisen. Um einen zweidimensionalen Vektor zu deklarieren, wird die zweite Dimension hinter der ersten angeführt. Ein zweidimensionaler `int`-Vektor mit den Dimensionen `x`, `y` ist ein Vektor von `x` Vektoren mit jeweils `y` `int`-Elementen.

```
int   matrix[9][10];  // Vektor mit 9x10 ints
float x[2][3];        // zweidimensionaler Vektor
                      // mit 2x3 Elementen
float y[4][5][6];     // dreidimensionaler Vektor
                      // mit 4*5*6=120 Elementen
```

Neben der Deklaration von Vektoren stellt sich die Frage nach den zulässigen Operationen für Vektoren. Vektoren können *nicht* als Ganzes zugewiesen oder verglichen werden, sie können aber als Zeiger aufgefaßt (siehe unten) und indiziert werden. Einen Vektor zu indizieren bedeutet, auf sein `i`. Element zuzugreifen. Dies erfolgt über den Indexoperator `[]` . Das erste Element eines Vektors hat den Index 0, das zweite 1, das *x*. Element den Index *x-1*.

```
simpleVector[0] = 12;     // 1. Element zuweisen
simpleVector[1] = 23;     // 2. Element zuweisen
cout << simpleVector[0];  // Ausgabe des 1. Elements
```

Bei der Indizierung mehrdimensionaler Vektoren ist die mehrfache Verwendung der eckigen Klammern notwendig:

```
matrix[0][0] = 1;  // 1. Element der 1. Zeile v. Matrix
matrix[1][2] = 3;  // 3. Element der 2. Zeile v. matrix
```

Zu beachten ist, daß C++ im Gegensatz zu anderen Programmiersprachen *keinerlei* Indexprüfungen vornimmt. Es liegt also in der Verantwortung des Programmierers, bei einem Vektor mit *x* Elementen nur diese *x* Elemente zu referenzieren. Das Verhalten von Programmen, die nicht allokierte Elemente von Vektoren referenzieren, ist nicht vorhersehbar!

Nachfolgendes Beispiel zeigt die elementweise Matrizenaddition einer *N***N*-Matrix `a` mit einer *N***N*-Matrix `b`:

```
int a[N][N];  // zweidimensionale Matrix a
int b[N][N];  // zweidimensionale Matrix b
int c[N][N];  // Ergebnis in Matrix c

for (int line = 0; line<N; ++line) {
  for (int col = 0; col<N; ++col) {
    c[line][col] = a[line][col]+b[line][col];
  }
}
```

Vorige Seite: Zeiger auf Funktionen **Eine Ebene höher:** 8 Höhere Datentypen und **Nächste Seite:** 8.3.1 Vektoren und Zeiger

Vorige Seite: 8.3 Vektoren **Eine Ebene höher:** 8.3 Vektoren **Nächste Seite:** 8.3.2 Zeigerarithmetik

8.3.1 Vektoren und Zeiger

Zwischen Zeigern und Vektoren besteht in C++ ein enger Zusammenhang. Vektoren sind in C++ bestimmt durch einen konstanten (= nicht veränderbaren) Zeiger auf das erste Element. Da die einzelnen (Teil-) Objekte eines Vektors sequentiell abgelegt sind, ergibt sich die Speicheradresse des `i`. Elements durch die Addition des Offsets `i` zur Basisadresse (= Adresse des 1. Elements) (= `i`* Größe eines Elements).

Um die Adresse des ersten Elements festzustellen, genügt es, den Namen einer Vektor-Variablen zu verwenden. Ähnlich wie bei Adressen von Funktionen (vergleiche Abschnitt 8.2.7) kann der Adreßoperator & verwendet werden, er ist aber in diesem Fall überflüssig.

```
int  intArr[128];  // int-Vektor mit 128 Elementen
int* pI = intArr;  // pI zeigt auf 1. Element v. intArr
pI = &intArr;      // wie oben, & ist unnoetig
```

Wesentlich ist der Unterschied zwischen Zeigern und Vektoren: Vektoren werden durch einen Zeiger auf das erste Element repräsentiert, ein Vektor ist aber mehr als ein Zeiger. Jeder Vektor mit *x* Elementen reserviert Speicher für diese. Ein Zeiger hingegen stellt nur einen Verweis dar, er reserviert keinen Speicher!

Vektoren werden durch konstante Zeiger auf ihr erstes Element repräsentiert. Vektoren und Zeiger unterscheiden sich allerdings darin, daß bei der Deklaration eines Vektors Speicherplatz für alle seine Elemente allokiert wird, während ein Zeiger einen einfachen Verweis darstellt und zu keiner Speicherplatzallokierung führt.

Vektoren können auch dynamisch allokiert werden. Dazu wird der `new[]`-Operator verwendet. Auch von diesem Operator existieren (wie von `new`) drei verschiedene Versionen, deren Verhalten analog den `new`-Operatoren ist.

```
void* operator new[](size_t) throw(bad_alloc);
void* operator new[](size_t, const nothrow_t&) throw();
void* operator new[](size_t, void*) throw();
char* pChar1 = new char[120]; // Allokierung eines
```

```
                                    // Vektors (120 Zeichen)
```

Der Rückgabewert von `new[]` ist (wie bei `new`) die Adresse des neuen Objekts beziehungsweise 0. Aufgrund des Zusammenhangs zwischen Zeigern und Vektoren können die einzelnen Elemente von `pChar1` wie gewohnt indiziert werden:

```
pChar1[10] = 'A';   // 11. Element, ueber Indexoperator
```

Auch mehrdimensionale Vektoren können so angelegt werden. Allerdings ist zu beachten, daß ein Zeiger auf einen zweidimensionalen Vektor vom Typ „Zeiger auf Vektor`` ist.

Da ein Vektor als Zeiger angesprochen wird, werden zweidimensionale Vektoren damit als „Zeiger auf Zeiger`` angesprochen:

```
int   f[12][12];          // zweidimensionaler Vektor
int*  p1;                 // Zeiger
int** p2;                 // Zeiger auf Zeiger
int** p3 = new int[3][3]; // Zweidimensionaler Vektor,
                          // dynamisch allokiert

p1 = f;  // Fehler! p1 darf nur auf eindimensionale
         // Vektoren verweisen
p2 = f;  // OK, p ist Zeiger auf Zeiger
```

Speicherobjekte, die mit dem Operator `new[]` angelegt wurden, müssen mit dem Operator `delete[]` zerstört werden. Der Operator `delete` zerstört nur *ein einzelnes* Element des Vektors (das erste)!

```
delete p3;     // Falsch! Loescht nur erstes Element
delete[] p3;   // OK, loescht gesamten Vektor
```

- Speicherobjekte, die mit `new[]` angelegt wurden, müssen mit dem `delete[]`-Operator zerstört werden. Objekte, die mit dem `new`-Operator angelegt werden, müssen mit dem `delete`-Operator zerstört werden.

Vorige Seite: 8.3 Vektoren **Eine Ebene höher:** 8.3 Vektoren **Nächste Seite:** 8.3.2 Zeigerarithmetik

Vorige Seite: 8.3.1 Vektoren und Zeiger **Eine Ebene höher:** 8.3 Vektoren **Nächste Seite:** 8.3.3 Vektoren als Parameter

8.3.2 Zeigerarithmetik

Im Zusammenhang mit Vektoren soll eine wesentliche Eigenschaft von C++-Zeigern erörtert werden, die sogenannte Zeigerarithmetik: Zeiger auf Objekte können um `int`-Werte erhöht und erniedrigt werden, andere Zeiger können addiert und subtrahiert werden und Zeiger können verglichen werden.

Die Addition von `int`-Werten zu Zeigern ist dabei so definiert, daß das Ergebnis der Addition von `x` zu einem Vektor `p` definiert ist als die Addition von `p` und `x` mal der Größe des Typs auf den der Zeiger `p` verweist (*pAnyType* + *x* bedeutet *pAnyType* + `sizeof(`*AnyType*`)`**x*). Von Bedeutung ist dies besonders im Zusammenhang mit Vektoren:

Da die Addition von `i` zu einem Zeiger der Erhöhung um `i` mal der Größe des Vektor-Elements entspricht, sind die Ausdrücke `*(p+i)` und `p[i]` semantisch identisch.

```
int* pI;           // Zeiger auf int
int  intArr[128];  // int-Vektor mit 128 Elementen
int  f[12][12];    // zweidimensionaler Vektor

pI = intArr;   // pI zeigt auf das erste Element
++pI;          // pI wird um den Wert size(int) erhoeht,
               // zeigt also auf das zweite Element
*pI = 2;       // entspricht intArr[1] = 2;
pI += 3;       // pI zeigt auf intArr[4]

if (f[3][4]==*(f+3*12+4)) { ... // immer wahr
```

Tatsächlich stellt der Indexoperator `[]` nichts anderes als eine etwas andere Schreibweise für die Adressierung mittels Zeigerarithmetik dar:

```
char   str[100];
...
str[3] = 'A';        // ist gleichbedeutend mit
*(str+3) = 'A';
```

Im ersten Fall wird auf das vierte Element des Vektors `str` über den Index zugegriffen. Im zweiten Fall erfolgt der Zugriff durch Addition der Zahl 3 zur Adresse des ersten Elements des Vektors (`str`). Der dabei entstehende Zeigerwert wird durch den Indirektionsoperator dereferenziert.

Die Subtraktion von Zeigern ist nur dann definiert, wenn beide Zeiger auf eine Adresse innerhalb

desselben Vektors verweisen. Dann liefert die Subtraktion die Anzahl der Vektor-Elemente zwischen den beiden Zeigern:

```
int   f[100];
int   g[20]
char* p1;
char* p2;
...
p1 = f;
p2 = f+3;
cout << p2-p1;  // ok, p1 & p2 zeigen auf Elemente v. f

p2 = g;
cout << p2-p1;  // Fehler!
```

Ebenso wie die Subtraktion ist auch der Vergleich von zwei Zeigerwerten nur dann sinnvoll, wenn beide Zeiger auf Elemente im selben Vektor verweisen.

Folgendes Beispiel zeigt noch einmal den Unterschied zwischen Zeigern und Vektoren:

```
char  str[100];
char* p;          // Deklaration char-Zeiger
...
p[3] = 'A';       // Fehler! p verweist auf keinen
                  // gueltigen Speicherplatz!
*(str+3) = 'A';   // ok, fuer str wurde automatisch
                  // Speicher allokiert
```

Häufig kommt der Indirektionsoperator * in Ausdrücken zusammen mit Inkrement- und Dekrement-Operatoren vor. Die Inkrementierung kann sich entweder auf den Speicherplatzinhalt oder den Zeigerwert beziehen. Folgende Anweisungen sollen das Zusammenspiel der Operatoren ++ (beziehungsweise --) und * verdeutlichen.

```
int     f[N];
int*    z1;
int*    z2;
int*    z3;
int*    z4;
int     x, i;
...
i = 0;
z1 = &f[i];
z2 = z1;
...           // Inkrementieren des Vektor-Index
              // (Zeigerarithmetik)
x = *z1++;    // analog zu: x = *z1; z1++;

x = *(++z2);  // analog zu ++z2; x = *z2;
```

```
                // Inkrementieren des referenzierten
                // Vektor-Elements
z3 = f;         // z3 verweist auf das erste Element
z4 = &f;        // ebenso z4, "&" ist redundant!
x = (*z3)++;    // analog zu x = f[0]++;
x = ++*z4;      // analog zu x = ++(*z4); beziehungsweise
                // zu x = ++f[0];
```

Um den Programmcode lesbarer zu gestalten, sollte bei Vektoren der Indirektionsoperator * sparsam eingesetzt werden. Wann immer dies möglich ist, sollte der dafür vorgesehene Indexoperator [] verwendet werden. Die Verwendung von Konstrukten wie *++, ++* sollte im Sinne der Lesbarkeit und Klarheit überhaupt unterbleiben. Sie bringen keinen Vorteil.

Vorige Seite: 8.3.1 Vektoren und Zeiger **Eine Ebene höher:** 8.3 Vektoren **Nächste Seite:** 8.3.3 Vektoren als Parameter

Vorige Seite: 8.3.2 Zeigerarithmetik **Eine Ebene höher:** 8.3 Vektoren

8.3.3 Vektoren als Parameter

Um einen Parameter vom Typ „`char`-Vektor`` zu vereinbaren, genügt es, den Parametertyp mit `char*` oder `char[]` festzulegen. Es ist keine explizite Größenangabe bezüglich des Vektors nötig. Sie wäre auch nicht sinnvoll, da in C++ ohnehin keine Indexprüfungen bei Vektoren erfolgen.

Die Typen `char[]` und `char*` sind im wesentlichen identisch und bezeichnen beide einen Zeiger auf `char`. Allerdings geht aus `char[]` klar hervor, daß es sich hier um einen `char`-Vektor handelt, während `char*` auch einen Zeiger auf ein einzelnes Zeichen darstellen kann. Im Fall von Vektoren empfiehlt sich daher die Verwendung von `[]` anstelle von `*`.

```
void foo1(char* p);  // Zeiger auf char oder char-Vektor
void foo2(char p);   // char-Vektor
```

Zu beachten ist, daß bestimmte Typen (Vektoren) immer als Zeiger übergeben werden. Vektoren werden in C/C++ durch einen konstanten Zeiger auf das erste Element repräsentiert. Bei der Übergabe eines Vektor-Parameters wird daher keine Kopie des gesamten Vektors angelegt, sondern nur eine Kopie des Zeigers auf das erste Element.

Ein Beispiel für eine Funktion mit einem Vektor-Parameter:

```
  // "sort" sortiert einen Vektor von ints. Da die
  // Laenge des Vektors nicht bekannt ist, wird ein
  // zusaetzlicher Parameter verwendet.
void sort(int intArr[], int size)
{
  int min, help;

  for (int i=0; i<size-1; ++i) {
    min = i;
    for (int j=i+1; j<size; ++j)
      if (intArr[j]<intArr[min]) min = j;

    if (min != i) {
      help = intArr[min];
```

```
      intArr[min] = intArr[i];
      intArr[i] = help;
    }
  }
}
```

- Bei der Übergabe von Vektoren als Parameter an Funktionen werden keine Kopien der Vektoren, sondern Kopien der Zeiger auf das erste Element angelegt.

Vorige Seite: 8.3.2 Zeigerarithmetik **Eine Ebene höher:** 8.3 Vektoren

Vorige Seite: 8.3.3 Vektoren als Parameter **Eine Ebene höher:** 8 Höhere Datentypen und **Nächste Seite:** 8.5 Parameterübergabe mit Zeigern

---

8.4 Zeichenketten

Zeichenketten werden in C++ durch Vektoren vom Basistyp `char` dargestellt, die durch das Zeichen mit dem ASCII-Wert 0 terminiert werden.

```
char* p;                  // Deklaration eines Zeigers
                          // (kein Speicherplatz fuer die
                          // Zeichenkette reserviert)
char  myString1[N];       // Zeichenkette der Laenge N
char* p2 = "Hallo!";      // Zeiger auf eine Zeichenkette,
                          // initialisiert mit Adresse von
                          // "Hallo!" (automatisch vom
                          // System angelegt)
```

Im obigen Beispiel werden die Variablen `p`, `myString1` und `p2` vereinbart. `myString1` ist ein Vektor mit der Länge `N`, daher werden auch *N* mal die Elementgröße Bytes an Speicher reserviert (in diesem Fall *N**1). Der Name `myString1` ist damit ein konstanter (= nicht veränderbarer) Zeiger auf das erste Element des Vektors `myString1`.

Im Gegensatz dazu wird für `p` nur der Speicherplatz für die eigentliche Zeigervariable reserviert, da `p` nur ein Zeiger und kein Vektor ist. Allerdings kann `p` auf Objekte vom Typ `char` verweisen, also auch auf eine Reihe von Zeichen (Zeichenketten). `p2` stellt ebenfalls einen Zeiger dar, für den nur der Speicherplatz für die eigentliche Zeigervariable angelegt wird. Die Zeichenkette `Hallo!` ist ein globales Zeichenketten-Literal und wird vom C++-System abgelegt. Der Zeiger `p2` wird mit der Adresse dieser konstanten Zeichenkette initialisiert und verweist damit auf eine Folge von 7 Zeichen: `H, a, l, l, o, !` und das Stringende-Zeichen (ASCII-0).

Beispiele für Zuweisungen mit den oben angeführten Variablen:

```
p = myString1;  // Richtig, p verweist jetzt auf
                // das erste Element von myString1
myString1 = p2; // Falsch, myString1 ist ein Vektor,
                // der Name "myString1" deshalb ein
                // konstanter Zeiger!
*p2 = 'h';      // Das 1. Element des Strings wird ver-
                // aendert, p2 zeigt jetzt auf "hallo!"
```

Der Wert von `p` und `p2` kann verändert werden. Im Fall von `p2` geht bei einer Veränderung des

Zeigerwerts allerdings das Literal `Hallo!` verloren, da kein Zeiger mehr darauf verweist. `myString1` kann nicht verändert werden, da Vektoren als *konstante* Zeiger auf das erste Element realisiert sind.

Im folgenden sind einige Beispiele für „typische`` Fehler bei der Verarbeitung von Zeichenketten angeführt:

```
char  s1[N];  // Zeichenkette der Laenge N-1
char  s2[N];  // Zeichenkette der Laenge N-1
char* p;
memcpy(s1, s2, N);   // ok, kopiert N Bytes von s2
                     // nach s1
memcpy(s1, s2, 2*N); // Fehler, kopiert 2*N Bytes von
                     // s2 nach s1, obwohl s1 und s2
                     // nur aus N Elementen bestehen
memcpy(p, s1, N);    // Fehler, p ist ein Zeiger - es
                     // wurde aber kein Speicher
                     // allokiert
```

Anmerkung: `memcpy(dest, source, len)` ist eine Funktion, die `len` Bytes von `source` nach `dest` kopiert.

In der Anweisung `memcpy(s1, s2, 2*N);` wird eine Vektorgrenze überschrieben: `s1` ist ein Vektor mit *N* Elementen und durch die `memcpy`-Anweisung werden `2*N` Elemente kopiert. In der Anweisung `memcpy(p, s1, N);` erfolgt ebenfalls ein unerlaubter Speicherzugriff: `p` ist ein Zeiger und kein Vektor, daher müßte in diesem Fall explizit Speicher angefordert werden oder `p` entsprechend initialisiert werden. Beide oben angeführten Fehler werden nicht vom Compiler entdeckt. Zur Laufzeit ist das Programmverhalten undefiniert. Eventuell terminiert das Betriebssystem das Programm, weil nicht angeforderter Speicher referenziert wird. Ebenso können aber Codeteile oder der Inhalt von Variablen überschrieben werden. Das weitere Programmverhalten ist damit undefiniert.

In C++ wird jede Zeichenkette durch das Zeichen mit dem ASCII-Wert 0 abgeschlossen. Werden Zeichenketten-Literale vereinbart, so fügt das C++-System automatisch `0` am Ende an. Die korrekte Terminierung von Zeichenketten ist auch dann gewährleistet, wenn die von jeder Standard-C++-Bibliothek bereitgestellten Zeichenketten-Verarbeitungsfunktionen (`strcmp`, `strcpy`, `strncpy` etc.) zur Verarbeitung verwendet werden. Werden Zeichenketten aber zeichenweise (oder mit Operationen wie memcpy) verarbeitet, so hat der Programmierer dafür Sorge zu tragen, daß die Zeichenketten entsprechend terminiert sind. Im folgenden Beispiel wird die Zeichenkette in einer `while`-Schleife so lange durchlaufen, bis das Terminierungszeichen erreicht ist (Feststellen der Zeichenketten-Länge):

```
int length(char string[])
{
  int len = 0;
  while (string[len] != 0x00) { // Stringende erreicht?
    ++len;
  }
  return len;
```

```
}
```

Eine textuell kürzere (allerdings nicht unbedingt besser lesbare) Variante der Funktion `length`:

```
int length(char* string)
{
  int len = 0;
  while (*string++) len++;
  return len;
}
```

Eine andere Version zur Berechnung der Stringlänge sucht zuerst die Endadresse der Zeichenkette und subtrahiert diese dann von der Anfangsadresse:

```
int length(char string[])
{
  char* pHelp = string;
  while (*pHelp++);
  return pHelp-string+1;
}
```

Achtung: Die beiden zuletzt angeführten Versionen sind zwar textuell kürzer, aber zweifellos wesentlich schwerer lesbar. Der ohnedies geringe Vorteil des *Performance*-Gewinns durch die Erstellung derartiger „Konstrukte`` wird durch den Nachteil der schwereren Wartbarkeit zumindest aufgehoben.

Vorige Seite: 8.3.3 Vektoren als Parameter **Eine Ebene höher:** 8 Höhere Datentypen und **Nächste Seite:** 8.5 Parameterübergabe mit Zeigern

Vorige Seite: 8.4 Zeichenketten **Eine Ebene höher:** 8 Höhere Datentypen und **Nächste Seite:** 8.5.1 *Call by Value* und *Call by Reference*

8.5 Parameterübergabe mit Zeigern und Referenzen

Wie bereits in Kapitel 7 dargestellt, müssen Funktionen in der Lage sein, mit ihrer Umgebung Daten auszutauschen. Dies kann einerseits über globale Daten erfolgen, andererseits über die in der Parameterliste angeführten Variablen, die Parameter. Der Datenaustausch mittels globaler Variablen ist in höchstem Maße gefährlich: Die Kopplung (= die ,,logische`` Bindung) zwischen Programmteilen ist sehr hoch, die Änderung von globalen Daten führt oft zu nur sehr schwer auffindbaren Fehlern, die Schnittstelle der Funktion ist nirgends erkennbar etc. Globale Variablen sollten daher nur in wirklich begründeten Fällen verwendet werden. In allen anderen Fällen ist der Datenaustausch mittels Parameter vorzuziehen.

Wie bereits in Abschnitt 7.3 erwähnt, werden die in der Schnittstelle angeführten Parameter als formale Parameter bezeichnet. Die formalen Parameter sind ,,Platzhalter`` für die eigentlichen, aktuellen Parameter. Beim Funktionsaufruf ersetzen dann die angegebenen aktuellen Parameter die formalen. In der Informatik unterscheidet man grundsätzlich zwischen zwei Arten von Parameterübergabe: *Call by Reference* und *Call by Value*.

Bei der Parameterübergabe mittels *Call by Value* wird eine Kopie des aktuellen Parameters übergeben, während bei *Call by Reference* eine Referenz auf den aktuellen Parameter (also gleichsam die Variable selbst, keine Kopie) verwendet wird (vergleiche VAL/VAR-Parameter in Modula-2/Pascal). Bei der *Call by Value*-Parameterübergabe bleibt jede Änderung eines Parameters innerhalb einer Funktion für den Aufrufer bedeutungslos, weil ja nur eine Kopie des aktuellen Parameters verändert wird.

Vorige Seite: 8.4 Zeichenketten **Eine Ebene höher:** 8 Höhere Datentypen und **Nächste Seite:** 8.5.1 *Call by Value* und *Call by Reference*

Vorige Seite: 8.5 Parameterübergabe mit Zeigern **Eine Ebene höher:** 8.5 Parameterübergabe mit Zeigern **Nächste Seite:** 8.5.2 Gegenüberstellung der zwei

8.5.1 *Call by Value* und *Call by Reference*

C kennt nur die *Call by Value*-Parameterübergabe. In der folgenden Funktion `exchange`

```
void exchange(int a, int b)      // Call by Value
{
  int h;

  h = a;
  a = b;
  b = h;
}
```

hat daher der Aufruf

```
exchange( myVal, yourVal);
```

keine Auswirkungen, da innerhalb der Funktion nur Kopien der Parameter `myVal` und `yourVal` verwendet werden.

Eine Möglichkeit, VAR-Parameter (Stichwort *Call by Reference*) in C++ zu simulieren, besteht darin, nicht die Objekte selbst zu übergeben, sondern ihre Adressen. In der Funktion kann nun über die Dereferenzierung des übergebenen Zeigers die eigentliche Variable verändert werden. (Änderungen des übergebenen Zeigers selbst bleiben ohne Bedeutung, da ja nur eine Kopie übergeben wurde.)

Die folgende Funktion `exchange1` verwendet Zeigerparameter zur ,,Simulation`` von *Call by Reference*:

```
void exchange1(int* pA, int* pB)
{
  int h;
  h = *pA; *pA = *pB; *pB = h;
}
```

Beim Aufruf der Funktion `exchange1` müssen natürlich die Adressen der Variablen und nicht die Variablen selbst übergeben werden:

```
exchange1(&myVal, &yourVal);
```

In C++ besteht aber im Gegensatz zu C die Möglichkeit, die *Call by Reference*-Parameterübergabe

nachzubilden. `exchange2` zeigt das Vertauschen zweier `int`-Variablen mittels Referenzparameter:

```
void exchange2(int& a, int& b)
{
  int h;
  h = a; a = b; b = h;
}
```

Beim Aufruf von `exchange2` werden die Variablen selbst übergeben und nicht deren Adressen:

```
exchange2(myVal, yourVal);
```

Der C++-Compiler übergibt hier lediglich die ,,Referenz-Objekte``.

Vorige Seite: 8.5.1 *Call by Value* und *Call by Reference* **Eine Ebene höher:** 8.5 Parameterübergabe mit Zeigern

8.5.2 Gegenüberstellung der zwei Arten der Parameterübergabe

Die *Call by Value*-Parameterübergabe hat zwei wesentliche Nachteile:

- Zum einen können *VAR-Parameter* nur mehr über den Umweg der Adressen von Objekten simuliert werden.
- Zum anderen muß bei jedem Funktionsaufruf eine Kopie der aktuellen Parameter erzeugt werden. Bei einfachen Datentypen stellt das kein Problem dar. Da C++ aber eine objektorientierte Programmiersprache ist, werden sehr oft komplexe Objekte, die große Datenmengen umfassen können, als Parameter verwendet. Jeder Funktionsaufruf bedeutet, daß eine Kopie eines Objekts angelegt und diese nach Verlassen der Funktion wieder zerstört werden muß. Das bringt einen erheblichen Effizienzverlust mit sich.

Die *Call by Reference*-Parameterübergabe stellt eine bessere Lösung dar: Die aktuellen Parameter können innerhalb einer Funktion verändert werden, da bei der Übergabe keine Kopie angelegt wird. So entstehen auch keine unnötigen Performance-Probleme.

Der Nachteil, daß jeder Parameter (also auch Eingangsparameter) als Referenz übergeben wird, kann durch die Verwendung von `const` ausgeglichen werden: Eingangsparameter werden als Referenzen konstanter Objekte vereinbart. Damit können sie innerhalb einer Funktion nicht (versehentlich) verändert werden. (Eine absichtliche Veränderung ist auch hier möglich)

Ein weiterer Vorteil der Parameterübergabe mit Referenzen besteht darin, daß das sogenannte *Slicing*-Problem (Problem im Zusammenhang mit Klassen und Objekten, siehe Kapitel 14) vermieden wird.

Daher sollten Objekte in Hinblick auf die objektorientierte Programmierung als Referenzen beziehungsweise als konstante Referenzen oder Zeiger übergeben werden. Vor allem im Zusammenhang mit komplexen Objekten ist dies von Bedeutung.

Allerdings bringt es keine Performance-Vorteile, einfache Objekte (wie etwa `int`, `bool` etc.) als

Referenzen zu übergeben. Derartige Eingangsparameter werden üblicherweise *Call by Value* übergeben.

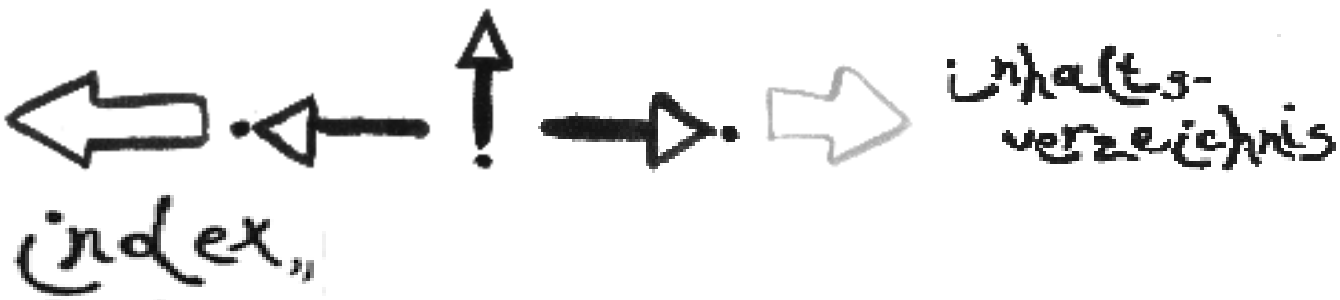
Beispiele:

```
  // Zwei Eingangsparameter, einfache Objekte
  // wahlweise mit Referenzen oder ohne
int ggt(const int& a, const int& b);
int ggt(const int a, const int b);

  // Komplexe Eingangsparameter (const &)
String getWinName(const Window& theWin);

  // Uebergangsparameter immer mit Referenzen:
void invertInteger(int& theNumber);

  // Zwei komplexe Eingangsobjekte, ein
  // Ausgangsparameter
void anyFoo(const ComplexObject& op1,
            const ComplexObject& op2,
            ComplexObject& theResult);
```

Tabelle 8.2 zeigt die Möglichkeiten der Parameterübergabe. In dieser Tabelle werden die Parameter nach ihrer Komplexität in einfache und komplexe Objekte unterschieden. Einfache Objekte sind Variablen unstrukturierter, vordefinierter Datentypen (`char`, `int`, `short`, `long`, Zeiger...). Alle anderen werden als komplexe Objekte bezeichnet.

Tabelle 8.2: Möglichkeiten der Parameterübergabe

Parameterart	Eingangsparameter	Ausgangsparameter
einfache Objekte	Einfache ,,normale`` Parameter,	Referenzen
	konstante Referenzen	
komplexe Objekte	Konstante Referenzen	Referenzen



Vorige Seite: 8.5.1 *Call by Value* und *Call by Reference* **Eine Ebene höher:** 8.5 Parameterübergabe mit Zeigern

Vorige Seite: 8.5.2 Gegenüberstellung der zwei **Eine Ebene höher:** 8 Höhere Datentypen und **Nächste Seite:** 8.6.1 Definition von Klassen

---

8.6 Strukturierte Datentypen: Klassen

Ähnlich wie Vektoren Zusammenfassungen von Elementen gleichen Typs zu größeren Einheiten sind, so sind Klassen in einer ersten Näherung Zusammenfassungen von Elementen beliebiger (also auch verschiedener) Typen zu einer Einheit. An dieser Stelle erfolgt nur eine kurze Einführung, da Klassen das zentrale Thema von Kapitel 11 sind und dort ausführlich behandelt werden.

C++ kennt drei verschiedene Klassentypen, die sich im Detail unterscheiden: `class`, `struct` und `union`. Vorerst sind die Klassentypen `class` und `struct` von Interesse.

Vorige Seite: 8.6 Strukturierte Datentypen: Klassen **Eine Ebene höher:** 8.6 Strukturierte Datentypen: Klassen **Nächste Seite:** 8.6.2 Die Elementoperatoren `.` und `->`

8.6.1 Definition von Klassen

Eine Klasse `Student` wird wie folgt vereinbart:

```
class Student {    // Definieren der Klasse Student
  char  name[50];
  int   kNr;
  char  matNr[7];
};
```

Die Einheit `Student` besteht aus den Elementen `name`, `kNr` und `matNr`. Wichtig ist zu wissen, daß Klassen im Gegensatz zu `RECORD` von Modula2 oder Oberon eine Art ,,Sichtbarkeitsschutz`` für ihre Elemente kennen. Eine `class` verbirgt standardmäßig alle Elemente vor der Außenwelt. Die Elemente `name`, `kNr` und `matNr` sind zwar im Programmtext sichtbar, können aber außerhalb der Klasse nicht verwendet werden.

Um eine Verwendung zu ermöglichen, müssen die einzelnen Elemente mit dem Schlüsselwort `public:` als öffentlich zugänglich vereinbart werden:

```
class Student {     // Definieren der Klasse Student
public:             // Alle Elemente oeffentlich
  char  name[50];
  int   kNr;
  char  matNr[7];
};
```

Der Klassentyp `struct` ist eine `class`, die von Haus aus alle Elemente als `public` vereinbart. Obige Deklaration von `Student` könnte also wie folgt als `struct` vereinbart werden:

```
struct Student {
  char  name[50];
  int   kNr;
  char  matNr[7];
};
```

Im folgenden werden zwei Variablen vom Typ `Student` angelegt:

```
Student  s1, s2;
Student* ps;      // Zeiger auf Student
```

`Student`-Variablen können als Parameter übergeben, zugewiesen und als Rückgabewert retourniert werden. Die Übergabe als Parameter erfolgt durch Kopieren (sie wird im Detail im Zusammenhang mit Klassen besprochen). Die Zuweisung von Variablen eines Klassentyps erfolgt durch *Memberwise Assignment* , also elementweises Kopieren. Vergleiche von `Student`-Variablen sind nicht möglich, dazu muß ein entsprechender Operator vereinbart werden (Kapitel 11).

Eine Deklaration einer Klasse verbirgt (wie andere Deklarationen) gleiche Bezeichner, die in einem umgebenden Gültigkeitsbereich angegeben sind. Eine Klasse `Student` kann aber (auch wenn es wenig sinnvoll scheint) ohne Probleme in einem Gültigkeitsbereich mit einer Variable, einem Enumerator oder einer Funktion gleichen Namens deklariert werden:

```
void Student();     // Funktion Student

struct Student {    // Klasse Student
  char  name[50];
  int   kNr;
  char  matNr[7];
};
```

Allerdings kann die Klasse `Student` nun nicht mehr unter diesem Namen referenziert werden - der Name `Student` bezieht sich auf die Funktion gleichen Namens. Um die Klasse anzusprechen, muß ein sogenannter *elaborated-type-specifier* (Schlüsselwörter `class`, `struct` oder `union`) benutzt werden:

```
Student s;        // Fehler Student ist eine Funktion!

struct Student s; // OK
```

Vorige Seite: 8.6 Strukturierte Datentypen: Klassen **Eine Ebene höher:** 8.6 Strukturierte Datentypen: Klassen **Nächste Seite:** 8.6.2 Die Elementoperatoren `.` und `->`

Vorige Seite: 8.6.1 Definition von Klassen **Eine Ebene höher:** 8.6 Strukturierte Datentypen: Klassen

8.6.2 Die Elementoperatoren . und ->

Ähnlich wie bei Vektoren können natürlich auch die einzelnen Elemente von Klassen angesprochen werden. Für den direkten Zugriff auf Komponenten verwendet man die Operatoren `.` oder `->`.

Da es sich bei Klassen nicht um eine Aneinanderreihung von Elementen gleichen Typs handelt, wird kein Index zur ,,Adressierung`` der Elemente verwendet, sondern der Komponentenname. Der Unterschied zwischen den beiden Operatoren `.` und `->` besteht darin, daß `.` auf eine Variable eines Klassentyps angewandt wird, während `->` für Zeiger auf Klassentypen benutzt wird.

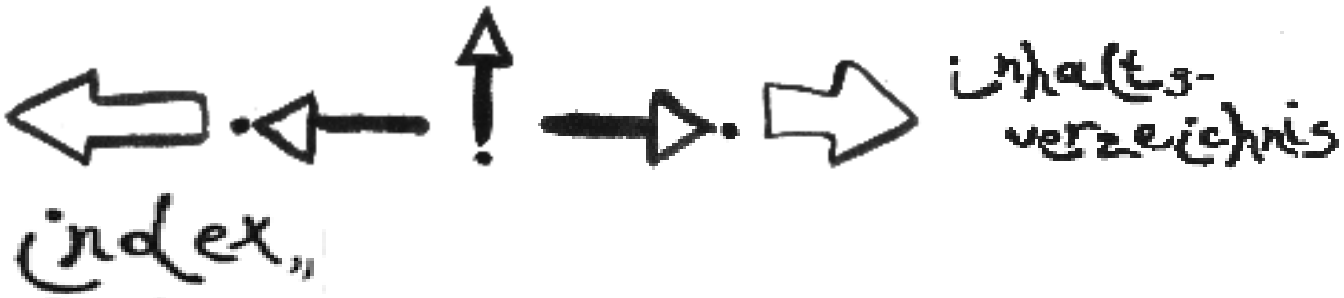
```
struct Birthday {
   int year;
   int month;
   int day;
};


Birthday   b;
Birthday*  p;

b.year = 1991;
p = &b;
p->month = 12;
p->day = 31;
cout << b.day << "-" << b.month << "-" << b.year;
```

Der Operator `->` stellt nichts anderes als eine vereinfachte Schreibweise für eine Kombination von `*` und `.` dar:

```
p->month = 12;
```

ist äquivalent zu:

```
(*p).month = 12
```



Vorige Seite: 8.6.2 Die Elementoperatoren . und -> **Eine Ebene höher:** C++ Programmieren mit Stil
Nächste Seite: 9.1 Gültigkeitsbereiche, Namensräume und

---

9 Gültigkeitsbereiche, Deklarationen und Typumwandlungen

Dieses Kapitel behandelt die Themen Gültigkeitsbereiche, Deklarationen und Typumwandlungen:

- Im ersten Teil werden die verschiedenen Arten von Gültigkeitsbereiche und das Konzept der Namensräume erläutert.
- Der zweite Teil behandelt die verschiedenen Möglichkeiten von Deklarationen im Detail. Der Abschnitt ergänzt die bisher vorgestellten ,,einfachen`` Deklarationen aus Kapitel 4.
- Den Abschluß bildet ein Abschnitt über Typumwandlungen.

Vorige Seite: 9 Gültigkeitsbereiche, Deklarationen und **Eine Ebene höher:** 9 Gültigkeitsbereiche, Deklarationen und **Nächste Seite:** 9.1.1 Gültigkeitsbereiche

9.1 Gültigkeitsbereiche, Namensräume und Sichtbarkeit

Wie bereits in Abschnitt 6.3 besprochen, wird jener Bereich im Programmtext, in dem ein deklarierter Bezeichner verwendet werden kann, als Gültigkeitsbereich dieses Bezeichners (*Scope*) verstanden. [Str92] spricht in diesem Zusammenhang von einem ,,Bezugsrahmen`` .

Nicht mit dem Begriff Gültigkeitsbereich verwechselt werden darf der Begriff Sichtbarkeit. Unter Sichtbarkeit eines Objekts versteht man den Bereich eines Programms, in dem auf das Objekt zugegriffen werden kann. Objekte können nur dann sichtbar sein, wenn sie gültig sind. Sie können aber gültig sein *und* von einem anderen Objekt gleichen Namens ,,überdeckt`` werden, also unsichtbar sein:

```
{
  int i;   // i ab hier gueltig
  ...
  {
    int i; // ueberdeckt aeusseres i
    ..
  }        // inneres i ab hier ungueltig
  ..       // aeusseres i wieder sichtbar
}          // aeusseres i ab hier ungueltig
```

Vorige Seite: 9.1 Gültigkeitsbereiche, Namensräume und **Eine Ebene höher:** 9.1 Gültigkeitsbereiche, Namensräume und **Nächste Seite:** 9.1.2 Namensräume

9.1.1 Gültigkeitsbereiche

C++ kennt verschiedene Gültigkeitsbereiche:

- Blockanweisungen führen einen eigenen Gültigkeitsbereich ein, den sogenannten lokalen Gültigkeitsbereich oder *Local Scope*. Alle dort deklarierten Bezeichner gelten in diesem Block, genauer gesagt von ihrer Deklaration an bis zum Ende des aktuellen Blocks. Unter diesen Punkt fallen auch die Kontrollanweisungen `if`, `switch`, `for` sowie `while` mit ihrem oft auch als *Substatement Scope* bezeichneten Gültigkeitsbereich.
- Der Gültigkeitsbereich von Funktionsprototypen erstreckt sich bis an das Ende der Deklaration und umfaßt die Funktionsparameter. Der Gültigkeitsbereich von Funktionen (also der Funktionsdefinition) erstreckt sich über die gesamte Funktion.
- Die sogenannten Namensräume (*Namespaces*, siehe Abschnitt 9.1.2) sind eigene Gültigkeitsbereiche, die alle darin deklarierten Bezeichner umfassen. Ein Bezeichner, der in einem Namensraum deklariert ist, gilt von seiner Deklaration bis an das Ende des Namensraums.
- Jede Klasse hat einen eigenen Gültigkeitsbereich (*Class Scope*), der sich über die gesamte Klasse erstreckt. Ein Klassen-Element gilt in seiner Klasse von seiner Deklaration an bis zum Ende der Klassendeklaration und kann nur in Verbindung mit einer entsprechenden Variable dieses Klassentyps (beziehungsweise mit dem Klassennamen) verwendet werden.

```
int x;            // "globaler" Bereich (Namensraum)
struct T {        // T hat einen eigenen Gueltigkeits-
                  // bereich:
  int x, y;       //  x,y in Klasse T
};

T o;

o.x = 7;
```

Alle Objekte, die in einem lokalen Gültigkeitsbereich (lokale Objekte in einer Funktion, Parameter, Objekte in einer Blockanweisung) deklariert sind, werden als lokale Objekte bezeichnet.

Dieser Gültigkeitsbereich wird in Kapitel 11 noch ausführlich behandelt werden.

Ein abschließendes Beispiel soll die Begriffe Gültigkeitsbereich und Sichtbarkeit noch einmal illustrieren:

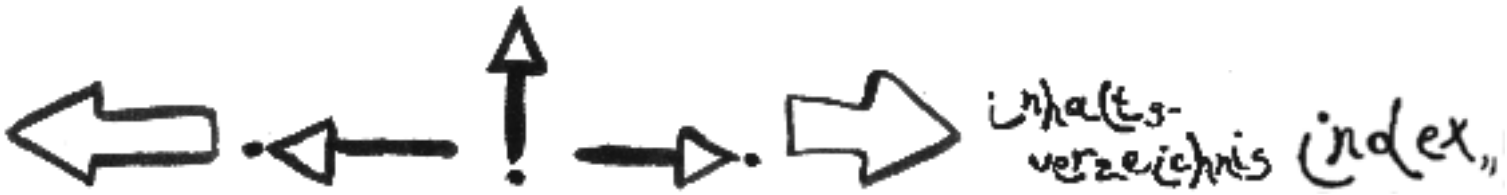
```
#include <iostream>
double x = 1.23;

void f();

void main()
{
   cout << "\n" << x;
   int x = 10;            // ueberdeckt globales x
   cout << "\n" << x;
   {
      float x = 47.11;    // ueberdeckt beide x
      cout << "\n" << x;
      ::x = 3.14;         // globales x
   }
   cout << "\n" << x;

   f();
}

void f()
{
   cout << "\n" << x;       // Ausgabe globales x
}
```

Einige Bemerkungen zum oben angeführten Beispiel:

- Der *Scope*-Operator `::` erlaubt es, auf den globalen Gültigkeitsbereich zuzugreifen. Die Anweisung `::x` referenziert die globale Variable `x`. Eine andere Variante des Operators wird im nächsten Abschnitt beziehungsweise in Kapitel 11 besprochen und erlaubt den Zugriff auf beliebige benannte Gültigkeitsbereiche. Dazu wird dem Operator der Name des jeweiligen Gültigkeitsbereichs vorangestellt (*Name*`::`*Bezeichner*).
- Neben

  den bereits bekannten Sichtbarkeits- und Gültigkeitsregeln ist vor allem zu beachten, daß der Gültigkeitsbereich von Objekten statisch bestimmbar ist. Das heißt, die Funktion `f` gibt immer den Wert der globalen Variablen `x` aus, unabhängig davon, von wo aus die Funktion aufgerufen wurde.
- Der (namenlose) globale Namensraum erstreckt sich von Beginn des Programms bis zum Ende. Er enthält eine globale Variable `x` und zwei Funktionen `f` und `main`. `f` gibt die globale Variable `x` aus. In `main` werden zwei weitere Variablen `x` deklariert, die ebenfalls ausgegeben werden. Es entsteht folgende Ausgabe:

  ```
  1.23
  10
  47.11
  10
  3.14
  ```



Vorige Seite: 9.1 Gültigkeitsbereiche, Namensräume und **Eine Ebene höher:** 9.1 Gültigkeitsbereiche, Namensräume und **Nächste Seite:** 9.1.2 Namensräume

Vorige Seite: 9.1.1 Gültigkeitsbereiche **Eine Ebene höher:** 9.1 Gültigkeitsbereiche, Namensräume und

Teilabschnitte

9.1.2 Namensräume

Namensräume (*Namespaces*) sind Gültigkeitsbereiche, in denen beliebige Bezeichner (Variablen, Klassen, Funktionen, andere Namensräume, Typen etc.) deklariert werden können. Ein Namensraum wird dazu benutzt, aus Elementen logische Gruppen zu bilden. Damit ist es zum Beispiel möglich, alle Programmelemente einer Bibliothek *Graphic* in einem entsprechenden Namensraum zusammenzufassen. Die einzelnen Elemente dieser Bibliothek können dann in keinem Namenskonflikt mit Elementen gleichen Namens einer anderen Bibliothek stehen. So kann zum Beispiel in den Bibliotheken *Graphic* und *Collections* je eine Funktion `initialize` deklariert werden, ohne daß es zu Namenskonflikten kommt.

Deklaration und Verwendung von Namensräumen

C++ stellt im Zusammenhang mit Namensräumen folgende Mechanismen zur Verfügung:

- Ein Namensraum kann deklariert werden. Alle enthaltenen Objekte werden diesem Namensraum zugeordnet. Auf Bezeichner eines Namensraums kann mit dem *Scope*-Operator `::` zugegriffen werden.
- Einem Namensraum kann ein sogenannter *Alias* zugeordnet werden, über den er angesprochen wird.
- Eine sogenannte *Using*-Deklaration erlaubt den direkten Zugriff (das heißt ohne explizite Qualifikation über den *Scope*-Operator) auf einen Bezeichner eines Namensraums.
- Mit einer sogenannten *Using*-Direktive kann auf alle Bezeichner eines Namensraums direkt

zugegriffen werden.

Namensräume werden deklariert, indem hinter dem Schlüsselwort `namespace` der Name des Namensraums gefolgt von geschweiften Klammern angegeben wird. Ähnlich einem lokalen Block umschließen die Klammern alle Deklarationen des Namensraums:

```
namespace MyLib1
{
  int i;
  void foo();
}

namespace MyLib2
{
  int i;
  void foo();
}
```

Beide angeführten Namensräume enthalten Bezeichner desselben Namens. Über den *Scope*-Operator `::` kann auf die Elemente der Namensräume `MyLib1` und `MyLib2` zugegriffen werden:

```
MyLib1::foo();   // foo aus MyLib1
MyLib2::i = 17;  // i aus MyLib2
```

Mit einem *Namespace Alias* kann ein anderer Name für einen Namensraum vereinbart werden. Damit können Namenskonflikte vermieden werden oder zum Beispiel kürzere Namen verwendet werden:

```
namespace FBSSLib = Financial_Branch_and_System_Service_Library;
FBSSLib::anyFoo();
```

Using-Deklarationen ,,importieren`` Namen aus einem Namensraum und machen ihn ohne die explizite Namensraumangabe verwendbar:

```
void main()
{
  using MyLib1::foo;  // Lokales Synonym
  foo();              // ruft MyLib1::foo auf
}
```

Eine *Using*-Deklaration stellt eine Deklaration (keine Definition) eines Namens im aktuellen Gültigkeitsbereich dar.

Using-Direktiven machen alle Namen aus einem Namensraum ohne explizite Scope-Angabe verwendbar:

```
using namespace MyLib1;  // importiert MyLib1

foo();              // MyLib1::foo
MyLib2::i = 17;     // i aus MyLib2
```

Eine *Using*-Direktive entspricht nicht einer Deklaration aller Namen eines Namensraums im aktuellen Gültigkeitsbereich, sondern einer einfachen Bekanntgabe des Namensraums. Dazu ist es wichtig zu wissen, wie das C++-System nach einem Bezeichner ,,sucht``(*Name Lookup*):

Findet der Compiler in einem Programm einen Bezug auf einen Bezeichner (Funktion, Variable, Typ etc.), so versucht er, diesen Bezug aufzulösen (Namensauflösung). Dabei geht er nach folgenden Regeln vor:

1. Zuerst wird die Liste der lokalen Deklarationen nach dem Bezeichner durchsucht.
2. Wird er in dieser Liste nicht gefunden, so werden der Reihe nach alle Bereiche, in denen der aktuelle Gültigkeitsbereich enthalten ist, durchsucht.
3. Als letztes werden alle importierten Namensräume nach dem Bezeichner durchsucht. Das umfaßt auch den sogenannten ,,globalen`` Namensraum, der in C++ durch einen namenlosen Namensraum (siehe unten) dargestellt wird.

Ein Beispiel illustriert die Namensauflösung:

```
namespace MyNameSpace
{
  int i=0;
}
using namespace MyNameSpace;

int j=1;

void foo()
{
  int k=2;
  {
    int l=3;

    l++;  // Suche in lokalen Deklarationen erfolgreich (1)
    k++;  // Suche in Gueltigkeitsbereich erfolgreich (2)
    j++;  // Suche im "globalen" Namensraum erfolgreich (3)
    i++;  // Suche in einem importierten Namensraum
  }       // erfolgreich (3)
}
```

Using-Direktiven werden zum Beispiel verwendet, um ,,alte`` C++-Programme auch weiterhin mit den Bibliotheken des C++-ANSI/ISO-Standards verwenden zu können. In diesem sind Bibliotheken in Namensräumen (zum Beispiel `std`) organisiert. Da ,,alte`` C++-Programme keine Namensräume kennen, wird eine *Using*-Direktive verwendet:

```
using namespace std;  // Macht alle Elemente
                      // von std sichtbar!
```

Namenlose Namensräume

Ein Namensraum ohne Namen wird behandelt wie ein spezieller Namensraum mit einem systemweit eindeutigen Namen und impliziter *Using*-Direktive. Der Namensraum

```
namespace
{
  ...
}
```

ist damit identisch zu folgender Konstruktion:

```
namespace SystemweitEindeutigerName
{
  ...
}
using namespace SystemweitEindeutigerName;
```

Tatsächlich hat jeder unbenannte Namensraum automatisch einen Namen, der allerdings vom Compiler automatisch generiert wird und für den Programmierer nicht sichtbar ist.

Der globale Gültigkeitsbereich wird als namenloser Namensraum betrachtet. Auf Elemente dieses Namensraums kann über den speziellen Operator `::` zugegriffen werden:

Namenlose Namensräume helfen,einige ,,Unsauberkeiten`` von C und C++ zu überdecken (Globaler Gültigkeitsbereich, Gültigkeitsbereich Übersetzungseinheit). So werden sie zum Beispiel in C++ verwendet, um globale Variablen mit dem Gültigkeitsbereich der aktuellen Übersetzungseinheit (zum Beispiel eine Datei) zu deklarieren. In C beziehungsweise früheren Versionen von C++ wurde dazu das Speicherklassenattribut `static` verwendet (siehe Abschnitt 9.2.1). Die Anweisung

```
static int i;
```

deklariert eine Variable vom Typ `int` mit statischer Lebensdauer *und* einer eingeschränkten Bindung (gilt nur in der aktuellen Übersetzungseinheit, näheres dazu folgt im nächsten Abschnitt).

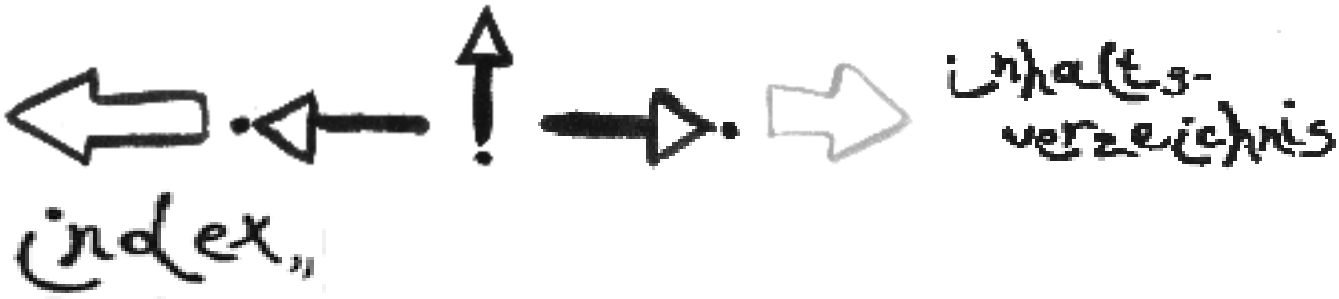
Dasselbe Ergebnis kann erreicht werden, indem ein namenloser Namensraum verwendet wird:

```
namespace
{
  int i;
}
```

`i` ist in einem eigenen Namensbereich deklariert und daher ebenfalls nur in der aktuellen Datei gültig.

Die Verwendung von `static` für diesen Zweck ist zwar üblich aber nicht anzuraten, da hier eine ,,Nebenwirkung`` des Speicherklassenattributs ausgenutzt wird.

Es ist guter Programmierstil, den Gültigkeitsbereich aller nur intern verwendeten Funktionen und Daten mit Hilfe von namenlosen Namensräumen auf den Bereich einzugrenzen, in dem die Objekte verwendet werden.



Vorige Seite: 9.1.1 Gültigkeitsbereiche **Eine Ebene höher:** 9.1 Gültigkeitsbereiche, Namensräume und

Vorige Seite: Namenlose Namensräume **Eine Ebene höher:** 9 Gültigkeitsbereiche, Deklarationen und
Nächste Seite: 9.2.1 Speicherklassenattribute

9.2 Deklarationen

In Kapitel 4 wurden bereits einfache Deklarationen besprochen. Diese Darstellung ist allerdings sehr vereinfachend. Im Detail sind Deklarationen nach der folgenden Grammatik aufgebaut [ISO95, dcl.dcl]:

declaration-seq:
declaration
declaration-seq declaration
declaration:
block-declaration
function-definition
template-declaration
linkage-specification
namespace-definition
block-declaration:
simple-declaration
asm-declaration
namespace-alias-definition
using-declaration
using-directive
simple-declaration:
decl-specifier-seq$_{opt}$ *init-declarator-list*$_{opt}$;

Eine Deklaration von Bezeichnern besteht aus folgenden Teilen: Der deklarierte Bezeichner

(beziehungsweise die deklarierten Bezeichner) ist (sind) in *init-declarator-list* angegeben. Der Typ ist in *decl-specifier-seq* und der rekursiven Folge von *declaration-seq* festgelegt, die vor dem Bezeichner angeführt sind.

decl-specifier-seq:

decl-specifier-seq$_{opt}$ *decl-specifier*

decl-specifier:

storage-class-specifier

type-specifier

function-specifier

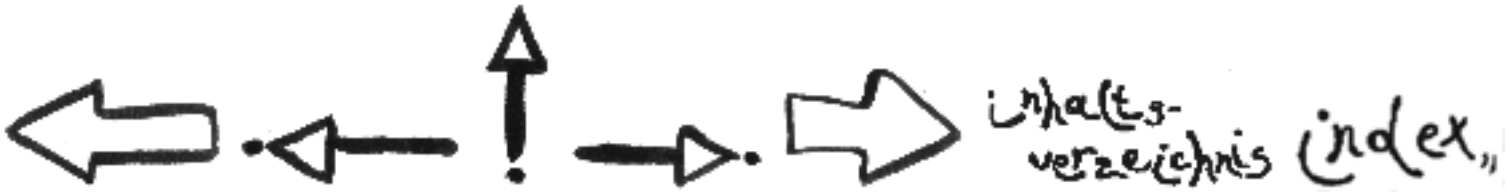
`friend`

`typedef`

decl-specifier werden wie folgt unterschieden:

- Das Speicherklassenattribut *storage-class-specifier* gibt die Speicherklasse eines Bezeichners an und bestimmt Lebensdauer und Bindung (*Linkage*) von Bezeichnern.
- Der Typ-Qualifikator *type-specifier* umfaßt die verschiedenen Typen von Deklarationen wie zum Beispiel `long` oder `short` und ihre Modifikatoren `const` und `volatile` (siehe Abschnitt 9.2.2).
- Ein *function-specifier* (Funktionsattribut) wiederum spezifiziert eine Funktion näher.
- Das `friend`-Schlüsselwort deklariert einen Funktions- oder Klassenbezeichner als ,,Freund`` mit speziellen Zugriffsregeln.
- Mittels `typedef` können eigene Typen (oder besser gesagt: neue Typnamen) eingeführt werden.

Im folgenden werden die verschiedenen Arten von *decl-specifier* mit Ausnhame von `friend` genauer beschrieben. `friend` wird im Zusammenhang mit Klassen in Abschnitt 11.4.2 besprochen.



Vorige Seite: Namenlose Namensräume **Eine Ebene höher:** 9 Gültigkeitsbereiche, Deklarationen und
Nächste Seite: 9.2.1 Speicherklassenattribute

Vorige Seite: 9.2 Deklarationen **Eine Ebene höher:** 9.2 Deklarationen **Nächste Seite:** 9.2.2 Typ-Qualifikatoren

---

Teilabschnitte

9.2.1 Speicherklassenattribute

Es existieren fünf verschiedene Speicherklassenattribute [ISO95, dcl.stc]:

```
storage-class-specifier:
                auto
                register
                static
                extern
                mutable
```

Jede Deklaration darf genau ein Speicherklassenattribut umfassen, wobei sich das Vorhandensein von Speicherklassenattributen und des Schlüsselworts `typedef` wechselseitig ausschließen.

Speicherklasse `auto`

Wird bei der Deklaration einer Variablen keine Speicherklasse angegeben, so ist der Standardwert `auto`. Eine mit `auto` deklarierte Variable hat den Gültigkeitsbereich des Blocks, in dem sie deklariert wurde. Sie wird bei der Deklaration automatisch angelegt und am Ende des aktuellen Blocks automatisch zerstört.

```
{                       // (1)
  int x, y;             // (2)
  x=1; y=2;             // (3)
  {                     // (4)
    auto int x;         // (5)
    x=7;                // (6)
    cout << x << y;     // (7)
  }                     // (8)
  cout << x << y;       // (9)
}                       // (10)
```

Im Beispiel oben sind die beiden Bezeichner `x` und `y` von der Speicherklasse `auto`. Sie gelten im Block (1)-(10) und werden automatisch beim Erreichen von (2) angelegt. Im inneren Block (4)-(8) erfolgt eine erneute Deklaration eines Bezeichners mit Namen `x` (5). Diese Deklaration ,,überdeckt`` den in (2) deklarierten Bezeichner `x`. Überdecken bedeutet, daß sich in (5)-(8) ein `x` auf den in (5) deklarierten Bezeichner bezieht. Das heißt aber nicht, daß die in (2) deklarierte Variable `x` nicht mehr existiert! Die Angabe `auto` kann auch entfallen, da dies die Standard-Speicherklasse ist. In (7) erfolgt die erste Ausgabe: `7` und `2`.

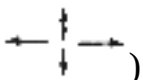
Beim Erreichen des inneren Blockendes (8) werden alle lokalen Objekte (`x`) automatisch zerstört. Die Ausgabe von `x` und `y` in (9) bezieht sich also wieder auf die in (2) deklarierten Variablen und führt zur Ausgabe von `1` und `2`.

Die Speicherklasse `auto` ist ausschließlich für lokale Objekte gültig - kein global gültiger Bezeichner kann als `auto` spezifiziert werden.

Speicherklasse `register`

Die Speicherklasse `register` entspricht der Speicherklasse `auto` und stellt zusätzlich einen ,,Hinweis`` an den Compiler dar, daß der Bezeichner sehr intensiv benutzt wird und daher aus Effizienzgründen möglichst in einem Register abgelegt werden sollte. Der Compiler kann den Hinweis ignorieren.

Speicherklasse `static`

Eine `static`-Variable lebt von der ersten Verwendung bis zum Programmende (statische Lebensdauer). Zusätzlich verändert `static` die Art und Weise, wie ein Programm gebunden (= aus mehreren Dateien zusammengefügt) wird: Ein Objekt, das in einer Datei als `static` deklariert ist, bleibt nur in dieser Datei bekannt - es erfolgt kein Binden über Dateigrenzen hinweg (siehe Beispiel auf Seite ). Als `static` können Funktionen und Variablen vereinbart werden, aber keine Parameter.

`static` *kann* benutzt werden, um ,,interne`` Funktionen und Daten zu vereinbaren, das heißt Funktionen und Daten, die ausschließlich lokal in einer Datei Gültigkeit haben. Dies sollte aber nicht geschehen, da zur Kennzeichnung von internen Objekten das Konzept von unbenannten Namensräumen verwendet werden kann (siehe Abschnitt 9.1.2).

In C++ ist es zwar (noch) üblich, aber nicht nötig, interne Objekte durch die Verwendung der Speicherklasse `static` zu kennzeichnen. Die Verwendung von `static` für diesen Zweck ist unklar und zeugt von keinem guten Programmierstil. Für interne Variablen werden ,,namenlose`` Namensräume verwendet.

`static` bewirkt, daß skalare Objekte *vor* ihrer ersten Verwendung automatisch mit Nullwerten initialisiert werden.

Ein Beispiel zum Thema Gültigkeitsbereich in Verbindung mit Speicherklassenattributen und `static`-Variablen:

```
static void foo()   // (1)
{                   // (2)
  static int x;     // (3)
  ++x;              // (4)
  cout << x;        // (5)
}                   // (6)
                    // (7)
static int x;       // (8)
...                 // (9)
cout << x;          // (10)
foo();              // (11)
foo();              // (12)
```

Die in (1) vereinbarte Funktion hat interne Bindung, das heißt die Funktion ist nur in der aktuellen Datei gültig und kann nicht von einer anderen Datei aus verwendet werden. Die Funktion enthält eine statische Variable `x` (3), die in (4) erhöht und in (5) ausgegeben wird. (8) vereinbart eine ,,globale`` Variable (Namensraum nicht angegeben, auf globalem Niveau), die ebenfalls als `static` deklariert ist. Da das Objekt als `static` vereinbart ist, wird es automatisch mit 0 initialisiert. (10) gibt daher `0` aus.

(11) ruft die Funktion `foo` auf. In `foo` wird zunächst (= vor der ersten Verwendung) die Variable `x` in (3) allokiert und mit (0) initialisiert. (4) erhöht `x` und (5) führt zur Ausgabe von `1`. Beim Verlassen von `foo` wird `x` *nicht* zerstört, da `static`-Variablen eine statische Lebensdauer haben. Allerdings kann diese Variable nur innerhalb der Funktion `foo` verwendet werden - außerhalb gilt die in (10) deklarierte Variable. Der erneute Aufruf von `foo` (12) hat zur Folge, daß `x` *nicht* neu angelegt oder initialisiert wird. Die Inkrementierung in (4) führt daher zur Ausgabe von `2` in (5).

Speicherklasse `extern`

Funktionen und Objekte, die in einer Datei auf globalem Niveau deklariert sind, können auch von einer anderen Datei aus benutzt werden, wenn sie nicht als `static` vereinbart sind. Der Linker ,,findet`` diese Objekte, indem er alle Dateien durchsucht. Allerdings muß der Programmierer in jeder Datei die ,,externen`` Funktionen und Objekte dem Compiler bekanntgeben. Ansonsten meldet der Compiler beim

Übersetzen, daß nicht deklarierte Bezeichner verwendet wurden.

Um Funktionen oder andere Objekte als ,,extern`` zu deklarieren, wird das Speicherklassenattribut `extern` verwendet. `extern` gibt einen Namen bekannt, vereinbart aber keinen Code beziehungsweise Speicherplatz.

Alle Bezeichner, die als `static` vereinbart sind, haben interne Bindung. Alle anderen Bezeichner haben automatisch externe Bindung und können auch von anderen Dateien aus verwendet werden. Eine Ausnahme sind `const`-Bezeichner. Sie sind nicht explizit als `extern` vereinbart und damit automatisch ,,intern``.

```
Datei a.cpp:                Datei b.cpp:
static void a_foo1()        extern int y;          // (1)
{                           extern int x;          // (2)
}                                                  // (3)
                            extern void a_foo1(); // (4)
void a_foo2()               extern void a_foo2(); // (5)
{                                                  // (6)
}                           y++;                   // (7)
                            x++;                   // (8)
static int x;               a_foo1();              // (9)
int y;                      a_foo2();              // (10)
```

Im Beispiel oben sind zwei Dateien (`a.cpp` und `b.cpp`) angeführt, die beide getrennt übersetzt und anschließend gebunden werden. Die Datei `a.cpp` enthält eine statische Funktion (`a_foo1`), eine ,,normale`` Funktion (`a_foo2`), ein statisches `int`-Objekt (`x`) und ein ,,normales`` `int`-Objekt (`y`).

Datei `b.cpp` benutzt diese von `a.cpp` vereinbarten Bezeichner. (1) ist eine Bekanntgabe des externen Bezeichners `y`, (2) gibt den externen Bezeichner `x` bekannt. Diese Anweisung ist fehlerhaft, da `x` in `a.cpp` als `static` vereinbart ist und daher ausschließlich interne Bindung hat. `x` kann ausschließlich in `a.cpp` benutzt werden. Ebenso verhält es sich mit der in (4) angeführten Deklaration: `a_foo1` ist als `static` vereinbart und kann nur in der Datei `a.cpp` verwendet werden.

Speicherklasse `mutable`

Das Speicherklassenattribut `mutable` kann nur auf Klassen-Elemente angewandt werden. Es spezifiziert ein Klassen-Element als ,,veränderbar``. Damit können einzelne Klassen-Elemente auch dann verändert werden, wenn eine entsprechende Variable als `const` vereinbart wurde. `mutable` kann aber nicht auf Klassen-Elemente angewandt werden, die explizit als `const` oder `static` deklariert wurden.

Die Anwendung dieses Speicherklassenattributs wird in Kapitel 11 im Zusammenhang mit Klassen gezeigt.

Der Zusammenhang zwischen Speicherklasse, Lebensdauer und Gültigkeit

Zusammenfassend werden hier die verschiedenen Speicherklassenattribute und ihre Auswirkungen auf Lebensdauer und Gültigkeitsbereich von Objekten dargestellt.

Objekte, die mit `static` oder `extern` deklariert sind, besitzen eine statische Lebensdauer. Diese Objekte werden beim Programmstart allokiert und erst dann zerstört, wenn das Programm beendet wird (siehe Tabelle 9.1).

Dynamische Objekte sind nicht in Tabelle 9.1 angeführt. Bei ihnen handelt es sich um Objekte, denen mit speziellen Funktionen während des Programmablaufes Speicherplatz zugeteilt und entzogen wird. Werden dynamische Objekte verwendet, so wird der benötigte Speicherplatz explizit vom Benutzer angefordert beziehungsweise freigegeben. Diese Objekte leben also von ihrer Allokierung bis zu ihrer Zerstörung.

Speicherklasse	Lebensdauer	Gültigkeit
`auto`	Block	lokal
`register`	Block	lokal
`static`	Statisch	lokal
`extern`	Statisch	global

Tabelle 9.1: Speicherklasse, Lebensdauer und Gültigkeit

Vorige Seite: 9.2 Deklarationen **Eine Ebene höher:** 9.2 Deklarationen **Nächste Seite:** 9.2.2 Typ-Qualifikatoren

Vorige Seite: 9.2.1 Speicherklassenattribute **Eine Ebene höher:** 9.2 Deklarationen **Nächste Seite:** 9.2.3 Funktionsattribute

9.2.2 Typ-Qualifikatoren

Jede der Typangaben kann durch die Schlüsselwörter `const` und `volatile` modifiziert werden. Mit den beiden Qualifikatoren ergeben sich vier Möglichkeiten einer Typangabe: Eine Typangabe ohne vorangestelltes `const` oder `volatile` wird in C++ als ,,unqualifizierte Typangabe`` bezeichnet, ein Objekt, dem das Schlüsselwort `const` (beziehungsweise `volatile`) vorangestellt ist, wird als `const`-Objekt (beziehungsweise `volatile`-Objekt) bezeichnet. Die Kombination von `const` und `volatile` wird demnach als `const volatile` bezeichnet.

`const`-Objekte dürfen nicht verändert werden. Sie werden als *RValues* betrachtet und können daher nicht mehr

- auf der linken Seite einer Zuweisung vorkommen,
- als Aktualparameter verwendet werden, wenn als Formalparameter ein nicht konstanter Ausdruck angegeben ist, und
- nicht mittels (non-`const`-)Methoden verändert werden (siehe Abschnitt 11.2.2).

`volatile`-Variablen sind Variablen, deren Inhalt sich auf eine Art verändern kann, die vom Compiler nicht ohne weiteres festzustellen ist. Der Compiler wird damit angewiesen keine aggressiven Optimierungen (soweit sie die Variable betreffen) vorzunehmen. Auch Methoden können als `volatile` deklariert werden.

Typen, die mit `const` oder `volatile` qualifiziert sind, unterscheiden sich von den ursprünglichen Typen. Das heißt eine Funktion mit einem Parameter vom Typ `int` ist verschieden von einer Funktion mit demselben Namen und einem Parameter vom Typ `const int`. Das Beispiel unten führt vier Funktionen an, die alle verschiedene Argumenttypen aufweisen:

```
void print(int& i);
void print(const int& i);
void print(volatile int& i);
void print(const volatile int& i);
```

Zwischen den verschiedenen Möglichkeiten, einen Typ `T` zu qualifizieren, besteht eine Ordnungsrelation, die in Tabelle 9.2 angegeben ist.

```
T          < const T
T          < volatile T
T          < const volatile T
```

`const T`	< `const volatile T`
`volatile T`	< `const volatile T`

Tabelle 9.2: Beziehungen zwischen `const`- und `volatile`-Modifikatoren [ISO95, basic.type.qualifier]

Diese Relation ist besonders dann von Bedeutung, wenn eine Variable mit ,,falschem`` Qualifikator verwendet wird, da die Promotion eines Typs auf eine ,,höhere`` Qualifikation erlaubt ist und implizit erfolgt. Wird zum Beispiel die oben angeführte Funktion `print(const int& i)` mit einer nicht konstanten `int`-Variable `a` aufgerufen, so wird `a` automatisch als `const int` qualifiziert, da `T` < `const T`. Im folgenden Beispiel werden vier Funktionen `print` mit jeweils einem `int`-Parameter sowie vier `int`-Variablen vereinbart.

```
void print(int& i);
void printC(const int& i);
void printV(volatile int& i);
void printCV(const volatile int& i);
```

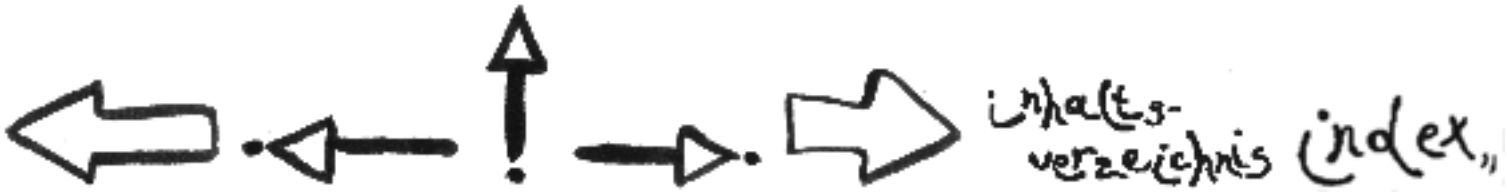
Die unten angeführten Aufrufe der Funktionen sind gültig, da die Aktualparameter in die jeweiligen Formalparameter aufgrund der Ordnungsrelation zwischen den Qualifikatoren umgewandelt werden können:

```
int i;
const int iC;
volatile int iV;
const volatile int iCV;

print(i);
printC(i);
printC(iC);

printV(i);
printV(iV);

printCV(i);
printCV(iC);
printCV(iV);
printCV(iCV);
```

Folgende Aufrufe führen zu Fehlern oder Warnungen beim Übersetzen, da die Ordnungsrelation zwischen den Modifikatoren der Aktual- und Formalparametern nicht beachtet wird:

```
print(iC);
print(iV);
print(iCV);
printC(iV);
printC(iCV);
printV(iC);
```

```
printV(iCV);
```



Vorige Seite: 9.2.1 Speicherklassenattribute **Eine Ebene höher:** 9.2 Deklarationen **Nächste Seite:** 9.2.3 Funktionsattribute

Vorige Seite: 9.2.2 Typ-Qualifikatoren **Eine Ebene höher:** 9.2 Deklarationen **Nächste Seite:** 9.2.4 `typedef`

9.2.3 Funktionsattribute

C++ kennt drei Funktionsattribute:

`function-specifier:`

`inline`

`virtual`

`explicit`

Das Funktionsattribut `inline` weist den Compiler an, anstelle von Funktionsaufrufen textuelle Substitutionen von Funktionsrümpfen vorzunehmen. Die genauen Auswirkungen wurden bereits in Abschnitt 7.5 im Zusammenhang mit `inline`-Funktionen besprochen. Die Bindung von `inline`-Funktionen ist standardmäßig intern. Zu beachten ist, daß kein Funktionsaufruf einer `inline`-Funktion *vor* der eigentlichen Definition erfolgen darf.

Die Funktionsattribute `virtual` und `explicit` können nur in Zusammenhang mit Klassen und Methoden verwendet werden. Sie werden daher in Kapitel 14 beziehungsweise in Abschnitt 11.7.5 erläutert.

Vorige Seite: 9.2.3 Funktionsattribute **Eine Ebene höher:** 9.2 Deklarationen

9.2.4 typedef

Das Schlüsselwort `typedef` ermöglicht die Einführung neuer Bezeichner, die dann im Programm anstelle von anderen Typen verwendet werden können. `typedef` führt allerdings keine neuen Typen sondern Synonyme für einen existierenden Datentyp ein.

```
typedef int Number;           // Number = int
typedef int Vector[25];       // Vector = int[25]
typedef char String[30];      // String = char[30]
typedef float Matrix[10][10]; // Matrix = float[10][10]

Number  a;  // Deklaration einer Ganzzahl
String s;   // Ein String mit 30 Elementen
Matrix m1;  // Deklaration einer Matrix (10x10 floats)
Matrix m2;  // Deklaration einer Matrix (10x10 floats)

  // Deklaration eines Zeigers auf ein Funktion
  // vom Typ void foo(int, char)
typedef void (*PIntCharFct)(int, char);

  // Prototyp einer Funktion:
void foo(int i, char ch);

  // Zuweisung der Adresse von foo an pFoo
PIntCharFct   pFoo = foo;
```

Vorige Seite: 9.2.4 `typedef` **Eine Ebene höher:** 9 Gültigkeitsbereiche, Deklarationen und **Nächste Seite:** 9.4 Typumwandlungen

9.3 Initialisierung

Bei der Initialisierung wird der Anfangswert eines Objekts festgelegt. Initialisiert man ein Objekt der Klasse `auto` nicht, so ist sein Wert unbestimmt; numerische statische Objekte hingegen werden implizit mit null initialisiert.

```
int w=243;      // w=243
int x;          // Wert v. x ist unbestimmt
static int y;   // y = 0
```

Achtung: Die Anweisung `int w=243;` ist eine Deklaration mit Initialisierung. Die Initialisierung ist *keine* Zuweisung! Bei einer Zuweisung existeren bereits zwei Objekte (die linke und die rechte Seite des Operators =), eine Initialisierung hingegen ist das Festlegen des Variablen-Zustands beim Anlegen!

Bei der expliziten Initialisierung sind folgende Regeln zu beachten:

- Alle Initialisatoren müssen Konstantenausdrücke sein.
- Konstantenausdrücke müssen eine Konstante oder die Adresse eines externen oder statischen Objekts plus minus einer Konstante liefern.
- Bei Konstantenausdrücken dürfen nur unäre und binäre Operatoren sowie Funktionen verwendet werden.
- Die Werte von Vektoren und strukturierten Datentypen werden durch die Angabe der einzelnen Komponenten in geschwungenen Klammern festgelegt.

```
int x = 1234;  // Variable x wird mit 1234 initialisiert
int y = 12/4;  // Variable y wird mit 3 initialisiert
int z = f(y);  // Variable z wird mit Rueckgabewert der
               // Funktionsprozedur f(y) initialisiert

char* str = "das_ist_ein_String";
char  myName[] = {'T','h','o','m','a','s'};
int   help[4][2] = { { 1, 2}, {3, 4}, {5, 6}, {7, 8}};


typedef struct {
   int day;
   int month;
```

```
   int year;
   char name[30];
   long matrnr;
} StudentType;

StudentType student =
   {11, 11, 1970, "Rudisch", 8999777};
```

Vorige Seite: 9.3 Initialisierung **Eine Ebene höher:** 9 Gültigkeitsbereiche, Deklarationen und **Nächste Seite:** 9.4.1 Standard-Typumwandlung

9.4 Typumwandlungen

Auf Maschinenebene werden alle Datenobjekte als Bitkette dargestellt. Diese Bitkette wird vom System dann je nach Typ zum Beispiel als `int`- oder als `char`-Objekt interpretiert. Wird eine Variable eines Datentyps in einen anderen Datentyp umgewandelt, so bleibt die ursprüngliche Bitkette zwar erhalten, allerdings ändern sich Interpretation und Länge der Bitkette (`int`-Zahlen haben zum Beispiel eine Länge von zwei beziehungsweise vier Bytes, `char`-Werte sind ein Byte lang).

Können bei der Typumwandlung signifikante Stellen verloren gehen, so spricht man von unsicheren Konvertierungen (*Unsafe Conversion*), andernfalls von sicheren Konvertierungen (*Safe Conversion*). Beispiele für unsichere Typumwandlungen sind die Konvertierung von Gleitpunktwerten auf ganzzahlige Werte oder von `long`-Zahlen auf den Datentyp `int`.

In C++ wird zwischen Standard-Typumwandlungen und expliziten Typumwandlungen unterschieden. Standard-Konvertierungen werden vom System automatisch vorgenommen, explizite hingegen vom Programmierer. Daneben existiert noch die Möglichkeit, Typumwandlungen selbst zu definieren (siehe Abschnitt 11.7.5).

Vorige Seite: 9.4 Typumwandlungen **Eine Ebene höher:** 9.4 Typumwandlungen **Nächste Seite:** 9.4.2 Explizite Typumwandlung

---

Teilabschnitte

9.4.1 Standard-Typumwandlung

Ausdrücke werden implizit (= automatisch) in Ausdrücke anderen Typs umgewandelt, wenn der tatsächliche Typ eines Ausdrucks und der ,,erwartete`` Typ nicht übereinstimmen. Möglich ist das zum Beispiel, wenn ein Ausdruck

- als Operand eines Ausdrucks verwendet wird und der erwartete Typ nicht mit dem Ergebnistyp dieses Ausdrucks übereinstimmt,
- in einer Bedingung (logischer Ausdruck) verwendet wird,
- an einer Stelle verwendet wird, an der ein Integer-Ausdruck (zum Beispiel in einer `case`-Anweisung) erwartet wird, oder
- in einer Initialisierung verwendet wird und der Typ des zu initialisierenden Objekts nicht mit dem Typ des Ausdrucks übereinstimmt.

Derartige Umwandlungen werden Standard-Typumwandlungen genannt. Daneben gibt es verschiedene Möglichkeiten zur benutzerdefinierten Typumwandlung von Ausdrücken im Zusammenhang mit Klassen, die in Abschnitt 9.4.2 beziehungsweise in den Kapiteln 11 und 14 beschrieben werden.

LValue-RValue-Typumwandlungen

Ein *LValue*-Ausdruck kann in einen *RValue*-Ausdruck umgewandelt werden. Ein Beispiel dafür ist die Verwendung eines *LValue*-Operanden im Zuammenhang mit einem Operator, der einen *RValue* erfordert:

```
cout << a;  // << fordert RValue, a ist aber
            // ein LValue, daher: Konvertierung
```

Vektor-Zeiger-Typumwandlungen

Jeder Vektor kann in einen konstanten Zeiger auf seine Basisadresse (das erste Element) umgewandelt werden.

```
char  f[100];  // Zeichenkette f
char* p;
int   len;

cin >> f;      // f einlesen
p = f;         // Konvertierung!
while (*p != 0x00) { ++p; }
len = p-f;
```

Funktion-Zeiger-Typumwandlungen

Ähnlich wie Vektornamen geben auch Funktionsnamen die Adressen der jeweiligen Funktionen an.

```
void (*fctPtr)(int);

void myFoo(int i)
{
  ...
}
...
fctPtr = myFoo;  // Konversion Fkt-Name -> Zeiger
                 // auf Funktion
```

Qualifikations-Typumwandlungen

Wie bereits in Abschnitt 9.2.2 gezeigt, können die verschiedenen Qualifikatoren eines Typs anhand einer

Ordnungsrelation zwischen den Qualifikatoren `T`, `const T`, `volatile T` und `const volatile T` umgewandelt werden. Für entsprechende Beispiele sei auf diesen Abschnitt verwiesen.

Integral- und Gleitkomma-Promotionen

RValue-Ausdrücke integralen Datentyps können automatisch in solche eines anderen integralen Datentyps umgewandelt werden. Tabelle 9.3 zeigt, welche integralen Typen (auf der linken Seite) in andere integrale Typen (auf der rechten Seite) automatisch umgewandelt werden. Jede dieser Umwandlungen findet nur dann statt, wenn dabei keine Werte verloren gehen. Das heißt, eine Promotion von `wchar_t` nach `int` findet nur dann statt, wenn `int` einen zumindest gleichgroßen Wertebereich wie `wchar_t` aufweist.

Anmerkungen zu Tabelle 9.3:

- Ein Ausdruck vom Typ `wchar_t` wird zum ersten möglichen Typ konvertiert, der alle Ausprägungen von `wchar_t` umfaßt.
- Bitfelder sind spezielle Klassen und werden in Kapitel 11 besprochen. Sie können nur dann in `int`- beziehungsweise `unsigned int`-Ausdrücke umgewandelt werden, wenn ihre Größe dies erlaubt.

Typ	wird zu
`char`, `signed char`, `unsigned char`	`int`
`short int`, `unsigned short int`	`int`
`wchar_t`	`int`, `unsigned int`, `long` oder `unsigned long`
`enum`	`int`, `unsigned int`, `long` oder `unsigned long`
`class` (Bitfelder)	`unsigned int`
`bool`	`int`

Tabelle 9.3: Integrale Promotion [ISO95, conv.prom]

Ein *RValue*-Ausdruck vom Typ `float` kann in einen Ausdruck vom Typ `double` konvertiert werden (Gleitkomma-Promotion).

Integral- und Gleitkomma-Typumwandlungen

Bei der integralen Typumwandlung werden entweder `int`-Ausdrücke in Ausdrücke eines entsprechenden `unsigned`-Typ umgewandelt oder umgekehrt. Im ersten Fall ist das Ergebnis der geringste `unsigned int`-Wert, der kongruent zum Ausgangswert ist (errechnet durch modulo 2 hoch *n*, wobei *n* die Anzahl der Bits ist, die den Datentyp bilden [ISO95, conv.integral]). Im Falle einer

Umwandlung von `signed`-Werten auf `unsigned`-Werte ist das Ergebnis undefiniert (genauer gesagt: compilerabhängig), wenn es im Zieltyp nicht dargestellt werden kann. `bool`-Werte werden, wie bereits angeführt in, `0` beziehungsweise `1` umgewandelt.

Gleitkomma-Typumwandlungen sind Typumwandlungen von Ausdrücken eines Gleitkommatyps in einen anderen. Sie sind dann definiert, wenn der ursprüngliche Wert *genau* im neuen Typ abgebildet werden kann. Andernfalls ist das Ergebnis implementierungsabhängig.

Gleitkomma-Integral-Typumwandlungen

Ein *RValue*-Ausdruck eines Gleitkommatyps kann in einen integralen Typ umgewandelt werden, wobei der Nachkommateil verloren geht. Kann der ursprüngliche Wert nicht als Integral dargestellt werden, so ist das Ergebnis implementierungsabhängig.

Umgekehrt kann ein *RValue*-Ausdruck eines integralen Typs in einen Gleitkomma-Ausdruck umgewandelt werden. Das Ergebnis ist die exakte Gleitkomma-Repräsentation beziehungsweise die nächst höhere oder niedrigere Gleitkommazahl (implementierungsabhängig).

```
float f = 1.234;
int i;
i = f;          // Gleitkomma -> Integral
i = 3.1415;
f = 1;          // Integral -> Gleitkomma
f = i;
```

Zeiger-Typumwandlungen

- Jeder integrale konstante Ausdruck, der null ergibt, kann auf den speziellen Zeigerwert `0`, den sogenannten Nullzeiger, umgewandelt werden.
- Jeder Zeiger, der auf einen Typ `T` zeigt, kann implizit in einen Zeiger auf den Typ `void` umgewandelt werden. Das Ergebnis entspricht einem Zeiger auf die Startadresse des jeweiligen Objekts.
- Im Zusammenhang mit Klassen und Vererbung gelten folgende Regeln (genaueres dazu in Kapitel 13):
  - Jeder Nullzeiger kann auf einen Nullzeiger vom Typ ,,Zeiger auf Klassen-Element`` konvertiert werden.
  - Jeder Zeiger auf eine Klasse kann implizit auf einen Zeiger zu seiner Basisklasse umgewandelt werden.
  - Analog zu oben kann jeder ,,Zeiger auf ein Klassen-Element`` auf einen ,,Zeiger auf ein Klassen-Element`` einer abgeleiteten Klasse umgewandelt werden.

Basisklassen-Typumwandlungen

Ein *RValue* vom Typ einer Klasse kann auf einen Ausdruck einer Basisklasse umgewandelt werden. Mehr dazu in Kapitel 13 im Zusammenhang mit Vererbung.

Bool-Typumwandlungen

Jeder *RValue* eines Ausdruck, der von integralem Typ, Enumerationstyp oder Zeigertyp ist, kann auf einen `bool`-Wert umgewandelt werden. Ein Wert `0` sowie jeder Nullzeiger wird zum `bool`-Wert `false`, jeder andere Wert zu `true`.

Vorige Seite: 9.4 Typumwandlungen **Eine Ebene höher:** 9.4 Typumwandlungen **Nächste Seite:** 9.4.2 Explizite Typumwandlung

Vorige Seite: 9.4.1 Standard-Typumwandlung **Eine Ebene höher:** 9.4 Typumwandlungen

Teilabschnitte

9.4.2 Explizite Typumwandlung

C++ kennt sechs verschiedene Typumwandlungsoperatoren (*Cast*-Operatoren). Ein *Cast*-Operator (`( )`) stammt noch aus dem C-Subset von C++, ein zweiter (ebenfalls `( )`) stellt eine alternative Notation dar, und die anderen vier (`const_cast`, `static_cast`, `reinterpret_cast` und `dynamic_cast`) sind erst seit kurzem Teil der Sprache.

Typumwandlung im C-Stil und im Funktionsstil

Wie [Str94] schreibt, ist die Typumwandlung mit dem C-*Cast*-Operator eine ,,Hammerschlag-Lösung``: ,,The C and C++ cast is a sledgehammer ``. Ein Ausdruck der Form `(T) expression` (C-Stil) beziehungsweise der Form `T( expression )` (Funktionsstil) wandelt `expression` in (fast) jedem Fall auf die eine oder andere Art in einen Ausdruck des Typs `T` um. Die Umwandlung kann dabei

- eine einfache Reinterpretation der bitweisen Darstellung des Ausdrucks sein,
- eine einfache arithmetische Größenanpassung umfassen,
- ein `const`- oder `volatile`-Attribut zu einem Ausdruck hinzufügen oder entfernen oder
- eine andere (eventuell implementierungsabhängige) Umwandlung vornehmen.

Aus dem Source-Code geht nicht hervor, welche der oben angeführten Typumwandlungen der Programmierer verwenden wollte. Mehr noch, bei der Änderung von Basisklassenhierarchien kann es passieren, daß die Bedeutung von *Casts* implizit verändert wird [Str94, S. 328].

Neben diesen Nachteilen ist die Notation mittels Klammersymbol zumindest unklar (Stroustrup beschreibt sie elegant als ,,close to minimal``).

Beispiele für Typumwandlungen mit dem *Cast*-Operator `()`:

```
class X {
  int   i;
  bool  b;
};
int   i = 7;
long  l;
char* str = "Hallo";
X     o;
int*  p1;
void* p2;

i = (int)true;    // Cast bool-int, C-Stil
l = long(i);      // Cast im Funktionsstil
p2 = (void*)o;
p1 = (int*)str;   // str als int-Zeiger
X(*p1).b = false; // *p1 als X, unsichere Umwandlung!
```

Die Probleme mit den ,,alten`` *Cast*-Operatoren sind also offensichtlich und brachten schließlich die Einführung neuer *Cast*-Operatoren, die diese Probleme zumindest zum Teil umgehen. Die Verwendung der ,,alten`` Typumwandlungsoperatoren wird nicht mehr empfohlen - sie sind in der Sprache aufgrund der nötigen Kompatibilität zu ,,alten`` Programmen aber weiterhin vorhanden.

Die ,,alten`` Typumwandlungsoperatoren sind nur mehr aus Kompatibilitätsgründen in der Sprache vorhanden und sollten *nicht* verwendet werden.

Neue *Cast*-Operatoren

Bei der Einführung von neuen, sicheren *Cast*-Operatoren wurde darauf geachtet, mehrere Operatoren für die verschiedenen Arten von Typumwandlungen zu verwenden.

Eine Typumwandlung mit dem Operator `const_cast` wandelt einen Ausdruck vom Typ `T` mit den optionalen Qualifikatoren `const` und `volatile` in einen Ausdruck desselben (Basis-)Typs ohne den Qualifikator `const` um. Der Sinn und Zweck eines *Casts* der Form `const_cast<T1>` ist *ausschließlich* die (vorübergehende) ,,Entfernung`` des `const`-Qualifikators. Dazu ein Beispiel:

```
const char* findSubString(const char* str,
                          const char* subStr)
{
```

```
  return strstr(str, subStr); // Fehler! strstr hat
                              // "nur" char*-Argumente
}
```

Die Funktion `findSubString` im Beispiel oben sucht nach einem *SubString* `subStr` in der Zeichenkette `str` und verwendet dazu die C-Funktion `strstr`. Alle Argumente der Funktion sind vom Typ `const char*`, da keiner der Parameter verändert wird. Die Funktion `strstr` hat aber Parameter vom Typ `char*`. Der Compiler meldet beim Übersetzen der ersten `return`-Anweisung oben einen Fehler, da versucht wird, `const`-Objekte als Aktualparameter für nicht konstante Formalparameter zu verwenden.

Grundsätzlich sind drei Lösungen des Problems möglich:

- Eine Lösung des Problems besteht darin, die Parameter der Funktion `findSubString` nicht als `const` zu deklarieren. Das widerspricht aber der Richtlinie, Eingangsparameter als `const` zu deklarieren.

  Ein Benutzer der Funktion, der nur die Schnittstelle kennt, kann keine `const`-Objekte als Parameter übergeben und würde zudem annehmen, daß die Parameter verändert werden könnten.
- Eine andere Lösung liegt in der Verwendung von *Casts* im C-Stil. Dann ist allerdings nicht klar, was der Programmierer mit dem Typumwandlungsoperator bezweckt. Zudem kann ein derartiger *Cast* bei einer Änderung der Aktualparametern zu ungewünschten Fehlern führen.

  ```
  return strstr((char*)str, (char*)subStr);
  ```
- Wesentlich besser ist es, den `const`-Qualifikator der Argumente mit einem `const_cast` zu entfernen. Damit ist einerseits die Schnittstelle der Funktion `findSubString` weiterhin korrekt definiert, und andererseits die Absicht des Programmierers klar:

  ```
  return strstr(const_cast<char*>str,
                const_cast<char*>subStr);
  ```

Wichtig ist zu wissen, daß ein `const_cast` nicht auf ,,echte`` `const`-Objekte angewandt werden darf. Echte `const`-Objekte sind Objekte, die bei ihrer Deklaration als `const` vereinbart sind. Das Ergebnis eines derartigen *Casts* ist undefiniert, da eine Kennzeichnung eines Objekts als `const` dem Compiler mitteilt, daß dieses Objekt nicht verändert wird. Der Compiler kann damit spezielle Optimierungen durchführen. Werden derartige Objekte in non-`const`-Objekte umgewandelt, kann es zu Problemen kommen! Unvorhersehbare Fehlerfälle sind die Folge.

```
const char ch = 'a';
cout << const_cast<char>ch;  // Problematisch!
```

- Soll der `const`-Qualifikator eines vorübergehend als `const` vereinbarten Objekts entfernt werden, so ist der *Cast*-Operator `const_cast` zu verwenden. Er darf aber nicht dazu verwendet werden, ,,echte`` `const`-Objekte zu verändern.

Typumwandlungen, die vom Compiler zur Übersetzungszeit als gültig beurteilt werden können, werden als ,,definiert`` bezeichnet. Sie werden mit dem Operator `static_cast` verwirklicht. Definierte Typumwandlungen sind Umwandlungen von Objekten einer Klasse auf Objekte einer Basisklasse oder die Umwandlung mittels einer Umwandlungsfunktion (siehe Kapitel 13). Beispiele für derartige Typumwandlungen werden in den jeweiligen Abschnitten gegeben.

Alle Typumwandlungen, die nicht von `static_cast` und `const_cast` durchgeführt werden können, werden mittels `reinterpret_cast` verwirklicht. Der Name des Operators stammt daher, daß eine derartige Typumwandlung eine neue Interpretation der zugrundeliegenden Bitkette, die das Objekt darstellt, erfordert.

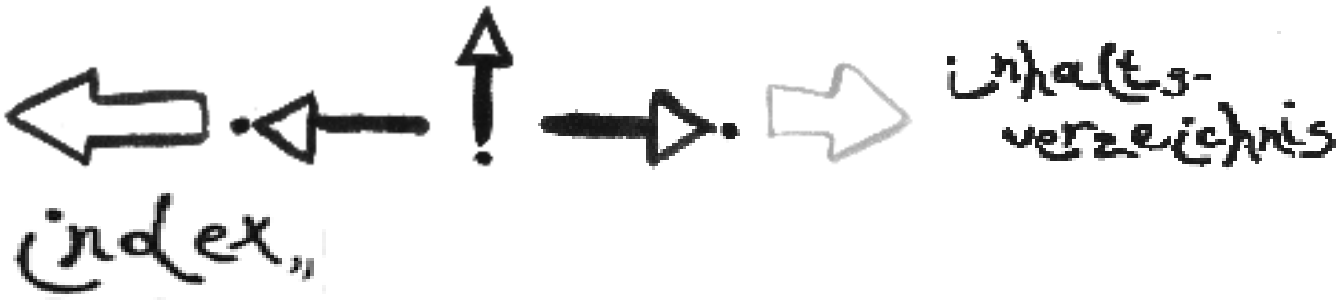
```
char*  p = new char[20];
...
int* pi = reinterpret_cast<int*>p;
```

Eine vierte Art der Typumwandlung ist die mit Hilfe von `dynamic_cast`. Diese Typumwandlung steht in engem Zusammenhang mit dem Typsystem von C++ (*Runtime Type Information System*, RTTI) und wird in dem Abschnitt über Klassen und Vererbung erläutert (Abschnitt 14.6).

Tabelle 9.4 zeigt, in welchen Situationen welche *Casts* zur Typumwandlung verwendet werden:

Tabelle 9.4: Einsatz der verschiedenen *Cast*-Operatoren

Situation	*Cast*-Operator
Vorübergehende Entfernung von `const`	`const_cast`
Umwandlung von polymorphen Objekten (zum Beispiel *Down-Casts*)	`dynamic_cast`
Definierte (= gültige) Umwandlung zwischen Speicherobjekten	`reinterpret_cast`
Undefinierte Umwandlungen (= alle anderen Situationen)	`static_cast`



Vorige Seite: 9.4.1 Standard-Typumwandlung **Eine Ebene höher:** 9.4 Typumwandlungen

Vorige Seite: Neue *Cast*-Operatoren **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** 10.1 Motivation

10 Module und Datenkapseln

Dieses Kapitel ist eine Art ,,Übergang`` zwischen den bisherigen, ,,traditionellen`` Elementen von C++ und den objektorientierten Elementen der nächsten Kapitel. Es stellt einen einfachen Ansatz der Modularisierung und Strukturierung von Programmen vor und ist wie folgt gegliedert:

- Eine Motivation führt kurz in die Thematik ein.
- Der folgende Abschnitt beschreibt das allgemeine Modulkonzept und spezielle Module, sogenannte Datenkapseln, in aller Kürze. Diese Module sind die Grundbausteine für unsere ,,einfachen`` Programme.
- Die Umsetzung in C++ wird als nächstes anhand eines konkreten Beispiels erläutert.
- Ein Resümee beschließt das Kapitel und damit den ,,traditionellen`` Teil des Buchs.

Vorige Seite: 10 Module und Datenkapseln **Eine Ebene höher:** 10 Module und Datenkapseln **Nächste Seite:** 10.2 Vom Modul zur

10.1 Motivation

Wie in Kapitel 2 gezeigt, bestehen einfache C++-Programme aus einer Textdatei, die übersetzt und anschließend gebunden wird. Im Fall von größeren Programmen führt dieser Ansatz aus mehreren Gründen nicht mehr zum Ziel:

- Arbeitsteilung
  Große Programme werden in der Regeln von mehreren Personen erstellt. Besteht das Programm aus einer einzelnen Datei, so würde dies zur Folge haben, daß alle Personen mit derselben Datei arbeiten müßten.
- Effizienz
  Jede Änderung einer Datei hat zur Folge, daß diese neu übersetzt werden muß. Im Fall einer großen Datei heißt das, daß jede Änderung eine vollständige Neuübersetzung des gesamten Programms nach sich zieht.
- Strukturierung
  Jedes große Programm kann logisch in verschiedene Bereiche aufgeteilt werden (zum Beispiel einen Teil mit Grafik-Funktionen, einen Teil, der das Dateihandling übernimmt, und so weiter). Eine physische Trennung von verschiedenen Bereichen kann erst dann vorgenommen werden, wenn Programme aus mehr als einer Datei bestehen.

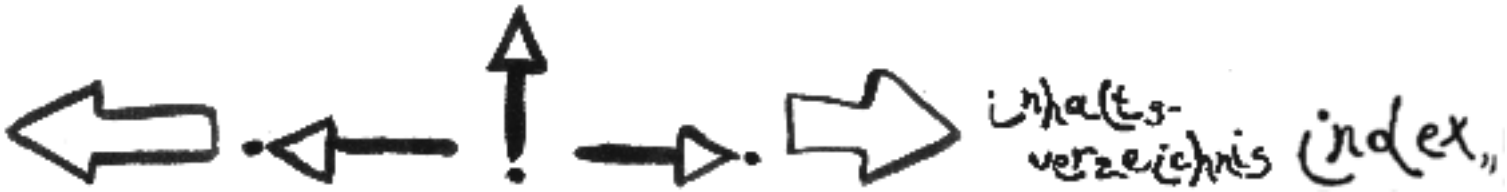
  Diese physische Strukturierung ist zudem Basis für die Erstellung von einzelnen wiederverwendbaren Programmteilen. So kann die gesamte Ansammlung von Grafikfunktionen eines Programms in Dateien abgelegt werden, die sowohl in Projekt A wie auch in den Projekten B und C genutzt werden.

Größere Programme bestehen daher aus einer Reihe von Dateien, die alle einzeln übersetzt werden und anschließend zu einem ausführbaren Programm gebunden werden.

Die einzelnen Dateien enthalten jeweils bestimmte Teile des gesamten Programms. Die Zerlegung des Programms in diese Teile ist ein wesentliches Problem des Programmentwurfs und nicht Teil dieses Buchs.

Aufgrund der Auswirkungen auf die Programmstruktur und die Implementierung der einzelnen Bausteine soll hier versucht werden, eine Strukturierungsmöglichkeit, die auf dem Modulprinzip beruht, grob vorzustellen und damit eine einfache ,,Richtlinie`` vorzugeben. Dabei bestehen C++-Programme aus der ,,Hauptfunktion`` `main` und einer Reihe von unabhängigen Programmbausteinen, die als Module

bezeichnet werden.



Vorige Seite: 10 Module und Datenkapseln **Eine Ebene höher:** 10 Module und Datenkapseln **Nächste Seite:** 10.2 Vom Modul zur

Vorige Seite: 10.1 Motivation **Eine Ebene höher:** 10 Module und Datenkapseln **Nächste Seite:** 10.3 Module und Datenkapseln

10.2 Vom Modul zur Datenkapsel

Ein Modul ist nach Goos [Goo73] eine Ansammlung von Daten und Operationen, die

- eine in sich abgeschlossene Aufgabe realisiert,
- mit ihrer Umgebung über eine definierte Schnittstelle kommuniziert,
- ohne Kenntnisse ihres inneren Verhaltens in ein Gesamtsystem integriert werden kann und
- deren Korrektheit ohne Kenntnis ihrer Einbettung in ein Gesamtsystem nachgewiesen werden kann.

Ein Modul ist damit ein Programmbaustein, der anders als eine Ansammlung von Funktionen sowohl Funktionen als auch Daten enthält. Seine Kommunikation erfolgt ausschließlich über die festgelegte Schnittstelle.

Die Schnittstelle beschreibt, *was* der Modul zur Verfügung stellt. Die Schnittstelle beschreibt aber nicht, *wie* dieses Verhalten konkret realisiert ist. Das *Wie* ist im Innern des Moduls verborgen.

Module realisieren damit das *Geheimnisprinzip* in bezug auf interne Daten und Operationen: Das Geheimnisprinzip fordert, daß die interne Konstruktion von Modulen in Form von Funktionen, Daten und Datenstrukturen vor der Umgebung verborgen bleiben soll, da das Programm diese Informationen nicht benötigt (nach [Sch92]).

Damit wird in der Informatik nachgebildet, was in anderen technischen Disziplinen längst üblich ist: Wie Komponenten intern realisiert sind, ist nicht von Interesse, wichtig ist, was sie tun. So durchschauen zum Beispiel lediglich Spezialisten die genaue Funktionsweise von Komponenten einer modernen Stereoanlage, die Benutzung hingegen ist einfach. Auch die Kombination von mehreren ,,Bausteinen`` (CD-Player, Tuner, Verstärker, Dat und so weiter) ist anhand der ,,Schnittstellen`` der Bausteine problemlos möglich.

Die Kenntnis von Modul-Internas ist auch für die Integration eines Moduls nicht nötig, da die Funktionalität ausschließlich über die Schnittstelle festgelegt ist. Damit ist auch die Korrektheit eines Moduls nachweisbar, indem die Korrektheit der exportierten Schnittstelle nachgewiesen wird.

Module stellen einen wesentlichen Schritt im Software Engineering dar, indem sie unabhängige Bausteine zur Basis von Programmen machen. Um die einzelnen Programmbausteine so einfach benutzbar und unabhängig als möglich zu gestalten, wird aber oft noch einen Schritt weiter gegangen: Module exportieren keine Daten, sondern verbergen sie im Innern. Derartige Module werden auch als

Datenkapseln bezeichnet.

Wesentlicher Vorteil von Datenkapseln ist, daß Benutzer auf diese Bausteine ausschließlich über die bereitgestellten Funktionen zugreifen können. Die Schnittstelle kann sehr einfach gehalten werden und bewirkt, daß die Bausteine insgesamt einfacher zu testen sind: Zeigt man die Korrektheit dieser Funktionen, so kann man davon ausgehen, daß der Baustein korrekt arbeitet. Zudem wirken sich Änderungen von ,,Internas`` nicht auf die Klienten einer Datenkapsel aus. Die Abhängigkeit zwischen den einzelnen Modulen bleibt damit auf die Schnittstelle beschränkt.

Die Effizienzeinbußen, die sich durch den ,,indirekten`` Zugriff auf die Daten ergeben, sind dagegen in den meisten Fällen vernachläßigbar.

Vorige Seite: 10.1 Motivation **Eine Ebene höher:** 10 Module und Datenkapseln **Nächste Seite:** 10.3 Module und Datenkapseln

Vorige Seite: 10.2 Vom Modul zur **Eine Ebene höher:** 10 Module und Datenkapseln **Nächste Seite:** 10.3.1 Die Schnittstellendatei

---

10.3 Module und Datenkapseln in C++

In C++ sind zwei Dateien nötig, um ein Modul zu implementieren. Die erste Datei stellt die ,,öffentliche`` Schnittstelle des Moduls dar. In ihr sind alle Datentypen, Konstanten und Funktionen des Moduls *deklariert*, die für Klienten zur Verfügung gestellt werden. Die in der Schnittstelle angeführten Funktionen werden in einer zweiten Datei *definiert*. In dieser Datei sind zudem alle Hilfsfunktionen und internen Daten abgelegt.

Zur Illustration wird ein Modul `Stack` verwendet. Ein `Stack` ist eine Datenstruktur, die Elemente nach Art eines Bücherstapels ,,zwischenspeichert``. Die einzelnen Elemente werden auf einen imaginären Stapel abgelegt. Das jeweils oberste Element kann wieder abgerufen werden.

`Stack` verfügt über die Zugriffsfunktionen `stackInit`, `stackPush`, `stackPop`, `stackPeek` `stackIsEmpty` und `stackWasError`. `stackPush` legt ein Element im `Stack` ab, `stackPop` nimmt das oberste Element vom `Stack` und gibt es zurück, `stackPeek` retourniert das oberste Element ohne es vom Stack zu nehmen. `stackIsEmpty` gibt an, ob ein Element im `Stack` enthalten ist und `stackWasError` gibt an, ob die letzte Operation fehlerhaft war. Fehler können zum Beispiel auftreten, wenn die Funktion `stackPop` aufgerufen wird, obwohl kein Element vorhanden ist.

Vorige Seite: 10.2 Vom Modul zur **Eine Ebene höher:** 10 Module und Datenkapseln **Nächste Seite:** 10.3.1 Die Schnittstellendatei

Vorige Seite: 10.3 Module und Datenkapseln **Eine Ebene höher:** 10.3 Module und Datenkapseln **Nächste Seite:** 10.3.2 Die Implementierungsdatei

10.3.1 Die Schnittstellendatei

Die Schnittstelle des Moduls wird in einer Datei, die auf `.h` endet, abgelegt (`stack.h`). Schnittstellendateien enthalten nach [HN93] folgende Programmelemente:

- Kommentare, die die Modul-Schnittstelle erklären und verschiedene Informationen zum Modul enthalten (Angaben über den Autor, das Datum der Erstellung, der letzten Änderung).
- Schnittstellen von Klassen und Funktionen
- Alle exportierten (= von aussen sichtbaren) Typen, Konstanten
- Präprozessor-Anweisungen, die ein mehrfaches Inkludieren verhindern. Das Problem von Mehrfach-*Includes* tritt auf, wenn eine `.h`-Datei mehrmals in ein Programm eingefügt wird. Die wiederholten Deklarationen führen dann zu Übersetzungsfehler (mehr dazu später in Abschnitt B.1). Um derartige Probleme von vornherein auszuschließen, werden folgende Richtlinien angegeben:
  - Jede .h-Datei enthält als erste Anweisungsfolge ein Reihe von Makros, die das Mehrfacheinfügen verhindern.

    ```
    #ifndef <DATEINAME>__H
    #define <DATEINAME>__H
    ```

    Der Name des Makros ist willkürlich gewählt. Aufgrund der nötigen systemweiten Eindeutigkeit ist es jedoch naheliegend, das Makro unter anderem aus dem Dateinamen zusammenzusetzen.
  - Die Datei wird durch die folgende Anweisung abgeschlossen:

    ```
    #endif
    ```

Schnittstellen enthalten *keine* Variablen und keinen Code. Ausnahmen von dieser Regel sind spezielle Funktionen (zum Beispiel *Access*-Funktionen, siehe Kapitel 11).

Das führt zur folgenden Datei `stack.h`:

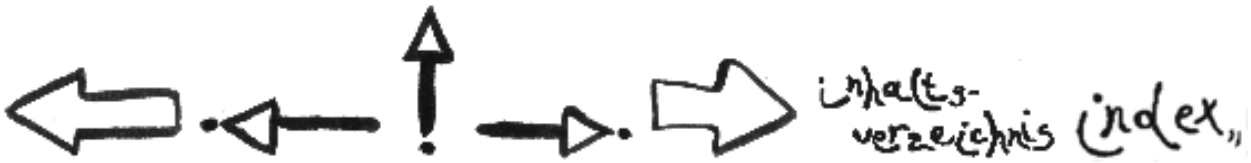
Programm 10.1: Modul `Stack`, Schnittstelle

```
  // Datei stack.h, allg. Kommentare (Ersteller, Datum)

  // Schnittstelle des Moduls Stack, enthaelt "Prototypen"
  // aller vereinbarten Objekte
#ifndef Stack__H
#define Stack__H

  // Funktionen fuer die Verwaltung d. Stacks

  // Init initialisiert den Stack (loescht alle Elemente)
void stackInit();

  // Push legt ein int-Element auf den Stack
void stackPush(int elem);

  // Pop liefert das oberste Element des Stacks
  // (falls vorhanden) und entfernt es vom Stack
```

```
int stackPop();

  // Peek liefert das oberste Element des Stacks
  // (falls vorhanden)
int stackPeek();

  // IsEmpty gibt an, ob der Stack leer ist
bool stackIsEmpty();

  // WasError gibt an, ob die letzte
  // Operation feherhaft war
bool stackWasError();

#endif
```

Schnittstellendateien werden nur übersetzt, um ihre syntaktische Korrektheit zu überprüfen. Sie enthalten keinen Code und erzeugen daher auch kein Programm. Ihre einzige Aufgabe ist es, Vereinbarungen zur Verfügung zu stellen.

Vorige Seite: 10.3 Module und Datenkapseln **Eine Ebene höher:** 10.3 Module und Datenkapseln **Nächste Seite:** 10.3.2 Die Implementierungsdatei



Vorige Seite: 10.3.1 Die Schnittstellendatei **Eine Ebene höher:** 10.3 Module und Datenkapseln **Nächste Seite:** 10.3.3 Modul-Klienten

10.3.2 Die Implementierungsdatei

Die Implementierungsdatei eines Moduls (beziehungsweise einer Klasse) endet üblicherweise auf `.cpp` (DOS, Windows, OS/2 etc.) oder auf `.C` (Unix).

Sie enthält folgende Programmelemente:

- Einen einleitenden Modul-Kommentar
- Verschiedene `include`-Anweisungen, mit denen die Schnittstellen aller verwendeten Module/Bibliotheken eingefügt werden. Auch die ,,eigene`` Schnittstelle ist einzufügen, damit die dort getroffenen Vereinbarungen auch in der Implementierungsdatei bekannt sind.
- Lokale Objekte (Konstanten, Typen, Variablen), die nicht von außen zugänglich sind. Diese Objekte sollen nur innerhalb der aktuellen Datei Gültigkeit haben und werden deshalb in einem eigenen Namensraum deklariert. Insbesonders werden lokale Variablen *nicht* als `static` vereinbart (vergleiche dazu Abschnitt 9.1.2).
- Die Implementierung aller in der Schnittstelle deklarierten Funktionen oder Klassen.
- Gegebenfalls weitere interne Funktionen oder Klassen.

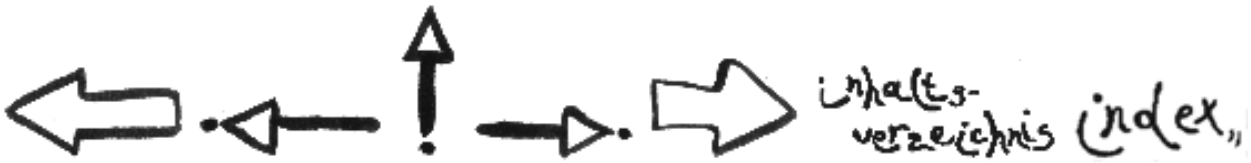
Die Implementierungsdatei `stack.cpp` sieht damit wie folgt aus:

Programm 10.2: Modul `Stack`, Implementierung

```
  // File: stack.cpp. Implementierung des Moduls Stack
  // Einfuegen der eigenen Schnittstelle
#include "stack.h"

  // Vereinbarung der globalen Variablen des Moduls (intern!)
namespace {
  const int MaxElems=30;  // Vereinbarung einer Konstante
                          // fuer Stackgroesse
  int  elems[MaxElems];   // Vektor von int-Zahlen
  int  top;               // Index des ersten freien Elements
  bool errOccd;           // Gibt an, dass ein Fehler auftrat
}

void stackInit()
{
  top = 0; errOccd = false;
}

void stackPush(int elem)
{
  errOccd = (top==MaxElems);  // Ueberlauf?
  if (!errOccd) {
    elems[top] = elem;
    ++top;
  }
```

```
}

int stackPop()
{
  errOccd = stackIsEmpty();   // Unterlauf?
  if (!errOccd) {
    --top;
    return elems[top];
  } else {
    return 0;
  }
}

int stackPeek()
{
  errOccd = (top==0);
  if (!errOccd) {
    return elems[top-1];
  } else {
    return 0;
  }
}

bool stackIsEmpty()
{
  return (top==0);
}

bool stackWasError()
{
  return errOccd;
}
```

Implementierungsdateien enthalten Code und müssen daher auch übersetzt werden.



Vorige Seite: 10.3.2 Die Implementierungsdatei **Eine Ebene höher:** 10.3 Module und Datenkapseln **Nächste Seite:** 10.3.4 Einige Kommentare zu `Stack`

10.3.3 Modul-Klienten

Als Klienten eines Moduls werden all jene Programmteile bezeichnet, die einen Modul benutzen. Klienten eines Moduls inkludieren die Schnittstellendatei des Moduls und können alle dort enthaltenen Programmelemente verwenden.

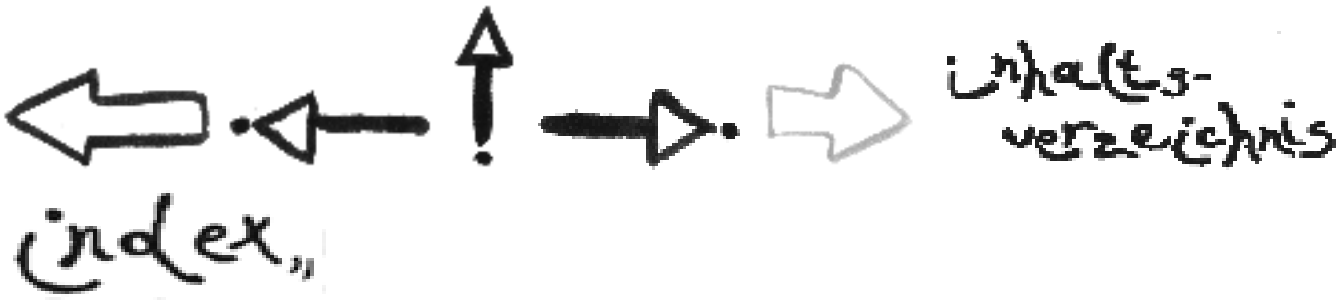
Im folgenden ist auszugsweise ein kleines Beispiel für einen Klienten des Moduls `Stack` angeführt. Der Klient inkludiert zunächst die Dateien `iostream` und `stack.h`. Damit „erhält`` er alle Deklarationen für die Ein-/Ausgabe und des Moduls `Stack` und kann diese verwenden:

Programm 10.3: Hauptprogramm `Stacktest`

```
  //File: stacktst.cpp
  // Testprogramm fuer das Modul Stack

  // Schnittstelle von iostream inkludieren
  // iostream ist ein vordefiniertes Modul, daher wird
  // der Name mit den Zeichen <, > begrenzt
#include <iostream>

  // Schnittstelle von Stack inkludieren
  // Stack ist ein selbsterstelltes Modul, daher wird
  // der Name mit Anfuehrungszeichen begrenzt
#include "stack.h"

void main()
{
  char          ch;
  int           i;

  stackInit();
  cout << "\n\nOperation (Quit, pUsh, pOp, pEek, Isempty) ";
  cin >> ch;

  while (ch != 'Q') {
    switch (ch) {
     case 'U':
      cout << "\n Element fuer push? ";
      cin >> i;
      stackPush(i);
      if (stackWasError()) {
        cout << " Fehler!";
      }
      break;
     case 'O':
      i = stackPop();
      if (stackWasError()) {
```

```
        cout << " Fehler!";
      } else {
        cout << "\n pop-Element: "<< i;
      }
      break;
     case 'E':
      i = stackPeek();
      if (stackWasError()) {
        cout << " Error!";
      } else {
        cout << "\n Oberstes Stack-Element "<< i;
      }
      break;
     case 'I':
      if (stackIsEmpty()) {
        cout << "\n Stack ist leer.";
      } else {
        cout << "\n Stack enthaelt Elemente.";
      }
      break;
     default:
      cout << "\n Falsche Operation.";
    }

    cout << "\n\nOperation (Quit, pUsh, pOp, pEek, Isempty) ";
    cin >> ch;
  }
}
```

Anmerkung: Testprogramme wie das oben angeführte werden auch oft *interaktive Testtreiber* genannt und häufig zum Testen von Modulen eingesetzt. Sie sind sehr einfach aufgebaut und stellen dem Benutzer die Auswahl der Testfälle frei.



Vorige Seite: 10.3.3 Modul-Klienten **Eine Ebene höher:** 10.3 Module und Datenkapseln

10.3.4 Einige Kommentare zu `Stack`

Die beiden Dateien `stack.h` und `stack.cpp` bilden eine Datenkapsel: Die Schnittstelle enthält keine Daten, die Funktionalität wird über Funktionen realisiert. Alle Daten sind intern im Modul verborgen und können von aussen nicht eingesehen werden. Mehr noch, die nicht sichtbare Implementierung von `Stack` führt dazu, daß Internas jederzeit abgeändert werden können, ohne daß Klienten (= Programme, die den `Stack` benutzen) davon tangiert wären. Erst wenn die Schnittstelle geändert wird, sind Klienten betroffen.

Das führt zum Begriff *Abhängigkeit* : Zwischen den einzelnen Dateien eines Programmsystems besteht eine Abhängigkeit: Werden in einer Datei `A` Elemente einer Datei `B` verwendet, so spricht man von einer ,,Benutzt``-Beziehung zwischen `A` und `B`.

In unserem Beispiel ist `stack.cpp` von `stack.h` abhängig und `stacktst.cpp` ebenfalls von `stack.h` (siehe Abbildung 10.1). Diese Abhängigkeit ist wesentlich für den Entwicklungsprozeß: jedesmal wenn `stack.h` verändert wird, müssen alle abhängigen Dateien ebenfalls neu übersetzt werden. Moderne Entwicklungssysteme erledigen dies in der Projektverwaltung automatisch. Sie erkennen, welche Dateien von welchen abhängig sind, und übersetzen automatisch alle ,,ungültigen``.

In einigen Entwicklungsumgebungen muß dieses Verhalten mit speziellen Programmen (sogenannte `make`-*Utilities*) nachgebildet werden. Bei kleinen Projekten (wie in diesem Beispiel) kann dies auch noch von Hand erfolgen.

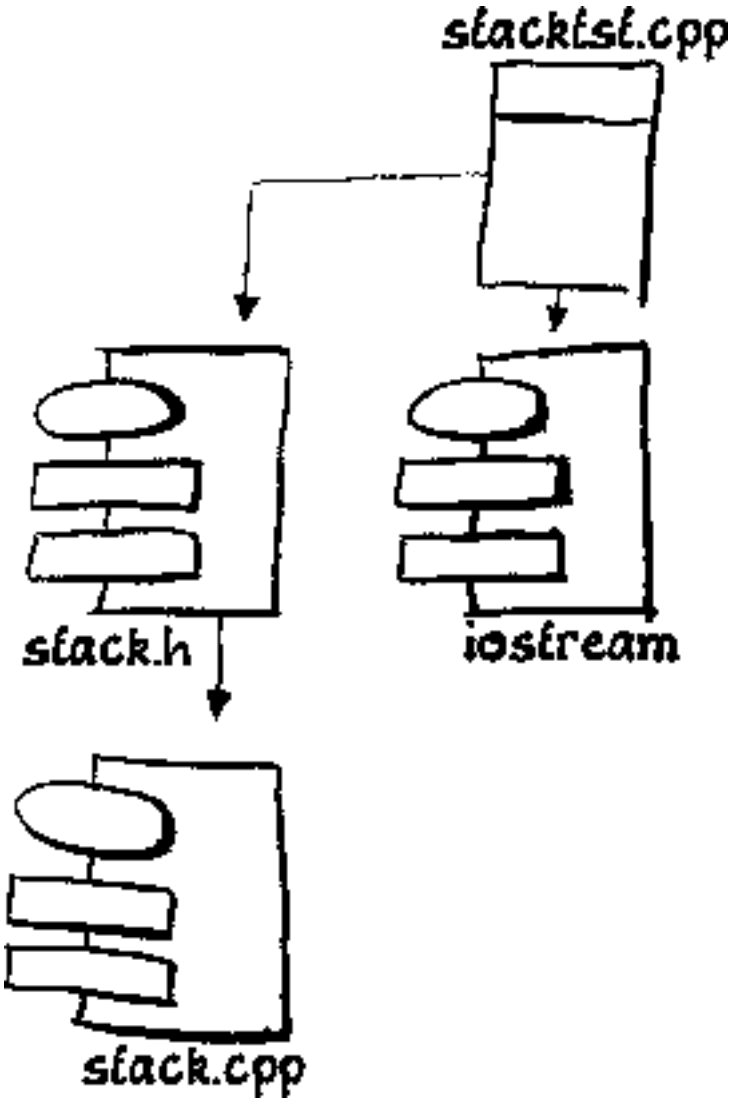


Abbildung 10.1: Abhängigkeit zwischen den `Stack`-Komponenten

Als logische Abhängigkeit wird die Abhängigkeit zwischen den einzelnen Bausteinen, abstrahiert von den jeweiligen Dateien, bezeichnet. In unserem Beispiel ist `main` von `iostream` und von `stack.h` abhängig. `stack.h` ist ebenfalls von `iostream` abhängig.

Die Korrektheit von `Stack` ist nachweisbar, indem sämtliche exportierten Funktionen auf ihre Korrektheit überprüft werden. `Stack` kann auch als relativ sicher gelten, da durch das Verbergen von allen Daten im Innern eine fehlerhafte Verwendung nahezu ausgeschlossen wird.

Vorige Seite: 10.3.3 Modul-Klienten **Eine Ebene höher:** 10.3 Module und Datenkapseln *(c)* *Thomas*

*Strasser, **d**punkt 1997*

Vorige Seite: 10.3.4 Einige Kommentare zu `Stack` **Eine Ebene höher:** 10 Module und Datenkapseln
Nächste Seite: 11 Das Klassenkonzept

10.4 Resümee

Nach dem hier vorgestellten Schema lassen sich auch größere Programme strukturieren. Erst wird dazu die Aufgabe in Teilaufgaben zerlegt, die durch möglichst unabhängige Module repräsentiert werden. Die einzelnen Module übernehmen in sich abgeschlossene Aufgaben und können zu einem gewissen Teil unabhängig voneinander und arbeitsteilig entwickelt werden.

Die ,,Unabhängigkeit`` von Modulen gibt an, wie stark Module eines Programmsystems miteinander ,,verflochten`` sind. Je höher die Verflechtungen, desto schwerer können einzelne Module getestet oder für andere Aufgaben eingesetzt werden. Man spricht in diesem Zusammenhang auch von Kopplung.

Demgegenüber wird mit dem Begriff ,,Bindung`` der ,,innere Zusammenhalt`` eines Moduls beschrieben. Module, die zufällige Ansammlungen von Funktionen und Daten sind, haben eine sehr niedrige Bindung. Datenkapseln enthalten Funktionen, die alle Aufgabe an derselben Datenstruktur verrichten und haben deshalb eine hohe Bindung.

Das Ziel beim Entwurf von Programmsystemen ist es, Module mit möglichst hoher Bindung und zugleich möglichst geringer Kopplung zu erhalten. Das Ergebnis ist ein System, dessen einzelne Bausteine einen großen Zusammenhalt haben und dennoch möglichst unabhängig voneinander sind.

Der Entwurfsprozeß ist zwar nicht Teil des vorliegenden Buchs, dafür wird auf entsprechende Literatur (zum Beispiel [Boo87, PB96]) verwiesen. Weniger wichtig ist dabei aber die Erstellung von speziellen Modulen. Schon in Kapitel 11 wird dieses Konzept durch das Klassenkonzept abgelöst. Wesentlich ist vielmehr die Aufteilung eines Programms in einzelne ,,Bausteine`` und die konsequente Anwendung des weiterhin gültigen Geheimnisprinzips.

Vorige Seite: 10.3.4 Einige Kommentare zu `Stack` **Eine Ebene höher:** 10 Module und Datenkapseln
Nächste Seite: 11 Das Klassenkonzept

Vorige Seite: 10.4 Resümee **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** 11.1 Motivation

---

11 Das Klassenkonzept

Strukturierte Datentypen in Form von Klassen sind das zentrale Konzept von C++. Das gesamte Kapitel ist diesem Konzept gewidmet:

- Nach einer Motivation werden Element-Funktionen im allgemeinen näher erläutert.
- Der nächste Abschnitt beschäftigt sich mit den verschiedenen Aspekten des Zugriffsschutzes im Zusammenhang mit Klassen.
- Dann werden das Konzept der statischen Klassen-Elemente sowie geschachtelte Klassen (Klassen, die in anderen Klassen deklariert sind) erläutert.
- Im Anschluß daran erfolgt die detaillierte Besprechung von speziellen Element-Funktionen und Operatoren anhand eines durchgängigen Beispiels.
- Den Abschluß bildet ein kurzer Überblick über spezielle Themen im Bereich des Klassenkonzepts.

Vorige Seite: 11 Das Klassenkonzept **Eine Ebene höher:** 11 Das Klassenkonzept **Nächste Seite:** 11.1.1 Klassen als Mittel

11.1 Motivation

Die beiden folgenden Abschnitte geben eine kurze Einführung in das Klassenkonzept in C++. Sie stellen die Motivation für die ,,Einheit von Daten und Methoden`` dar und zeigen die Umsetzung des Geheimnisprinzips in bezug auf Klassen in C++.

Vorige Seite: 11.1 Motivation **Eine Ebene höher:** 11.1 Motivation **Nächste Seite:** 11.1.2 Klassen und das

11.1.1 Klassen als Mittel zur Abstraktion

Abstraktion ist eine Technik, die der Mensch zum Verständnis und zur Lösung von komplexen Problemen einsetzt. Da es unmöglich ist, alle Aspekte eines umfangreichen Problems auf einmal zu erfassen, werden die für eine Aufgabe weniger wichtigen Details ignoriert und die (für die Aufgabe) wesentlichen Eigenschaften hervorgehoben. Man bildet sozusagen ein ,,idealisiertes`` Modell, in dem die ursprüngliche Aufgabe gelöst wird.

Das Prinzip der Abstraktion ist ein Schlüsselkonzept der Informatik. Durch Abstraktion werden charakteristische Eigenschaften eines Objekts und sein Verhalten modelliert. In der Programmiersprache C++ sind Klassen der primäre Mechanismus, um Abstraktionen zu bilden.

Wie bereits in Abschnitt 8.6 besprochen, können Klassen in der einfachsten Form als eine Art Strukturen oder Records, also eine Zusammenfassung von Datenelementen zu einer größeren Einheit, bezeichnet werden. Im ersten und einfachsten Beispiel wollen wir also eine derartige Klasse vereinbaren. Wir verwenden dazu das Beispiel `Stack` aus Kapitel 10.3.

Die Daten `elems`, `top` und `errOccd` können mit dem Schlüsselwort `class` in einer ,,Struktur`` zusammengefaßt werden und bilden die Datenstruktur für `Stack`. Das unten angeführte Codestück vereinbart eine Klasse mit den Komponenten `MaxElems`, `elems`, `top` und `errOccd`.

```
class Stack
{
  const int MaxElems = 10;

  int       elems[MaxElems];
  int       top;
  bool      errOccd;
};
```

Eine Variable vom Typ der Klasse `Stack` kann ebenso wie jede andere Variable eines beliebigen Typs vereinbart werden, indem Typname (= Klassenname) und Variablenname in einer Deklaration angeführt werden. Variablen eines Klassentyps werden *Objekte* (im objektorientierten Sinne) genannt.

Anmerkung: Bisher wurden verschiedene Elemente (Variablen) von C++ ganz allgemein als *Objekte* bezeichnet. Ein *Objekt* im Sinne der objektorientierten Programmierung (*OOP*) ist allerdings ausschließlich eine Variable eines Klassentyps.

```
Stack  myStack;   // Stack-Objekt
Stack* stackPtr;  // Zeiger auf ein Stack-Objekt
```

Im obigen Beispiel werden ein Objekt `myStack` sowie ein Zeiger auf ein Objekt der Klasse `Stack` vereinbart. Diese Objekte können wie „normale`` Variablen zum Beispiel zugewiesen, verglichen und kopiert werden.

Wird die Klasse `Stack` wie oben vereinbart, so können wir allerdings nicht auf die einzelnen Elemente zugreifen. Der Grund dafür liegt darin, daß grundsätzlich alle Elemente einer Klasse geheim (`private`) sind. Auf geheime Elemente kann von außen nicht zugegriffen werden. Um die Elemente von außen zugänglich zu machen, müssen wir das Schlüsselwort `public` benutzen:

```
class Stack
{
 public:
  const int MaxElems = 10;

  int    elems[MaxElems];
  int    top;
  bool   errOccd;
};
```

Nun können wir von außen auf die einzelnen Elemente wie bei einer Struktur zugreifen:

```
Stack  myStack;   // ein Stack-Objekt
Stack* stackPtr;  // ein Zeiger auf ein Stack-Objekt

myStack.top = 0;        // Zugriff ueber den .-Operator
stackPtr = &myStack;
StackPtr->errOccd = 0;  // Zugriff ueber einen Zeiger
```

Soweit besteht kein wesentlicher Unterschied zwischen Klassen und „herkömmlichen`` Strukturen oder `RECORD` wie wir sie zum Beispiel von C, Pascal oder Modula2 kennen. Klassen sind allerdings wesentlich mehr. Sie stellen ein Konstrukt dar, das es uns erlaubt, Daten *und* zugehörige Operationen zu *einer einzelnen Einheit* zusammenzufassen.

In unserem Fall sind die zugehörigen Operationen `init`, `push`, `pop`, `peek`, `isEmpty` und `wasError`. Diese Operationen sind inhärent mit der Klasse `Stack` verbunden, das heißt, jede der Operationen kann ausschließlich auf ein Objekt dieser Klasse angewandt werden. Was liegt daher näher, als die Operationen *in der Klasse* zu vereinbaren, anstatt sie (wie in Abschnitt 10.3) als eigenständige Funktionen zu codieren?

```
class Stack
{
 public:
    // Hier folgen die Methoden der Klasse
  void init()
  {
    top = 0;
```

```
    errOccd = 0;
  };
  void push(int elem)
  {
    ... // Implementierung von push
  };
  int pop()
  {
    ... // Implementierung von pop
  };
  int peek()
  {
    ... // Implementierung von pop
  };
  bool isEmpty()
  {
    ... // Implementierung von isEmpty
  };
  bool wasError()
  {
    ... // Implementierung von wasError
  };

    // Daten:
  const int MaxElems = 10;
  int  elems[MaxElems];
  int  top;
  bool errOccd;
};
```

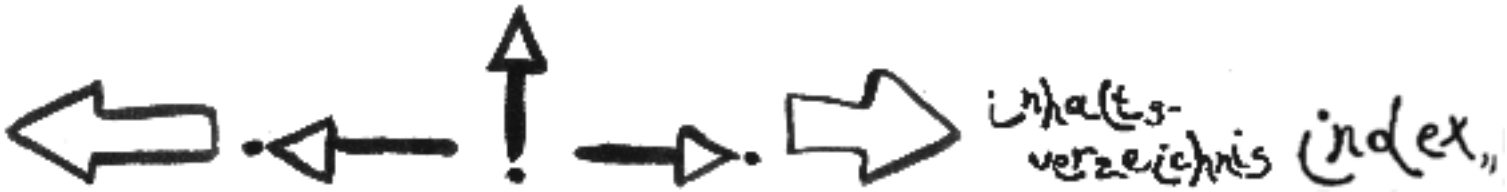
Zwei Bemerkungen zu den Operationen von Klassen:

- Operationen einer Klasse (= Funktionen, die im Klassenrumpf definiert sind) werden als *Element-Funktionen* oder *Methoden* bezeichnet.

- Üblicherweise beginnen Element-Funktionen mit einem Kleinbuchstaben und werden in *MixedCase* notiert.

Einige einfache Beispiele für die Verwendung der Klasse `Stack`:

```
Stack   myStack;   // ein Stack-Objekt
Stack*  stackPtr;  // ein Zeiger auf ein Stack-Objekt

myStack.init();    // Aufruf von init ueber .-Operator
myStack.push(910);
myStack.push(25);
cout << myStack.pop();

stackPtr = &myStack;
cout << StackPtr->pop();  // Zugriff ueber einen Zeiger
```



Vorige Seite: 11.1 Motivation **Eine Ebene höher:** 11.1 Motivation **Nächste Seite:** 11.1.2 Klassen und das

Vorige Seite: 11.1.1 Klassen als Mittel **Eine Ebene höher:** 11.1 Motivation

11.1.2 Klassen und das Geheimnisprinzip

Die Klasse `Stack` im Beispiel oben exportiert sowohl Daten als auch Methoden („exportiert`` werden die als `public` vereinbarten Elemente). Der Export von Daten ist in bezug auf mehrere Aspekte problematisch:

- Klienten (= Benutzer der Klasse) können auf Klasseninternas zugreifen und die vorgefertigten Zugriffsmethoden „umgehen``. Die Datenelemente können von Klienten direkt manipuliert werden.
- Mögliche Zugriffe von Klienten auf „Klasseninternas`` führen in weiterer Folge zu einer stärkeren Abhängigkeit zwischen der Klasse und ihren Klienten.

  Auf Klienten wirken sich natürlich nur Änderungen des für sie sichtbaren Teils (der Schnittstelle) aus. Werden nur die „nötigen`` Funktionalitäten exportiert und alle Daten und Hilfsfunktionen geheim gehalten, so wird diese Abhängigkeit minimiert.
- Vereinfacht gesagt, werden beim Testen von Komponenten die verschiedenen Möglichkeiten ihrer Verwendung auf Korrektheit hin überprüft. Die Korrektheit von Komponenten, die ausschließlich Methoden (und keine Daten) exportieren, kann gezeigt werden, indem die Korrektheit aller exportierten Methoden gezeigt wird. Damit wird das Testen wesentlich vereinfacht und der nötige Aufwand verringert. Werden auch Daten exportiert, so erweist sich ein sinnvoller Test als sehr viel schwieriger.

Um diese „Mißstände`` zu beseitigen, müssen wir die Daten der Klasse `Stack` vor „unbefugtem`` Zugriff schützen. Dies erfolgt durch die Angabe des Schlüsselworts `private`.

```
class Stack
{
   // public-Sektion, alle hier angefuehrten Elemente
   // sind von aussen zugaenglich
 public:
    // Hier folgen die Methoden der Klasse
  void init()
  {
    top = 0;
    errOccd = 0;
  };

  void push(int elem)
  {
    ... // Implementierung von push
  };
  int pop()
  {
    ... // Implementierung von pop
  };
  int peek()
  {
    ... // Implementierung von peek
  };
  bool isEmpty()
```

```
  {
    ... // Implementierung von isEmpty
  };
  bool wasError()
  {
    ... // Implementierung von wasError
  };

   // private-Sektion, alle hier angefuehrten
   // Elemente sind "geheim" und koennen nur intern
   // (= in Stack) verwendet werden
 private:
  const int MaxElems = 10;
  int    elems[MaxElems];
  int    top;
  bool   errOccd;
};
```

Nun kann von außen zwar auf alle Methoden, allerdings nicht mehr auf die in der `private`-Sektion angeführten Daten der Klasse zugegriffen werden (`elems`, `top`, `errOccd`). Jeder (versuchte) Zugriff führt bereits beim Übersetzen zu einem Fehler.

Allerdings weist die oben angeführte Klasse noch einen wesentlichen ,,Nachteil`` auf: Die Vereinbarung der Methoden läßt die Klassendefinition sehr umfangreich und unübersichtlich erscheinen und gibt zudem implementierungsspezifische Details bekannt. Daher wird die sogenannte ,,Klassenschnittstelle`` (Definition einer Klasse *ohne* die Implementierung der Methoden) von der eigentlichen Implementierung (= Implementierung der Methoden) getrennt.

Die Schnittstelle stellt die eigentliche Abstraktion dar und definiert Struktur und Verhalten von Klassen (also das *WAS*). Wie dieses Verhalten realisiert wird (das *WIE*), ist in der Implementierung festgelegt. Die Schnittstelle enthält in der Regel keinen Code (Ausnahmen davon sind zum Beispiel sogenannte *Access Functions*, mehr dazu in Abschnitt 11.2).

Ähnlich der Zerlegung in `DEFINITION` und `IMPLEMENTATION MODULE` in Modula2 werden Schnittstelle und Implementierung üblicherweise in verschiedenen Dateien vereinbart (vergleiche dazu auch das Beispiel in Kapitel 10).

Die Aufteilung auf zwei Dateien ist allerdings eine Konvention und in der Programmiersprache C++ nicht zwingend vorgesehen. Es ist durchaus möglich, sehr einfache Klassen (zum Beispiel abstrakte Klassen, siehe Kapitel 14) in einer einzelnen Datei zu notieren. Andererseits kann die Implementierung sehr komplexer Klassen auch auf mehrere Dateien aufgeteilt werden.

Die ,,endgültige`` Schnittstelle der Klasse `Stack` sieht damit wie folgt aus:

Programm 11.1: Klasse `Stack`, Schnittstelle

```
  // Datei stack.h, allg. Kommentare hier
#ifndef Stack__H
#define Stack__H

  // Schnittstelle der Klasse Stack
class Stack
{
   // public-Sektion, alle hier angefuehrten Elemente
   // sind von aussen zugaenglich
 public:
    // Hier folgen die Methoden der Klasse
  void init();
  void push(int elem);
```

```
  int  pop();
  int  peek();
  bool isEmpty();
  bool wasError();

   // private-Sektion, alle hier angefuehrten
   // Elemente sind "geheim" und koennen nur intern
   // verwendet werden.
 private:
  const int MaxElems = 10;
  int       elems[MaxElems];
  int       top;
  bool      errOccd;
};
#endif
```

Die Implementierung erfolgt getrennt davon in einer eigenen Datei:

Programm 11.2: Klasse `Stack`, Implementierung

```
  // Datei: stack.cpp. Implementierung der Klasse Stack

  // Einfuegen der eigenen Schnittstelle
#include "stack.h"

  // globale Variablen sind nicht noetig, alle Daten
  // sind in der Klasse vereinbart
void Stack::init()
{
  top = 0;
  errOccd = false;
}

void Stack::push(int elem)
{
  errOccd = (top==MaxElems);
  if (!errOccd) {
    elems[top] = elem;
    ++top;
  }
}

int Stack::pop()
{
  errOccd = (top==0);
  if (!errOccd) {
    --top;
    return elems[top];
  } else {
    return 0;
  }
}
```

```
int Stack::peek()
{
  errOccd = (top==0);
  if (!errOccd) {
    return elems[top-1];
  } else {
    return 0;
  }
}

bool Stack::isEmpty()
{
  return (top==0);
}

bool Stack::wasError()
{
  return errOccd;
}
```

Programm 11.3 ist ein einfaches Testprogramm für die oben angeführte Klasse.

Programm 11.3: Stack-Testprogramm

```
  // Test-Program:
#include <iostream>
#include "stack.h"

void main()
{
  Stack         s;
  char          ch;
  int           i;

  s.init();
  cout << "\n\nOperation (Quit, pUsh, pOp, isEmpty) ";
  cin >> ch;

  while (ch != 'Q') {
    switch (ch) {
      case 'U':
        cout << "\n Element to push? ";
        cin >> i;
        s.push(i);
        if (s.wasError()) cout << " Error!";
        break;
      case 'O':
        i = s.pop();
        if (s.wasError()) {
          cout << " Error!";
        } else {
          cout << "\n popped element " << i;
        }
```

```
      break;
    case 'K':
      i = s.peek();
      if (s.wasError()) {
        cout << " Error!";
      } else {
        cout << "\n peeked element " << i;
      }
      break;
    case 'E':
      if (s.isEmpty()) {
        cout << "\n Stack is empty.";
      } else {
        cout << "\n Stack contains elements.";
      }
      break;
    default:
      cout << "\n Wrong operation.";
  }
  cout << "\n\nOperation (Quit, pUsh, pOp, isEmpty) ";
  cin >> ch;
 }
}
```

Allgemein wird hier festgelegt:

Klassen exportieren generell ausschließlich Methoden. Alle Daten sind im Innern (= `private`-Abschnitt) verborgen, der Zugriff erfolgt über die sogenannten Element-Funktionen.

Jede Klasse besteht damit aus zwei Dateien, der Schnittstellendatei und der Implementierungsdatei.

Die beiden Dateien im Detail:

- Die Schnittstelle einer Klasse wird in einem sogenannten `.h`-File abgelegt. Der `.h`-File enthält die Präprozessor-Anweisungen zur Verhinderung von Mehrfach-Includes und die eigentliche Klassendefinition.
- Die Implementierung der in der Schnittstelle angeführten Methoden erfolgt in dem entsprechenden `.cpp`-File. In dieser Datei wird auch die eigene Schnittstellendatei inkludiert.

Die Schnittstellen bestimmen die Verwendbarkeit der Klassen. Allgemein kann festgehalten werden, daß zu kleine (= zu wenig umfassende) Schnittstellen die Verwendbarkeit einschränken, eine zu große Schnittstelle aber sowohl die Testbarkeit als auch die Verständlichkeit erschweren. Die Kunst beim Gestalten einer Schnittstelle liegt darin, den richtigen ,,Mittelweg`` zu finden:

Die Schnittstelle einer Klasse (Klassen-*Interface*) sollte minimal und vollständig sein. Vollständig in dem Sinne, daß Benutzer der Klasse alle sinnvollen Aktionen ausführen können. Minimal wiederum bedeutet, daß das Klassen-*Interface* so klein wie möglich sein sollte.

Damit wird eine allgemeine Verwendbarkeit der Klasse erreicht, die Testbarkeit wesentlich erleichtert und die Abhängigkeiten zwischen der Klasse und ihren Benutzern minimiert.

Nach diesem einführenden Beispiel werden im folgenden die verschiedenen Aspekte von Klassen detailliert besprochen.

Vorige Seite: 11.1.1 Klassen als Mittel **Eine Ebene höher:** 11.1 Motivation

Vorige Seite: 11.1.2 Klassen und das **Eine Ebene höher:** 11 Das Klassenkonzept **Nächste Seite:** 11.2.1 `inline`-Element-Funktionen

11.2 Element-Funktionen

Element-Funktionen sind Funktionen, die in der Schnittstelle der Klasse spezifiziert sind. Sie haben folgende Eigenschaften:

- Sie haben vollen Zugriff auf alle Klassen-Elemente. Innerhalb von Element-Funktionen kann auf *alle* Klassen-Elemente (auch auf solche, die als `private` spezifiziert sind) zugegriffen werden.
- Sie sind innerhalb von Klassen deklariert und daher außerhalb der Klasse nicht sichtbar. Ihr Gültigkeitsbereich beschränkt sich - im Gegensatz zu normalen Funktionen - auf die umgebende Klasse. Daher können sie nur unter Bezugnahme auf ein Objekt der Klasse beziehungsweise mit dem *Scope*-Operator verwendet werden.
- Ihre Implementierung kann innerhalb der Klasse oder auch außerhalb erfolgen. Erfolgt die Definition außerhalb, so muß dem Funktionsnamen der Klassenname gefolgt von `::` vorangestellt werden.

Element-Funktionen spezifizieren das Verhalten von Objekten. Einige von ihnen bilden auch die Schnittstelle des Objekts nach außen hin (*Public Interface*) und können ganz allgemein in verschiedene Kategorien eingeteilt werden.

Eine Möglichkeit ist es zum Beispiel, zwischen Konstruktoren/Destruktoren, Modifikatoren, Selektoren und Iteratoren zu unterscheiden [Boo94]:

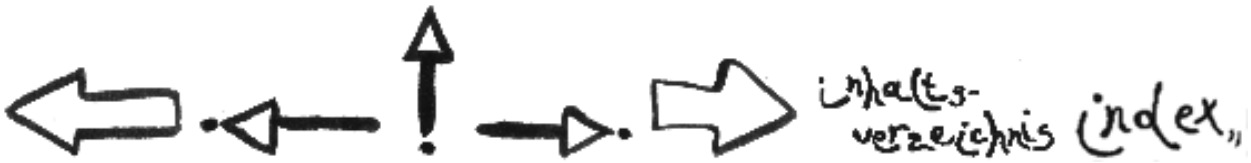
- Konstruktoren/Destruktoren
  Element-Funktionen zum Initialisieren beziehungsweise De-Initialisieren eines Objekts.
- Modifikatoren
  Element-Funktionen, die den Zustand eines Objekts verändern.
- Selektoren
  Zugriffsfunktionen, die nur lesend auf ein Objekt zugreifen.
- Iteratoren
  Element-Funktionen, die es erlauben, auf die Elemente eines Objekts in einer definierten Reihenfolge zuzugreifen.

Eine andere Möglichkeit ist die Einteilung in Managerfunktionen (*Manager Functions*), Implementierungsfunktionen (*Implementor Functions*), Hilfsfunktionen (*Helping Functions*) und Zugriffsfunktionen (*Access Functions*) [Lip91]:

- Managerfunktionen
  Spezielle Element-Funktionen, die die „Verwaltung`` der Klassen-Objekte übernehmen. Sie können neue Objekte erzeugen, bestehende zerstören, Objekte zuweisen, in andere konvertieren usw. Diese Art von Methoden wird normalerweise automatisch aufgerufen.

  Konstruktoren (*Constructors*) und Destruktoren (*Destructors*) sind zwei C++-spezifische Managerfunktionen, die automatisch aufgerufen werden, wenn ein Objekt neu erzeugt beziehungsweise zerstört wird.
- Implementierungsfunktionen
  Element-Funktionen, die die Funktionalität von bestimmenden Operationen realisieren. So wird zum Beispiel eine Klasse Stack zumindest die Implementierungsfunktionen `push`, `pop` oder `isEmpty` aufweisen.
- Hilfsfunktionen
  Element-Funktionen, die Hilfsdienste bereitstellen. Diese Funktionen werden üblicherweise als `private` vereinbart und nur intern verwendet.
- Zugriffsfunktionen
  Ermöglichen den Zugriff auf interne Datenelemente. Dabei kann es lesende und schreibende Zugriffsfunktionen geben (*Read Access Functions*, *Write Access Functions*). Lesende Zugriffsfunktionen sind Methoden, die die Instanzvariablen eines Objekts nicht verändern.

Vorige Seite: 11.1.2 Klassen und das **Eine Ebene höher:** 11 Das Klassenkonzept **Nächste Seite:** 11.2.1 `inline`-Element-Funktionen



Vorige Seite: 11.2 Element-Funktionen **Eine Ebene höher:** 11.2 Element-Funktionen **Nächste Seite:** 11.2.2 `const`-Element-Funktionen

---

11.2.1 `inline`-Element-Funktionen

`inline`-Element-Funktionen sind Funktionen, die entweder in der Klassenschnittstelle angeführt oder aber explizit als `inline` gekennzeichnet sind.

Element-Funktionen, die innerhalb einer Klassendefinition implementiert sind, werden als `inline`-Funktionen behandelt.

In Programm 11.4 ist eine Klasse `Vehicle` mit verschiedenen Element-Funktionen vollständig angeführt:

Programm 11.4: Klasse `Vehicle`

```
class Vehicle
{
 public:
   void  initialize(int inWheels, float inWeight);
   int   getWheels() { return wheels; }
   float getWeight() { return weight; }
   float wheelLoading();
 private:
   int   wheels;
   float weight;
};

  // als inline spezifizierte Element-Funktion
inline void  Vehicle::initialize(int inWheels,
                                 float inWeight)
{
   wheels = inWheels;
   weight = inWeight;
}

  // als inline spezifizierte Element-Funktion
inline float Vehicle::wheelLoading()
{
   return weight / wheels;
}
```

In Programm 11.4 sind die Element-Funktionen `getWheels` und `getWeight` direkt in der Schnittstelle der Klasse angeführt und damit implizit `inline` Element-Funktionen. Die Funktionen `initialize` und `wheelLoading` sind im Anschluß

daran explizit als `inline` definiert.

`inline`-Funktionen haben gegenüber „normalen`` Funktionen den Vorteil, daß der Compiler beim Aufruf den Code direkt einfügt (vergleiche dazu Abschnitt 7.5 und Anhang B.1). Damit kann bei einfachen und kleinen Funktionen eine höhere Effizienz erreicht werden, da kein Funktionsaufruf erfolgt.

Alle Klienten von Klassen inkludieren deren *Header-Files*, in denen die Schnittstellen spezifiziert sind. Damit entspricht die Angabe von Funktionsrümpfen in der Klassenschnittstelle der Bekanntgabe der Implementierung. Eine Änderung der Funktionsrümpfe ist zugleich eine Änderung der Schnittstelle der Klasse und zieht damit eine (unnötige und vermeidbare) Neu-Übersetzung aller Klienten nach sich. Aufgrund der mit `inline`-Element-Funktionen verbundenen Nachteile gilt daher:

Mit Ausnahme von sehr einfachen Methoden (zum Beispiel *Access*-Methoden) sollten keine Element-Funktionen in der Klassenschnittstelle vereinbart werden.

Vorige Seite: 11.2 Element-Funktionen **Eine Ebene höher:** 11.2 Element-Funktionen **Nächste Seite:** 11.2.2 `const`-Element-Funktionen

Vorige Seite: 11.2.1 `inline`-Element-Funktionen **Eine Ebene höher:** 11.2 Element-Funktionen

11.2.2 `const`-Element-Funktionen

Um zu verhindern, daß `const`-Objekte über den ,,Umweg`` von Element-Funktionen verändert werden, dürfen ,,normale`` Element-Funktionen nicht auf `const`-Objekte angewandt werden.

`const`-Element-Funktionen hingegen sind Element-Funktionen, die den Wert des Objekts, auf dem sie operieren, nicht verändern. Sie können daher sowohl auf `const`- als auch auf non-`const`-Objekte angewandt werden.

Die Definition einer Element-Funktion als `const` erfolgt durch das Hinzufügen des Schlüsselwortes `const` an den Funktionskopf. Dies muß sowohl im Prototyp der Element-Funktion als auch in der eigentlichen Implementierung erfolgen.

In Programm 11.5 werden die Element-Funktionen `isEmpty` und `wasError` des Programms 11.1 als `const` vereinbart, da sie den Zustand des Objekts nicht verändern. Das Ergebnis sieht wie folgt aus:

Programm 11.5: Fortsetzung Klasse `Stack`, Erweiterung um `const`-Element-Funktionen

```
class Stack
{
 public:
   void init();
   void push(const int& elem);
   int  pop();
   bool isEmpty() const;
   bool wasError() const;
 private:
   const int MaxElems = 10;
   int  elems[MaxElems];
   int  top;
   bool errOccd;
};

...
bool Stack::isEmpty() const
{
   return (top==0);
}
bool Stack::wasError() const
{
   return errOccd;
}
```

Um die Verwendung von `const`-Element-Funktionen weiter zu illustrieren, wird das bereits angeführte `Vehicle`-Beispiel (Programm 11.4) erweitert:

Programm 11.6: Klasse `Vehicle` mit `const`-Element-Funktionen inklusive Testprogramm

```
class Vehicle
{
 public:
   void  initialize(const int n=0, const int inWheels=0,
                    const float inWeight=0);
   void  setWeight(float w) { weight = w; }
   int   getWheels() const { return wheels; }
   float getWeight() const { return weight; }
   float wheelLoading() const;
   void  print() const;
 private:
   int    wheels;
   float  weight;
   int    nr;
};

void  Vehicle::initialize(int n, int inWheels,
                          float inWeight)
{
   wheels = inWheels;
   weight = inWeight;
   nr = n;
}

float Vehicle::wheelLoading() const
{
   return weight/wheels;
}

void  Vehicle::print() const
{
  cout << "Vehicle " << nr << " wheels " << wheels
       << " weight " << weight << endl;
}

  // **** Tests: Read Only-Parameter
void fooReadOnly(const Vehicle& v)
{
  v.print();                     // ok, const-Objekt und
                                 // const-Methode
  v.setWeight(v.getWeight()+19); // Fehler! const-Objekt
}                                // aber keine const-
                                 // Methode
void foo(Vehicle& v)
{
  v.print();                     // ok, non-const-Objekt,
                                 // const-Methode
  v.setWeight(v.getWeight()+19); // ok, non-const-Objekt,
}                                // non-const Methode
```

```
void main()
{
  Vehicle        v;

  v.initialize(1, 4, 1000);
  v.print();
  foo(v);
  fooReadOnly(v);
}
```

Das Programm liefert beim Übersetzen an der oben angegeben Stelle einen Fehler. Die Fehlermeldung des IBM-C++-Compilers zum Beispiel lautet so:

```
vehicle.cpp(48:14) : error EDC3164: non const member
function "Vehicle::setWeight(float)" cannot be called
for a const object.
```

Der C++-Compiler erkennt also bereits zur Übersetzungszeit, daß hier versucht wird, ein `const`-Objekt (möglicherweise) zu verändern. Dabei spielt es keine Rolle, ob eine non-`const`-Element-Funktion eine Instanzvariable *tatsächlich* verändert. Der Compiler untersucht lediglich, ob eine nicht als `const` deklarierte Element-Funktion auf ein als `const` deklariertes Objekt angewandt wird.

`const`-Element-Funktionen haben mehrere Vorteile. Zum einen können sie sowohl auf non-`const`- als auch auf `const`-Objekte angewandt werden, zum anderen bewirkt die Verwendung von `const` eine deutlich höhere Sicherheit, da in `const`-Element-Funktionen die Daten eines Objekts nicht (versehentlich) geändert werden können. Das führt bereits bei der Übersetzung zu einem Fehler. Daher sollten grundsätzlich alle Funktionen, die den Zustand eines Objekts nicht verändern (Selektoren), als `const` vereinbart werden.


Grundsätzlich sind alle Funktionen, die den Zustand eines Objekts nicht verändern (Selektoren), als `const` zu vereinbaren.

Achtung: `const`- und non-`const`-Funktionen unterscheiden sich in ihrer Signatur, auch wenn ansonsten Name und Parameterliste identisch sind. Aus dieser Tatsache resultieren viele auf den ersten Blick seltsam anmutende Fehlermeldungen des Linkers.

Im Zusammenhang mit `const`-Objekten beziehungsweise `const`-Methoden ist auch das Attribut `mutable` von Bedeutung. Folgendes Beispiel zeigt dies: In einer Klasse `Stack` werden Zähler für die Anzahl der lesenden und schreibenden Zugriffe eingeführt. Diese Zähler werden entsprechend in den verschiedenen Methoden erhöht: Ein Aufruf von `isEmpty` beziehungsweise `wasError` erhöht den Zähler für lesende Zugriffe, ein Aufruf von `push` oder `pop` führt zur Inkrementierung des Zählers für schreibende Zugriffe.

```
class Stack
{
 public:
  ...
  int  isEmpty() const;
  int  wasError() const;

 private:
  ...
  int    nrOfReaders; // Anzahl d. Lesezugriffe
  int    nrOfWriters; // Anzahl d. Schreibzugriffe
```

```
};
```

Da die lesenden Methoden (Methoden, die ein Objekt nicht verändern) als `const` deklariert sind, dürfen sie keine Werte von Datenelementen verändern und können damit auch den Zähler für die Anzahl der lesenden Zugriffe *nicht* inkrementieren. Grundsätzlich kann

1. die Verwendung von `const`-Methoden überhaupt vermieden werden oder
2. das entsprechende Datenelement über einen `const_cast` bei der Verwendung umgewandelt werden:

```
int Stack::isEmpty() const
{
  ++const_cast<int&>nrOfReaders; // const "weg-
  return (top==0);               // casten"
}
```

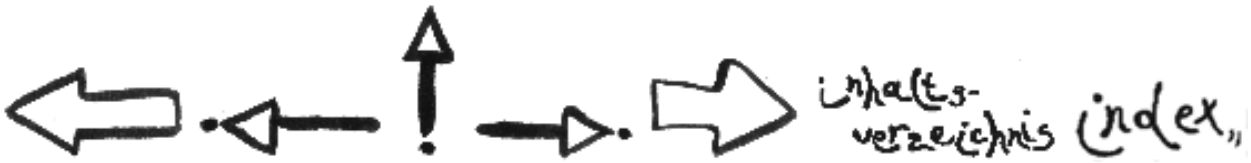
Beide Möglichkeiten sind nicht sehr zufriedenstellend. Im Falle der Vermeidung von `const`-Methoden wird das Problem nur umgangen und zudem in bezug auf `const`-Objekte nicht gelöst. Die zweite Möglichkeit ,,funktioniert`` zwar auch bei `const`-Objekten, ist aber ,,unschön``.

Für derartige Fälle gibt es in C++ das Speicherklassenattribut `mutable`. `mutable` deklariert ein Datenelement als ,,nie konstant`` und vermeidet damit die ansonsten nötigen `const`-Typumwandlungen:

```
class Stack
{
 public:
  ...
  int  isEmpty() const;
  int  wasError() const;

 private:
  ...
  mutable int nrOfReaders; // Anzahl Lesezugriffe
  mutable int nrOfWriters; // Anzahl Schreibzugriffe
};
...
int Stack::isEmpty() const
{
  nrOfReaders++;           // erlaubt, weil mutable
  return (top==0);
}
...
```

- Datenelemente in konstanten Objekten, die veränderbar sein sollen, werden als `mutable` deklariert.

Vorige Seite: 11.2.1 `inline`-Element-Funktionen **Eine Ebene höher:** 11.2 Element-Funktionen



Vorige Seite: 11.2.2 `const`-Element-Funktionen **Eine Ebene höher:** 11 Das Klassenkonzept **Nächste Seite:** 11.4 Der Zugriffsschutz bei

11.3 `this`-Zeiger

In jedem Objekt steht automatisch ein Zeiger zur Verfügung, der auf dieses selbst zeigt. Er heißt `this` und kann unter anderem dazu verwendet werden, auf die Adresse des eigenen Objekts und auf Elemente des Objekts zuzugreifen oder Referenzen auf das eigene Objekt zu bilden.

Im folgenden Beispiel wird `this` auf verschiedene Arten verwendet:

```
AnyClass::AnyClass(const AnyClass& o)
{
  this->anyFoo();       // ruft Element-Fkt. ueber this
                        // auf ("this" ist unnoetig)
  if (this == &o) ...   // testen, ob eigene Adresse
                        // gleich Adresse von o ist
  return *this;         // gibt eigenes Objekt zurueck
}
```

Als ein weiteres Beispiel für die Verwendung von `this` wird `Stack` so erweitert, daß mehrere Methoden direkt hintereinander ausgeführt werden können. Anstelle der drei Anweisungen

```
s.push(1);
s.push(7);
s.push(12);
```

kann dann auch die Anweisungsfolge

```
s.push(1).push(7).push(12);
```

codiert werden.

Bei einer Klasse `Stack` scheint das eher von akademischer Bedeutung, aber in vielen anderen Fällen (zum Beispiel bei mathematischen Klassen) hat diese Vorgehensweise durchaus Vorteile.

Anweisungen der Art `s.push(1).push(7).push(12);` werden von links nach rechts abgearbeitet:

`((s.push(1)).push(7)).push(12)`

Damit `s.push(1).push(7)` zum richtigen Ergebnis führt (`s` um die Elemente 1, 7 erweitert), muß `s.push(1)` nach der Ausführung der Operation `push(1)` sich selbst (= den `Stack` *nach* der Aufnahme des Elements `1`) zurückgeben. Dann wird die Operation `push(7)` auf dieses Objekt angewandt.

Das eigene Objekt kann in einer Methode nur unter Verwendung von `this` referenziert werden: `this` verweist auf das eigene Objekt, `*this` ist daher das eigene Objekt. Wichtig ist, daß der Rückgabewert lediglich eine Referenz und *kein* eigenes Objekt ist. Ist der Rückgabewert nicht als Referenz vereinbart, wird eine *Kopie* angelegt! (Siehe dazu auch die Abschnitte 8.1 und 8.5).

In Programm 11.7 ist eine erweiterte Schnittstelle und in Programm 11.8 die zugehörige Implementierung der Klasse `Stack` angegeben. Die Funktionalität der Klasse entspricht dabei im wesentlichen der der Klasse `Stack` aus den Programmen 11.1 und 11.2. Allerdings werden die Elemente des Stacks nun mittels einer einfach verketteten Liste gespeichert, um gegenüber einer statischen Lösung eine bessere Speicherplatzausnutzung zu garantieren.

Programm 11.7: Klasse `Stack` (dynamisch, Referenzen), Schnittstelle

```
  // Node ist ein einzelnes Stack-Element, data nimmt
  // die eigentlichen Daten auf, next verweist auf
  // das jeweils naechste Element
struct Node {
      int    data;
      Node*  next;
};

  // Stack wird mit einer einfach verketteten Liste
  // realisiert (Node)
class Stack
{
   public:
      Stack& init();
      Stack& push(const int& elem);
      Stack& pop(int& elem);
      bool isEmpty() const;
      bool wasError() const;
   private:
      Node*   topElem;
      bool    errOccd;
};
```

Programm 11.8: Klasse `Stack` (dynamisch, Referenzen), Implementierung

```
...
Stack& Stack::init()
{
  topElem = 0;
  errOccd = false;
  return *this;
}

Stack& Stack::push(const int& elem)
{
  Node*    newElem;

  newElem = new Node;
  newElem->data = elem;
  newElem->next = topElem;
  topElem = newElem;
  errOccd = false;
  return *this;
}

Stack& Stack::pop(int& elem)
{
  Node*    pNode;

  errOccd = isEmpty();
  if (!errOccd) {
    pNode = topElem;
    elem = topElem->data;
```

```
    topElem = topElem->next;
    delete pNode;
  }
  return *this;
}
...
```

Auf Objekte der Klasse `Stack` können zum Beispiel die Operationen `push` und `pop` wie oben angegeben hintereinander ausgeführt werden.

Vorige Seite: 11.2.2 `const`-Element-Funktionen **Eine Ebene höher:** 11 Das Klassenkonzept **Nächste Seite:** 11.4 Der Zugriffsschutz bei

Vorige Seite: 11.3 `this`-Zeiger **Eine Ebene höher:** 11 Das Klassenkonzept **Nächste Seite:** 11.4.1 Zugriffsschutz mit `publicprotected` und `private`

11.4 Der Zugriffsschutz bei Klassen

Eine kurze Einführung in die grundsätzlichen Möglichkeiten, die Daten von Objekten zu schützen, wurde bereits am Anfang des Kapitels gegeben. In diesem Abschnitt werden die verschiedenen Möglichkeiten, die C++ bietet, im Detail besprochen.

Vorige Seite: 11.4 Der Zugriffsschutz bei **Eine Ebene höher:** 11.4 Der Zugriffsschutz bei **Nächste Seite:** 11.4.2 `friend`-Funktionen und -Klassen

---

11.4.1 Zugriffsschutz mit `public`, `protected` und `private`

Das Schlüsselwort `public` in der Klassendefinition bedeutet, daß die danach angeführten Elemente (Funktionen und Daten) von außen sichtbar sind. Im Gegensatz dazu kann von außen nicht auf sogenannte `private`-Elemente zugegriffen werden.

Im Programm 11.6 sind `wheels` und `weight` unterhalb des Schlüsselworts `private` angeführt und damit von außen nicht sichtbar. Neben `private` und `public` gibt es in C++ eine dritte Möglichkeit des Zugriffsschutzes: `protected`. Die drei Möglichkeiten im Überblick:

- `public`-Elemente sind frei zugänglich und können sowohl von innerhalb der Klasse (= in den eigenen Element-Funktionen) als auch von außerhalb (= von anderen Funktionen aus) angesprochen werden.
- `protected`-Elemente sind für Element-Funktionen der *eigenen* Klasse frei zugänglich. Zusätzlich kann von abgeleiteten Klassen (auf Vererbung wird in Kapitel 13 eingegangen) auf `protected`-Elemente der eigenen Basisklassen zugegriffen werden.

  Der Zugriff auf ererbte `protected`-Elemente von anderen Objekten (auch wenn sie derselben Klasse angehören) ist nicht erlaubt. `friend`s der Klasse (siehe Abschnitt 11.4.2) dürfen ebenfalls auf `protected`-Elemente zugreifen. Für alle anderen Funktionen verhalten sich `protected`-Elemente wie `private`-Elemente und sind verborgen.
- Nur von innerhalb der Klasse (= von eigenen Element-Funktionen) und von `friend`-Klassen beziehungsweise -Funktionen aus kann auf die `private`-Elemente zugegriffen werden. Von außen sind diese Elemente nicht sichtbar. Auch von abgeleiteten Klassen aus ist kein Zugriff möglich.

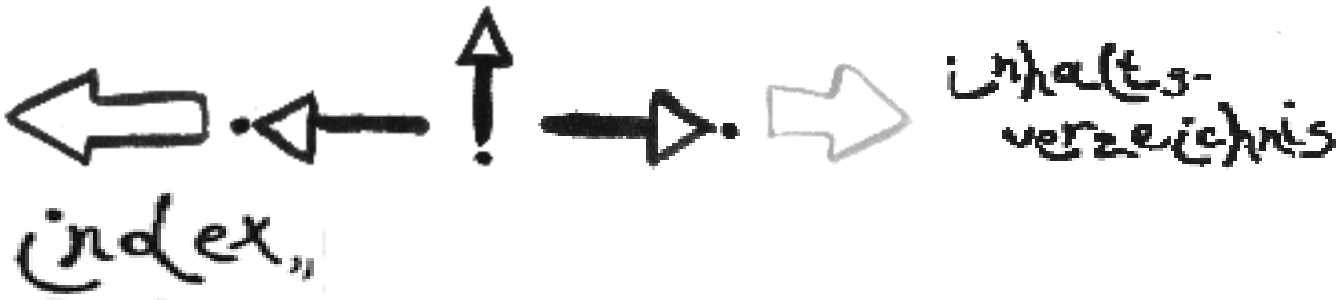
Die Zugriffsebenen `public`, `protected` und `private` gelten jeweils bis ein neuer Abschnitt spezifiziert wird. Damit ist es zum Beispiel möglich (aber nicht üblich), mehrere `public`-Sektionen innerhalb einer Klassendefinition zu codieren.

Da grundsätzlich keine Daten exportiert werden (sollten), können in Übereinstimmung mit [HN93] folgende Codierrichtlinien beziehungsweise -empfehlungen festgelegt werden:

Die Elemente einer Klasse werden in dieser Reihenfolge angeführt: Zuerst alle `public`-, dann die `protected`- und schließlich die `private`-Elemente. Alle Daten werden (bis auf wenige Ausnahmen) in der `private`-Sektion angeführt. Der Zugriff erfolgt über Element-Funktionen, die als `public` vereinbart sind.

In den einzelnen Gruppen (`public`, `protected`, `private`) können dann die Funktionen nach verschiedenen Kriterien (zum Beispiel nach den in Abschnitt 11.2 angegebenen) gruppiert werden. Auf die Verwendung von `protected` wird später (Kapitel 13) noch genauer eingegangen. Es kann aber bereits jetzt festgehalten werden, daß `protected` nur sehr sparsam eingesetzt werden sollte!

Vorige Seite: 11.4 Der Zugriffsschutz bei **Eine Ebene höher:** 11.4 Der Zugriffsschutz bei **Nächste Seite:** 11.4.2 `friend`-Funktionen und -Klassen



Vorige Seite: 11.4.1 Zugriffsschutz mit `publicprotected` und `private` **Eine Ebene höher:** 11.4 Der Zugriffsschutz bei

---

11.4.2 `friend`-Funktionen und -Klassen

C++ definiert sehr strenge Zugriffsregeln für Klassen (`public`, `protected` und `private`). In den meisten Fällen sind diese von Vorteil (Geheimnisprinzip, Kapselung, Zugriffsschutz und so weiter), allerdings gibt es einige wenige Fälle, in denen sie hinderlich sind.

`friend`-Deklarationen durchbrechen diese Regeln. Jede Klasse kann andere Klassen oder Funktionen „zum Freund`` erklären und ihnen damit spezielle Zugriffsrechte einräumen: Jeder `friend` darf auf *alle* Elemente der Klasse (`public`, `protected`, `private`) zugreifen.

Ein Beispiel für `friend`-Vereinbarungen:

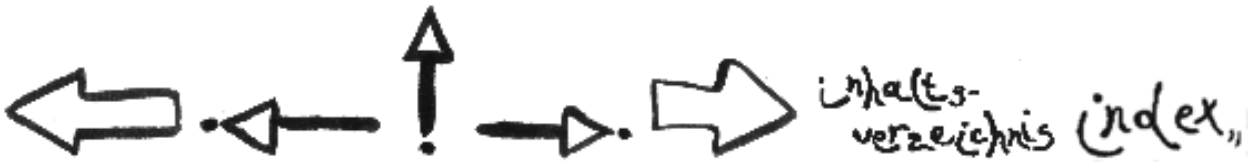
```
class X
{
 private:
  int  i;
  char c;

  friend class Y;
  friend void foo();
};

class Y
{
  ...
  void testX();
};

void Y::testX()
{
  X   x;
  x.i = 7;     // private-Zugriff erlaubt, weil Y
```

```
}               // ein friend von X ist

void foo()
{
  X   x;
  x.c = 'a';    // private-Zugriff erlaubt, weil foo
}               // ein friend von X ist
```

Die Vereinbarung von `friend`-Funktionen oder -Klassen stellt eine Verletzung des Geheimnisprinzips dar und ist nur in sehr wenigen Fällen zwingend notwendig. In einigen Fällen kann sie vorteilhaft sein, in den meisten Fällen gilt aber folgendes:

`friend`s, insbesondere `friend`-Klassen, sind ein Anzeichen für schlechtes Design und durchbrechen wichtige Prinzipien der objektorientierten Programmierung wie das Geheimnisprinzip. Die Verwendung von `friend` sollte daher weitgehend unterbleiben.

Ein Beispiel für die Verwendung von `friend` ist in Abschnitt 11.7.3 angeführt.



Vorige Seite: 11.4.2 `friend`-Funktionen und -Klassen **Eine Ebene höher:** 11 Das Klassenkonzept **Nächste Seite:** 11.6 Geschachtelte Klassen

---

11.5 `static`-Klassen-Elemente

`static`-Datenelemente sind eine Art globale Variablen, die klassenweit gelten. Alle statischen Elemente einer Klasse existieren nur einmal pro Klasse und alle Objekte einer Klasse haben Zugriff auf dieselben statischen Klassen-Elemente. Das hat gegenüber ,,normalen`` globalen Variablen den Vorteil, daß nur Objekte dieser Klasse Zugriff auf diese Variablen haben und daß statische Variablen nur im entsprechenden Klassen-*Scope* gültig sind.

Abbildung 11.1 zeigt eine Klasse `StatClass` und zwei Objekte. Die Klasse enthält zwei ,,normale`` sowie ein statisches Datenelement. Die beiden ,,normalen`` Datenelemente sind in jedem der beiden Objekte enthalten, während das statische Datenelement nur einmal vorhanden ist und von beiden Objekten referenziert wird.

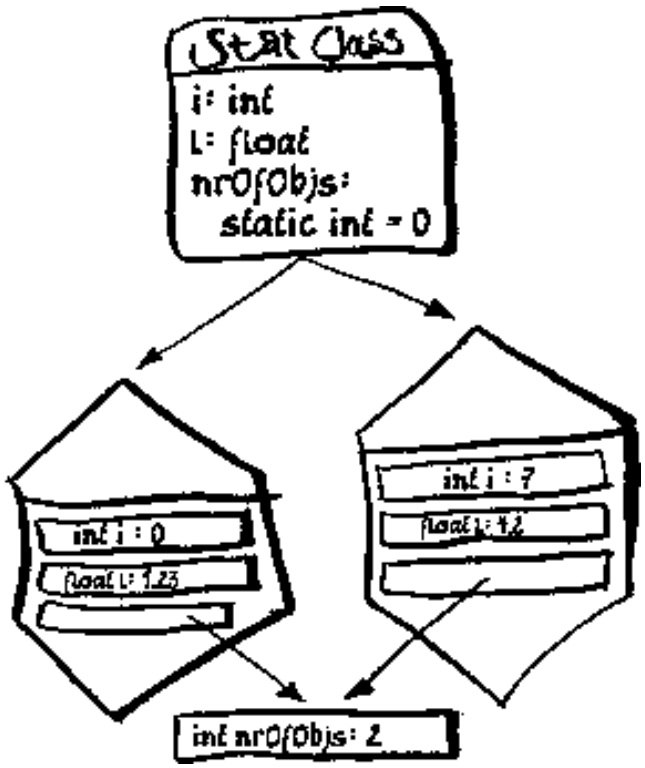


Abbildung 11.1: Statische Klassen-Elemente

```
  // Schnittstelle, Datei t.h
class T
{
  ...
 private:
  static int nrOfObjects=0;   // Deklaration
};


  // Implementierung, Datei t.cpp
```

```
...
static int T::nrOfObjects;  // Definition
```

Eine Konstante mit klassenweitem Gültigkeitsbereich kann direkt bei der Deklaration initialisiert werden:

```
class A
{
  ...
  static const int MaxSize = 100; // Initialisierung
  ...
};
```

Dennoch muß eine Definition der Konstante außerhalb erfolgen. Diese Definition darf *keine weitere* Initialisierung enthalten:

```
...
static const int A::MaxSize; // ohne Initialisierung!
...
```

Neben statischen Klassenvariablen können auch Element-Funktionen als statisch deklariert werden. Allerdings können statische Element-Funktionen keine nicht statischen Datenelemente oder Element-Funktionen referenzieren.

Auch ist zu beachten, daß statische Klassen-Objekte keinen `this`-Zeiger enthalten, da sie ja an kein bestimmtes Objekt gebunden sind, sondern klassenweit gelten. Jede Verwendung des `this`-Zeigers in statischen Element-Funktionen führt daher zu einem Compiler-Fehler.

`static`-Elemente können aber im Gegensatz zu „normalen`` Klassen-Elementen auch *ohne* ein entsprechendes Objekt referenziert werden, indem der Gültigkeitsbereich (= die Klasse) angegeben wird.

Beispiele für die Verwendung von `static`-Klassen-Elementen sind in Programm 11.9 angeführt.

Programm 11.9: Statische Klassen-Elemente

```
#include <iostream>

class A
{
 public:
   void setI(const int aInt) { i = aInt; };
   void printI() { cout << "i = " << i << "\n"; };

   static void setIStat(const int aInt) { iStat = aInt; }
   static void printIStat() { cout << "i-stat = " << iStat
                                   << "\n"; };
 private:
   int          i;
   static int   iStat = 100;
};
int A::iStat;  // Definition des static-Datenelements
void main()
{
  A    a;
  A    a1;
  a1.printIStat();
  a.printIStat();
  a1.setIStat(101);
  a.printIStat();
  a.setI(1);
  a.printI();
  a.printIStat();
  a1.setI(2);
```

```
  a1.printI();
  a1.printIStat();
  A::printIStat();  // statisch, daher Zugriff auch
}                   // ueber Klasse moeglich
```

Programm 11.9 liefert folgende Ausgabe:

```
i-stat = 100
i-stat = 100
i-stat = 101
i = 1
i-stat = 101
i = 2
i-stat = 101
i-stat = 101
```

Vorige Seite: 11.4.2 `friend`-Funktionen und -Klassen **Eine Ebene höher:** 11 Das Klassenkonzept **Nächste Seite:** 11.6 Geschachtelte Klassen

Vorige Seite: 11.5 `static`-Klassen-Elemente **Eine Ebene höher:** 11 Das Klassenkonzept **Nächste Seite:** 11.7 Spezielle Element-Funktionen und

11.6 Geschachtelte Klassen

Klassen können global deklariert werden oder aber auch lokal, das heißt innerhalb von anderen Klassen oder Funktionen. In diesem Fall gilt die deklarierte Klasse auch nur im umgebenden Gültigkeitsbereich.

Ein Anwendungsfall dafür sind etwa *Collection Classes* , die alle eine Datenstruktur Iterator enthalten.

Iteratoren sind Datenstrukturen, die es ermöglichen, die enthaltenen Elemente in einer ,,wohldefinierten`` Reihenfolge zu durchlaufen (mehr dazu später). Existieren die Klassen `List`, `Bag`, `Tree` und `SortedCollection`, so müssen auch Iteratoren für diese Klassen definiert werden. Diese Iteratoren haben nur im Zusammenhang mit ihrer jeweiligen *Collection Class* Sinn und können auch nur im Kontext mit dieser Klasse verwendet werden.

Um potentielle Namenskonflikte zu vermeiden, können zum Beispiel Namensräume eingesetzt oder die global gültigen Iterator-Klassen mit verschiedenen Namen (zum Beispiel `ListIterator`, `BagIterator` und `SortedCollectionIterator`) versehen werden.

Eine andere Möglichkeit ist, in allen *Collection Classes* eine interne Klasse `Iterator` zu vereinbaren. Wesentlicher Vorteil dieser Vorgangsweise ist die lokale Gültigkeit der Namen: Die Iteratoren aller Klassen heißen `Iterator`, Namenskonflikte treten aufgrund der lokalen Gültigkeit nicht auf, und die Klassen sind genau dort vereinbart, wo sie sinnvoll verwendet werden können.

```
class List
{
 public:
  ...
  class Iterator
  {
    ...
  }
};
```

In vielen Fällen soll von außen kein Zugriff auf Internas der geschachtelten Klasse möglich sein, die umgebende Klasse hingegen soll sehr wohl auf alle Elemente der enthaltenen Klasse zugreifen können. Wird eine Klasse `Node` innerhalb einer Klasse `List` definiert, so hat allerdings weder `Node` spezielle Zugriffsrechte auf `List`, noch `List` auf `Node`. Um dennoch `List` Zugriff auf `Node` zu gestatten, kann zum Beispiel `Node` im `public`-Teil von `List` vereinbart werden und `List` zu einem `friend`

gemacht werden:

```
class List
{
 public:
  ...
  class Node
  {
    friend class List;
    ...
  }
 private:
  ...
};
```

Elemente der Klasse `List` haben im obigen Beispiel Zugriff auf alle Elemente von `Node`.

Eine andere Möglichkeit der Deklaration räumt der umgebenden Klasse `B` Zugriff auf alle Elemente der internen Klasse `Node` ein, während die enthaltene Klasse `Node` von außen nicht sichtbar ist: Alle Elemente von `Node` werden als `public` vereinbart (beziehungsweise `Node` als `struct`) und die Klasse `Node` wird im `private`-Teil von `List` definiert. Damit kann nur von `List` aus auf `Node` zugegriffen werden.

```
class List
{
 public:
  ...
 private:
  ...
  struct Node
  {
    ...  // alle Elemente als public vereinbaren
  }
};
```

Um die unnötige „Aufblähung`` von Klassendefinitionen zu vermeiden, kann eine geschachtelte Klasse auch in der umgebenden Klasse *deklariert* und dann außerhalb *definiert* werden:

```
class B
{
  ...
 private:
  ...
  class A;
};
...
class B::A
{
  ... // Definition der geschachtelten Klasse
```

```
}
```

Bei der Definition kann die Klasse nur mehr über den Gültigkeitsbereich in Verbindung mit ihrem Namen angesprochen werden. Mit geschachtelten Klassen kann die sogenannte *Namespace Pollution* vermieden werden: Klassen, die nur innerhalb von anderen Klassen verwendet werden, sollten auch nur dort gelten. Werden solche Klassen global deklariert, so ,,verschmutzen`` sie den Namensraum.

Ein Beispiel für eine derartige Verschmutzung ist Programm 11.7: Die Klasse `Node` ist nur innerhalb von `Stack` von Bedeutung und sollte auch dort deklariert sein.

Die Schnittstelle der Klasse `Stack` muß daher wie folgt abgeändert werden:

```
class Stack
{
  ...
 private:
  struct Node;  // Deklaration der inneren Klasse Node

  Node*   topElem;
  ...
};
```

Die Klasse `Node` ist hier als intern deklariert. Die Definition ist in der Schnittstelle nicht nötig und erfolgt daher erst in der Implementierungsdatei. Die geschachtelte Klasse wird zum Beispiel in Abschnitt 11.7.3 verwendet, wo die Klasse `Stack` um Ein-/Ausgabeoperatoren erweitert wird.

Vorige Seite: 11.5 `static`-Klassen-Elemente **Eine Ebene höher:** 11 Das Klassenkonzept **Nächste Seite:** 11.7 Spezielle Element-Funktionen und

Vorige Seite: 11.6 Geschachtelte Klassen **Eine Ebene höher:** 11 Das Klassenkonzept **Nächste Seite:** 11.7.1 Konstruktoren

11.7 Spezielle Element-Funktionen und Operatoren

In diesem Abschnitt werden die wichtigsten speziellen Element-Funktionen der Sprache C++ erläutert. Zentrale Rolle kommt dabei naturgemäß den Funktionen zur Klasseninitialisierung beziehungsweise De-Initialisierung zu. Daneben zeigt der Abschnitt, wie im Zusammenhang mit eigenen Klassen verschiedene Operatoren (Ein-/Ausgabe, `[]`- oder Konversionsoperator etc.) überladen und sinnvoll eingesetzt werden können.

Um die verschiedenen Konzepte darzulegen, wird ein durchgängiges Beispiel verwendet. Dabei wird eine spezielle Klasse, eine einfache *String*-Klasse (`TString`), entworfen und schrittweise erweitert. Im ISO-Standard von C++ ist zwar eine derartige Klasse standardmäßig vorgesehen (was die Entwicklung einer Klasse `TString` unnötig erscheinen läßt), allerdings eignen sich *String*-Klassen als Einführungsbeispiel besonders gut, weil sie leicht verständlich sind und hohe Anforderungen bezüglich Flexibilität und Einsetzbarkeit aufweisen.

Vorige Seite: 11.7 Spezielle Element-Funktionen und **Eine Ebene höher:** 11.7 Spezielle Element-Funktionen und **Nächste Seite:** 11.7.2 Destruktoren

---

Teilabschnitte

11.7.1 Konstruktoren

Konstruktoren sind spezielle Element-Funktionen, die zur Klasseninitialisierung dienen. Mit ihnen werden die Elemente eines Objekts bei der Erzeugung mit bestimmten Werten belegt. Konstruktoren haben denselben Namen wie ihre Klasse und keinen Rückgabewert. Sie können aber beliebige Parameter aufweisen und auch überladen werden.

Wird für eine Klasse kein Konstruktor definiert (wie in den bisherigen Beispielen), so generiert der C++-Compiler automatisch einen sogenannten *Default*-Konstruktor . *Default*-Konstruktoren sind Konstruktoren ohne Parameter. Die vom System erzeugten *Default*-Konstruktoren sind im Endeffekt „leere`` Element-Funktionen und tun nichts.

Konstruktoren werden *automatisch* vom System aufgerufen, und zwar jedesmal bei der Objekt-Erzeugung. Bei der Definition

```
Stack     s;
```

wird Speicherplatz für `s` angelegt und (automatisch) der Konstruktor `Stack::Stack()` aufgerufen.

Da für `Stack` bisher kein Konstruktor definiert wurde, wird hier der vom Compiler generierte *Default*-Konstruktor aufgerufen. Bei den Anweisungen

```
Stack*    pS;
pS = new Stack;
```

wird durch den `new`-Operator Speicher für ein `Stack`-Objekt angefordert. Kann dieser Speicherplatz bereitgestellt werden (und nur dann), so wird für das erzeugte Objekt der Konstruktor (automatisch) aufgerufen.

Wie bereits erwähnt, werden Konstruktoren vom System automatisch beim Anlegen von Objekten aufgerufen. Element-Funktionen, die vom C++-System aufgerufen werden, sollten nicht explizit vom Progammierer aufgerufen werden.

Ist es nötig, die Initialisierung von Objekten ,,von Hand aus`` aufzurufen, so ist es besser eine eigene Initialisierungsfunktion zu erstellen, die dann auch im Konstruktor Verwendung findet.

Element-Funktionen, die automatisch aufgerufen werden (wie etwa Konstruktor, Destruktor etc.), sollten nicht von Hand aus aufgerufen werden.

Im folgenden Programm wird eine Klasse `TString` eingeführt. `TString` dient dazu, Zeichenketten zu verwalten. Im Gegensatz zu Standard-C-Zeichenketten werden allerdings beim Zugriff auf `TString`-Elemente Bereichsprüfungen vorgenommen. Zudem wird die Länge der Zeichenkette explizit mitgeführt.

Die Basis dieser Klasse wird im folgenden sukzessive erweitert und dient dazu, die Implementierung und die Funktionalität von verschiedenen Element-Funktionen und Operatoren zu zeigen. Dazu werden spezielle Konstruktoren, Destruktoren, Operatoren etc. hinzugefügt. Die Klasse ist [Lip91] entnommen, abgeändert und erweitert worden.

```
class TString
{
 public:
  TString();  // Konstruktor
  int getLen() const { return len; }
 private:
  int    len;
  char*  str;
};
```

`len` bestimmt die Länge der Zeichenkette, die in `str` gespeichert wird. Damit die Länge von `TString`-Objekten zur Laufzeit festgelegt und verändert werden kann, muß `str` ein `char*`-Zeiger sein, also kein `char`-Vektor sein darf. So können *Strings* beliebig lang werden, und Objekte dieser Klasse verbrauchen stets möglichst wenig Speicherplatz.

Bei der Erzeugung eines neuen `TString`-Objekts müssen diese Datenelemente in einem Konstruktor initialisiert werden, der so aussehen könnte:

```
TString::TString()
{
  len = 0;
  str = 0;
}
```

Achtung: Die Deklaration

```
TString myString();
```

ist keine Vereinbarung eines `TString`-Objekts. Sie entspricht vielmehr der Deklaration eines Funktionsprototyps. `TString`-Objekte, bei deren Vereinbarung der *Default*-Konstruktor aufgerufen

werden soll, werden so deklariert:

```
TString myString;
```

Objekt-Initialisierung mittels Initialisierungsliste

Objekte können bei der Erzeugung mit bestimmten Werten initialisiert werden. Grundsätzlich kann diese Initialisierung wie oben im Anweisungsteil von Konstruktoren oder aber durch die Angabe einer *Initialisierungsliste* erfolgen. Die Initialisierungsliste wird in der Implementierung von Konstruktoren *vor* dem eigentlichen Anweisungsteil angeführt und ist durch einen Doppelpunkt (`:`) vom Funktionskopf getrennt. In dieser Liste stehen alle Initialisierungen, die beim Aufruf des Konstruktors ausgeführt werden.

Ein Beispiel zeigt die Unterschiede zwischen der Initialisierung von Datenelementen mittels Zuweisung im Anweisungsteil beziehungsweise durch eine Initialisierungsliste.

```
class Dummy
{
 public:
  Dummy(const TString& initStr, const char* initPChar);
 private:
  TString   s;
  char*    data;
};
```

(1) Initialisierung mittels Initialisierungsliste:

```
Dummy::Dummy(const TString& initStr, const char* initPChar)
 : s(initStr), data(initPChar)
{}
```

(2) Initialisierung über Anweisungen:

```
Dummy::Dummy(const TString& initStr, const char* initPChar)
{
  s = initStr;
  data = initPChar;
}
```

Die beiden Konstruktoren (1) und (2) unterscheiden sich darin, daß in (1) die Instanzvariablen *initialisiert* werden, während in (2) die Wertbelegung durch *Zuweisungen* im Anweisungsteil erfolgt.

Eine Initialisierung ist *keine* Zuweisung. Jede Zuweisung bezieht sich auf ein bereits existierendes Objekt. Im Detail ist der Ablauf bei der Initialisierung über Anweisungen also wie folgt:

1. Zunächst werden die Datenelemente angelegt. Beim Anlegen wird Speicher angefordert und ein entsprechender Konstruktor aufgerufen.
2. Im Anweisungsteil erfolgt dann die Zuweisung. Zum genauen Ablauf einer Zuweisung sei auf

Abschnitt 13.5.4 hingewiesen. Vorerst genügt es zu wissen, daß bei der Zuweisung das gerade angelegte Objekt zum Teil wieder zerstört wird.

Im Gegensatz dazu erfolgt die Initialisierung bereits beim Anlegen eines Objekts. Der zweite Schritt fällt also weg.

Dies bringt zum Teil erhebliche Effizienzvorteile mit sich. Zudem können Konstanten und Referenzen zwar initialisiert, nicht aber zugewiesen werden. Daher wird folgende Empfehlung gegeben:

Objekt-Initialisierungen werden, sofern dies möglich ist, über die sogenannte Initialisierungsliste des Konstruktors und nicht im Anweisungsteil durchgeführt. Aus Übersichtsgründen sollten Datenelemente in der Reihenfolge initialisiert werden, in der sie in der Klassenschnittstelle angeführt sind.

Überladen von Konstruktoren

Konstruktoren können so wie andere Element-Funktionen auch überladen werden (vergleiche Abschnitt 7.7). Ein neuer Konstruktor für die Klasse `TString` zum Beispiel soll es erlauben, `TString`-Objekte bereits bei der Erzeugung mit einem Wert zu belegen. Damit können Objekte auf folgende Art vereinbart (und initialisiert) werden:

```
TString      s("Hello");
```

Der neue Konstruktor hat also einen Parameter vom Typ `const char*`. Will man neben dem eigentlichen `char*`-Parameter auch noch optional eine Länge angeben können, so ergibt sich folgende Schnittstelle für den neuen Konstruktor:

```
TString(const char* p, int l=0);
```

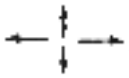
Die Implementierung könnte wie folgt aussehen:

```
TString::TString(const char* p, int l)
{
  if (p == 0) {
    len = 0;
    str = 0;
  } else {
    if (l == 0) {
      len = strlen(p);
    } else {
      len = l;
    }
    str = new char[len+1];
    memcpy(str, p, len);
    str[len] = 0x00;
```

```
  }
}
```

Ist `p` leer, so wird das aktuelle Element mit den Werten für einen leeren *String* initialisiert. Andernfalls muß das Objekt so initialisiert werden, daß es eine Kopie von `p` enthält. Dazu wird die Länge `len` gesetzt, ein entsprechend großer Speicherbereich für `str` allokiert (`len` + 1 Byte, um auch das Ende-Zeichen `0x00` aufnehmen zu können), und der Parameter `p` zeichenweise kopiert.

Die Anweisung `str = p` kann hier nicht verwendet werden. `str` würde in diesem Fall keine Kopie von `p` darstellen, sondern nur auf dieselbe Speicheradresse verweisen. Für eine genauere Betrachtung sei



auf die ähnliche Problematik im Fall des *Copy*-Konstruktors in diesem Abschnitt (Seite) verwiesen.

Konstruktoren können so wie alle Elemente als `public`, `protected` oder `private` deklariert werden. Durch die Einschränkung der Sichtbarkeit von Konstruktoren kann erreicht werden, daß Objekte nur mehr eingeschränkt angelegt werden können. Wird zum Beispiel der oben vorgestellte Konstruktor als `private` deklariert, so kann ein `TString` mit einem `char*`-Parameter nur innerhalb der Klasse (beziehungsweise innerhalb von `friend`) angelegt werden.

- Konstruktoren können - wie alle anderen Element-Funktionen auch - als `public`, `protected` oder `private` definiert werden. Damit wird ihre Verwendbarkeit und die Möglichkeit des Anlegens von Objekten eingeschränkt.

Um `TString`-Objekte mit *x* gleichen Zeichen vorbelegen zu können, wird zusätzlich folgender Konstruktor implementiert:

```
TString(int l, char fillChar=' ');

TString::TString(int l, char fillChar)
 : l(len)
{
  if (l==0) {
    str = 0;
  } else {
    str = new char[len+1];
    memset(str, fillChar, len);
    str[len] = (char)0;
  }
}
```

Der *Copy*-Konstruktor

Ein Konstruktor mit einem `const char*`-Parameter wurde bereits implementiert. Zusätzlich sollte es aber auch möglich sein, `TString`-Objekte mit anderen `TString`-Objekten zu initialisieren:

```
TString hStr1("Hello World!");
TString hStr2 = hStr1;  // Achtung - Initialisierung,
                        // keine Zuweisung!
TString hStr3(hStr1);   // wie oben - Initialisierung
                        // von hStr3 mit hStr1
```

Dies wird durch einen speziellen Konstruktor, den *Copy*-Konstruktor, der die Signatur `ClassName(const ClassName&)` hat, ermöglicht.

Copy-Konstruktoren werden automatisch aufgerufen, wenn

- ein Objekt mit einem anderen Objekt derselben Klasse initialisiert wird,
- ein Objekt als Wertparameter an eine Funktion übergeben wird (nicht aber bei Referenzparametern) oder
- ein Objekt als Resultat einer Funktion zurückgegeben wird (ebenfalls nicht bei Referenz-Rückgabewerten).

Wird für eine Klasse kein *Copy*-Konstruktor implementiert, so erzeugt das C++-Entwicklungssystem einen sogenannten ,,Standard *Copy*-Konstruktor``, der ein Objekt durch die Zuweisung aller Datenelemente kopiert (*Memberwise Assignment*).

Für die `TString`-Klasse würde der Standard-*Copy*-Konstruktor ungefähr so aussehen:

```
TString::TString(const TString& s)
 : len(s.len), str(s.str)
{}
```

Das führt im Fall der `TString`-Klasse allerdings dazu, daß der `str`-Zeiger von `helloStr2` auf denselben Speicherplatz verweist wie der `str`-Zeiger des Objekts `helloStr1`. Abbildung 11.2 illustriert ein Problem mit dem Standard-*Copy*-Konstruktor:

- Die erste Zeile der Abbildung zeigt die Ausgangssituation: zwei Objekte, von denen das eine noch nicht initialisiert ist.
- Zeile zwei zeigt die Situation *nach* der Initialisierung durch den Standard-*Copy*-Konstruktor - die Datenelemente beider Objekte haben dieselben Werte.
- Zeile drei zeigt die beiden Objekte, nachdem die Zeichenkette des linken Objekts deallokiert wurde: Das rechte Objekt verweist auf einen nicht mehr allokierten Speicherbereich.

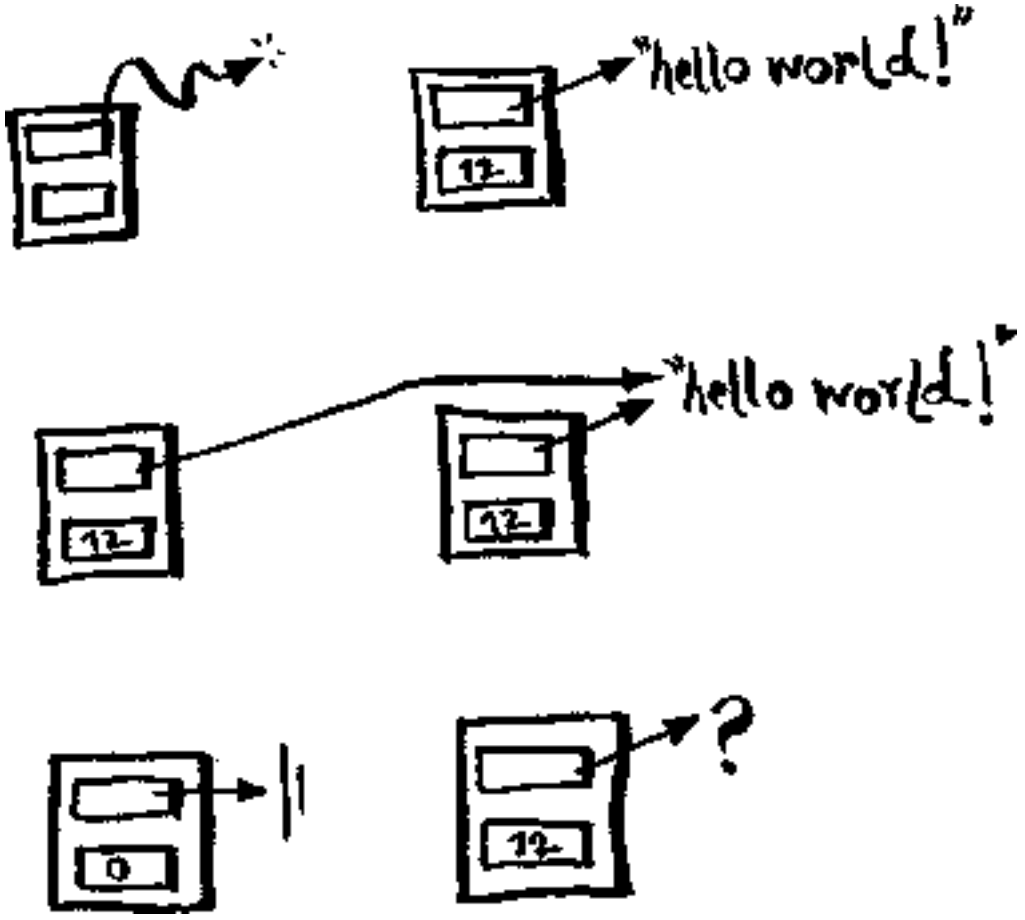


Abbildung 11.2: Problematik des Standard-*Copy*-Konstruktors bei dynamischen Datenstrukturen

Der Grund für das ,,Fehlverhalten`` liegt darin, daß der vom C++-System standardmäßig erzeugte *Copy*-Konstruktor nur ein sogenanntes *Shallow Copy* von Objekten durchführt. Das heißt, alle Datenelemente eines Objekts werden zugewiesen, etwaige Speicherbereiche, auf die im Fall von Zeigern verwiesen wird, aber nicht kopiert. Nach der Ausführung des *Copy*-Konstruktors verweisen die `str`-Zeiger der beiden involvierten Objekte auf *denselben* Speicherbereich.

Wird die `str`-Komponente eines der beiden Objekte deallokiert, so treten Probleme in Form von *Dangling Pointers* auf: ein `str`-Zeiger verweist auf einen Speicherbereich, der bereits deallokiert wurde.

,,Echte`` Kopien werden im Gegensatz zu den oben angeführten *Shallow Copies* als *Deep Copies* bezeichnet. Der Name rührt daher, daß nicht nur alle Elemente einer Ebene des Objekts kopiert werden (= flache Kopie), sondern auch alle Elemente und die Speicherbereiche auf die verwiesen wird.

Abbildung 11.3 zeigt das ,,richtige`` (= gewünschte) Verhalten des *Copy*-Konstruktors:

`s.len` kann einfach zugewiesen werden, `str` hingegen ist ein Zeiger auf eine Zeichenkette. Diese Zeichenkette muß *kopiert* werden, wenn die *Copy*-Konstruktor-Semantik erhalten bleiben soll. Zunächst wird daher Speicherplatz für `str` angefordert, dann wird `s.str` zeichenweise nach `str` kopiert.

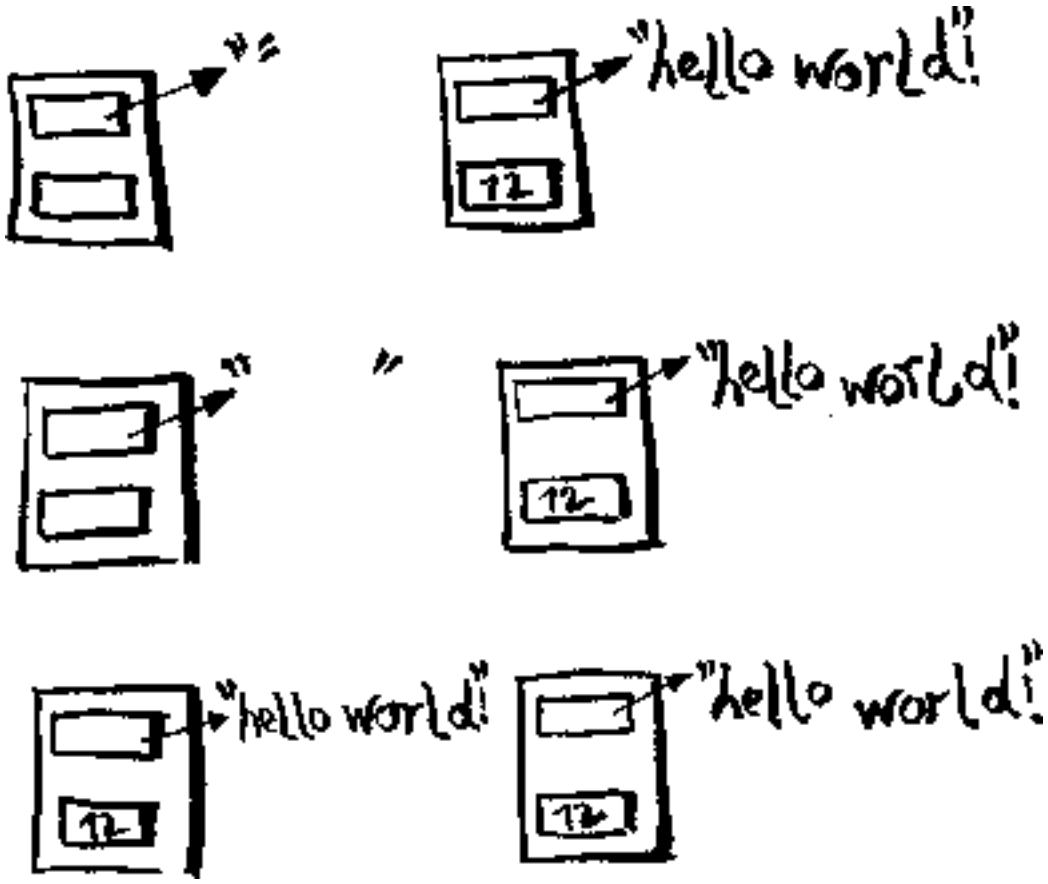


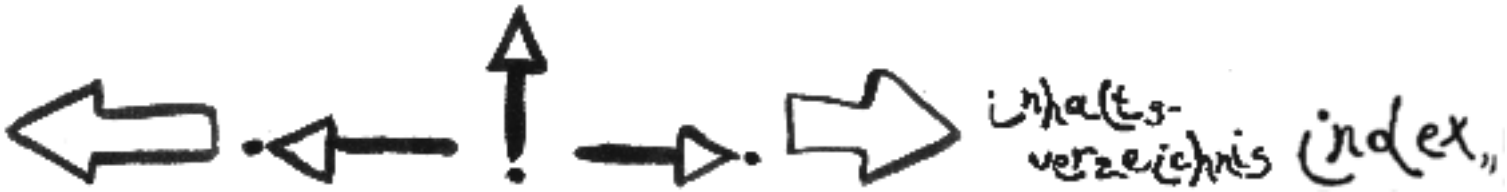
Abbildung 11.3: Richtiges Verhalten des *Copy*-Konstruktors bei dynamischen Datenstrukturen

```
class TString
{
  TString(const TString& s); // Copy-Konstruktor
  ...
};

TString::TString(const TString& s)
{
  len = s.len;
  if (s.str==0)  {
    str = 0;
  } else {
    str = new char[len+1];
    memcpy(str, s.str, len+1);
  }
}
```

Mit diesem *Copy*-Konstruktor können echte Kopien von `Tring`-Objekten hergestellt werden, der Konstruktor führt ein sogenanntes *Deep Copy* durch.

Im allgemeinen Fall reicht der Standard-*Copy*-Konstruktor durchaus aus. Enthalten Objekte aber dynamische Datenstrukturen, so muß der *Copy*-Konstruktor neu implementiert werden, so daß die dynamischen Datenstrukturen der Objekte kopiert (und nicht nur ihre Adressen zugewiesen) werden.

Für Klassen mit dynamischen Datenstrukturen muß ein *Copy*-Konstruktor definiert werden, da der vom System erzeugte *Copy*-Konstruktor lediglich ein *Shallow Copy* durchführt.



Vorige Seite: 11.7 Spezielle Element-Funktionen und **Eine Ebene höher:** 11.7 Spezielle Element-Funktionen und **Nächste Seite:** 11.7.2 Destruktoren

Vorige Seite: 11.7.1 Konstruktoren **Eine Ebene höher:** 11.7 Spezielle Element-Funktionen und **Nächste Seite:** 11.7.3 Überladen von Operatoren

---

11.7.2 Destruktoren

Destruktoren stellen das „Gegenteil`` von Konstruktoren dar. Mit ihnen werden Objekte „de-initialisiert``. Destruktoren sind Funktionen ohne Parameter und Rückgabe und haben den Namen der Klasse mit einer vorangestellten Tilde (~).

Destruktoren werden vom System immer dann (automatisch) aufgerufen, wenn Objekte zerstört werden. `auto`-Objekte werden zerstört, wenn der Gültigkeitsbereich, in dem sie vereinbart sind, verlassen wird. `static`-Objekte leben von ihrer ersten Verwendung an bis ans Programmende. Mit `new` angelegte Objekte müssen explizit durch den Aufruf des `delete`-Operators freigegeben werden.

Prinzipiell gilt, daß in Destruktoren all jene Speicherbereiche freigegeben werden müssen, für die im entsprechenden Konstruktor oder anderen Element-Funktionen mit `new` Speicherplatz explizit angefordert wurde.

In unserem Beispiel wurde im Konstruktor für das Element `str` Speicherplatz reserviert. Dieser Speicherplatz muß beim Zerstören des Objekts auch wieder freigegeben werden.

Ferner muß beachtet werden, daß Speicherbereiche, die mit `new` angefordert wurden, mit `delete` freigegeben werden müssen. Entsprechendes gilt für `new[]` und `delete[]`.

Alle mit `new` beziehungsweise `new[]` allokierten Speicherbereiche eines Objekts müssen im Destruktor mit `delete` beziehungsweise `delete[]` freigegeben werden.

Damit sieht der zugehörige Destruktor also so aus:

```
class TString
{
  virtual ~TString();
};
TString::~TString()
{
  delete [] str;
}
```

Anmerkungen:

- Destruktoren werden prinzipiell als *virtual* vereinbart. Der Grund dafür wird später erläutert (Kapitel 14).
- `delete[] str` kann in jedem Fall ausgeführt werden, da der Operator `delete` auch mit einem 0-Wert als Parameter aufgerufen werden kann.
- `len` muß nicht (und kann auch nicht) freigegeben werden: Für dieses Element wurde kein Speicherplatz explizit angefordert und die Freigabe der Instanzvariablen selbst wird vom System übernommen.

Konstruktoren und Destruktoren sind Initialisierungs- beziehungsweise De-Initialisierungsfunktionen. Sie sind *nicht* verantwortlich für das Anlegen beziehungsweise Freigeben des Objekts selbst beziehungsweise dessen Instanzvariablen. Das wird vom C++-System übernommen.

Beim Anlegen von Objekten (Variablen von Klassentypen) werden automatisch die entsprechenden Konstruktoren aufgerufen. Ebenso werden beim Zerstören die jeweiligen Destruktoren aktiviert:

```
#include "string.h"

void foo();

void foo()
{
  TString   s("Hallo");  // autom. Aufruf von
                         // TString::TString(const char*)
  TString*  pS1;
  TString*  pS2;
  pS1 = new TString;          // autom. Aufruf von
                              // TString::TString()
  pS2 = new TString("Hello"); // autom. Aufruf von
                              // TString::TString(const char*)
  ...
  delete pS1;            // autom. Aufruf von
                         // TString::TString() fuer *pS1
}                        // Zerstoerung von s =>
                         // TString::TString()
main()
{
   foo();
}
```

Im obigen Beispiel wird automatisch beim Betreten der Funktion `foo` der Konstruktor für `s` aufgerufen (`TString(const char*)`). Damit werden die Datenelemente von `s` initialisiert und `s` enthält jetzt

eine Kopie von „`Hallo`“.

Für `pS1` und `pS2` werden keine Konstruktoren automatisch aufgerufen, da es sich um Zeiger auf einen `TString` handelt und daher noch kein eigentliches Objekt angelegt wird. Der Aufruf der entsprechenden Konstruktoren erfolgt bei der Erzeugung der beiden Objekte durch `new`. Dabei wird `pS1` mit dem Leerstring initialisiert, `pS2` enthält „`Hello`“.

`pS1` wird vor dem Verlassen der Funktion mit `delete` deallokiert und der entsprechende Destruktor wird vom System automatisch aufgerufen. `s` wird ungültig, sobald `foo` verlassen und der Destruktor automatisch aufgerufen wird. Da `*pS2` nicht explizit freigegeben wird, bleibt dieser Speicherplatz erhalten. Er kann aber nicht mehr referenziert werden, da der Zeiger `pS2` beim Verlassen von `foo` zerstört wird und die Adresse damit verloren geht!

Vorige Seite: 11.7.1 Konstruktoren **Eine Ebene höher:** 11.7 Spezielle Element-Funktionen und **Nächste Seite:** 11.7.3 Überladen von Operatoren

Vorige Seite: 11.7.2 Destruktoren **Eine Ebene höher:** 11.7 Spezielle Element-Funktionen und **Nächste Seite:** 11.7.4 Automatisch generierte Element-Funktionen

---

Teilabschnitte

11.7.3 Überladen von Operatoren

In C++ besteht die Möglichkeit, die meisten vordefinierten Operatoren in der gleichen Art wie Funktionen zu überladen. So kann der Additionsoperator + beispielsweise derart überladen werden, daß er auch auf Matrizen, komplexe Zahlen oder auch *Strings* anwendbar ist.

Überladen von Operatoren hat gerade im Zusammenhang mit neuen Datentypen große Bedeutung. Auf diese Art können die neuen Typen nahtlos in das bestehende C++-System integriert werden: Variablen der neuen Typen werden mit << ausgegeben, mit >> eingelesen, mit = zugewiesen etc. Die genaue Besprechung des Überladens von Operatoren erfolgt daher hier im Zusammenhang mit Klassen anhand von konkreten Beispielen.

Allgemeines zum Überladen von Operatoren

Es können all jene Operatoren überladen werden, die durch sogenannte *Operator-Funktionen* implementiert sind. Eine Operator-Funktion ist eine Funktion mit einem speziellen Namen (Schlüsselwort `operator` gefolgt vom eigentlichen Operator-Symbol), die vom C++-System aufgerufen wird, um das Ergebnis eines entsprechenden Ausdrucks festzustellen. Tabelle 11.1 zeigt die Operatoren, die über Operator-Funktionen implementiert werden.

Anmerkung: `+`, `-`, `*` und `&` sind sowohl in ihrer unären als auch in ihrer binären Form über Operator-Funktionen implementiert.

```
+        -    *    /    &    ^    |      ~
!        =    <    >    +=   -=   *=     /=
%=       ^=   &=   |=   <<   >>   <<=    >>=
==       !=   <=   >=   &&   ||   ++     --
,        ->*  ->   ()   []   new  delete new[]
delete[]
```

Tabelle 11.1: Operator-Funktionen [ISO95, over.oper]

Die in Tabelle 11.1 angegebenen Operatoren können sowohl in der (üblichen) Operator-Notation, als auch in der Funktionsnotation verwendet werden. Der Ausdruck

```
complex1 + complex2
```

enthält einen +-Operator für den Datentyp `Complex` und entspricht folgender Anweisung:

```
complex1.operator+(complex2)
```

Überladene Operatoren werden wie normale Funktionen definiert, wobei dem Funktionsnamen (= *Operator Symbol*) das Schlüsselwort `operator` vorangestellt wird.

Zu beachten ist, daß überladene Operatoren die gleiche Anzahl von Argumenten haben müssen wie der ursprüngliche Operator und daß die Priorität des überladenen Operators ebenfalls gleich der des überladenen ist. So bleibt ein unärer Operator immer unär und `*` besitzt immer eine höhere Priorität als `+`. Außerdem können lediglich die vom System vorgegebenen Operatoren überladen werden.

Neue Operator-Symbole (zum Beispiel `**`) einzuführen ist ebenfalls unzulässig. Auch dürfen Operatoren keine *Default*-Argumente enthalten.

Operatoren können grundsätzlich auf zwei Arten implementiert werden: Als Element-Funktionen oder als `friend`s. So kann zum Beispiel der Operator `<` (= Kleiner-Test zweier `TString`-Objekte) folgendermaßen implementiert werden:

(1) Als Element-Funktion

```
class TString
{
  ...
  int operator <(const TString& s);
  ...
};

int TString::operator <(const TString& s)
{
  if (s.len==0) return 0;
  if (len==0) return 1;
  return (strcmp(str, s.str)<0);
}
```

(2) Als `friend`

```
class TString
{
  ...
  friend int operator <(const TString& s1, const TString& s2);
  ...
};

int operator <(const TString& s1, const TString& s2)
{
  if (s2.len==0) return 0;
  if (s1.len==0) return 1;
  return (strcmp(s1.str,s2.str)<0);
}
```

Im Prinzip bleibt es dem Programmierer überlassen, ob Operatoren als Methoden oder als `friend`-Funktionen implementiert werden. In einigen Fällen gibt es aber keine Wahlmöglichkeit. Element-Funktionen müssen als linkes Argument ein Objekt ihrer Klasse haben (Methoden können nur an Objekte der zugehörigen Klasse gesandt werden). In Fällen, wo das nicht möglich ist, muß der Operator daher als `friend` implementiert werden.

Ein Beispiel dafür ist etwa die Implementierung eines Operators < mit den Parametern `const char*` und `TString`. Dieser Operator vergleicht eine Zeichenkette (`char*`) mit einem Objekt der Klasse `TString`. Dieser Operator kann nicht als Methode implementiert werden, da die Methode (`<`) in diesem Fall an ein `char*`-*Objekt* geschickt würde. Aus diesem Grund ist in einigen Fällen die Implementierung als `friend`-Funktion einer Implementierung als Element-Funktion vorzuziehen.

- Zuweisungs-, Index-, Funktionsaufruf-, und Zeigeroperator (=, [], (), ->) müssen immer als Element-Funktionen implementiert werden.

Um eine unnötige Anzahl an Operatoren zu vermeiden, kann auch auf benutzerdefinierte Typumwandlungen zurückgegriffen werden. So ist es nicht nötig, die drei Operatoren

```
(1) int TString::operator<(const TString& s);
(2) int TString::operator<(const char* p);
(3) int operator<(const char* p, const TString& s);
```

zu implementieren, wenn ein entsprechender Konstruktor vorhanden ist, der es dem C++-System ermöglicht, aus `char*`-Variablen automatisch entsprechende `TString`-Objekte zu erzeugen (siehe Abschnitt 11.7.5). Daher kann der Operator (2) entfallen.

(3) kann nicht einfach entfallen, da das linke Argument einer Element-Funktion ein Objekt der eigenen Klasse sein muß. Wird aber (1) als `friend` implementiert, so kann auch (3) entfallen. Implementiert werden muß lediglich der Operator

```
(4) int operator<(const TString& s1, const TString& s2);
```

Implementierung von speziellen Operatoren

Für die Klasse `TString` sollen folgende Operatoren definiert werden:

- Indexoperator `[]`
- Operator `()`
- Zuweisungsoperator `=`
- Typumwandlungsoperator nach `char*`
- Vergleichsoperatoren `==`, `!=`, `<`, `>`
- Verkettungsoperatoren `+`, `+=`

Die Realisierung der Ein-/Ausgabeoperatoren `<<`, `>>` wird anhand eines anderen Beispiels (Klasse `Stack`) gezeigt.

Im folgenden ist die Realisierung der wichtigsten Operatoren angeführt. Nicht alle der `TString`-Operatoren sind hier im Detail besprochen, die Realisierbarkeit dieser kann aber nach dem Studium dieses Abschnitts als gegeben angenommen werden.

Der Indexoperator []

Auf Vektoren kann in C++ der Indexoperator `[]` angewandt werden. Damit ist der Zugriff auf ein einzelnes Element möglich. Auch für unsere Klasse `TString` soll dieser Operator vereinbart werden. Ein erster Versuch der Implementierung führt zu folgendem Ergebnis:

```
char TString::operator[](int elem)
{
  assert((elem>=0) && (elem<len));
  return str[elem];
}
```

Anmerkung: Das Makro `assert` beendet das Programm, falls die dahinter angegebene Bedingung nicht wahr ist (siehe Anhang B).

In der oben angeführten Implementierung werden Indexfehler erkannt, und der Zugriff auf ein einzelnes Zeichen eines TStrings ist möglich:

```
TString    s("Hello World!");
cout << "Erster Buchstabe:" << s[0] <<"\n";
```

Es ist allerdings unmöglich, das Ergebnis des Operatoraufrufs auf der linken Seite eines Ausdrucks zu verwenden:

```
TString    s("xello World!");
s[0] = 'H';
```

Dazu darf der Operator [] kein neues `char`-Objekt zurückgeben, sondern nur eine *Referenz* auf ein Zeichen des *Strings*. Der oben angeführte Operator muß also abgeändert werden:

```
char& TString::operator[](int elem)
{
  assert ((elem>=0) && (elem<len));
  return str[elem];
}
```

Jetzt kann das Ergebnis des Operator-Ausdrucks auch auf der linken Seite eines Ausdrucks verwendet werden.

```
TString  s("Hello");

cout << s[0];   // OK, Lesen eines String-Elements
s[0] = 'A';     // OK, Schreiben eines String-Elements
```

Die Vereinbarung von `[]` kann aber zu Fehlern führen:

```
const TString    sc("Hello const");

cout << sc[0];  // Fehler! [] ist nur fuer non-const-
                // Objekte definiert
```

Der Grund dafür liegt darin, daß der Operator `[]` in dieser Form nur auf non-`const`-Objekte angewandt werden kann. Um auch `const`-Objekte indizieren zu können, ist es nötig, einen zweiten Operator einzuführen. Dieser arbeitet auch auf `const`-Objekten und gibt eine `const`-Referenz zurück:

```
const char& operator[](int) const;

const char& TString::operator[](int elem) const
{
  assert((elem>0) && (elem<=len));
  return str[elem];
}
```

Jetzt können auch `const`-Objekte mit dem `[]`-Operator indiziert werden. Das Verändern von Zeichen eines `const-TString`-Objekts ist nach wie vor nicht möglich. Die Anweisung `sc[0]= 'A';` führt also richtigerweise zu einem Übersetzungsfehler.

Der Operator ()

Der Operator `()` soll in unserem Fall dazu verwendet werden, eine Art *SubString*-Funktion zu implementieren. `()` soll einen Teil der ursprünglichen Zeichenkette - spezifiziert durch Startposition und Länge - in einem neuen `TString`-Objekt zurückgeben. Die Realisierung beschränkt sich auf einen Aufruf eines Konstruktors:

```
TString operator()(int pos, int cnt) const;

TString TString::operator()(int pos, int cnt) const
{
  return TString(str+pos, cnt);
}
```

Anmerkung: Der Operator `()` gibt keine Referenz zurück. Der Rückgabewert muß ein neues Objekt sein und darf keinen Verweis auf ein bestehendes darstellen.

Die Operatoren new, delete

Auch new und delete können überladen werden. Damit kann zum Beispiel eine ,,Debug``-Version der beiden Operatoren implementiert werden, die beim Anlegen beziehungsweise Freigeben von Objekten Prüfungen durchführt und Debug-Meldungen mitprotokolliert. new kann beispielsweise so verändert werden, daß vor und nach dem allokierten Speicherbereich bestimmte Bytefolgen abgelegt werden. Beim Deallokieren von Objekten prüft delete dann, ob diese Folgen verändert wurden. Damit können Speicherfehler leichter gefunden werden.

Dabei ist zu beachten, daß neben den verschiedenen Arten von new- beziehungsweise delete-Operatoren (Abschnitt 8.2.4) auch der Ort der Deklaration von Bedeutung ist:

- Die globalen Operatoren new und delete (beziehungsweise new[] und delete[]) sind im Namensraum std deklariert. Sie sind vordefiniert und können überschrieben werden. Werden sie überschrieben, so sind die vordefinierten Versionen allerdings verborgen!

  Die globalen Operatoren new und delete (ob vordefiniert oder neu implementiert) werden immer dann verwendet, wenn Basisdatentypen-Objekte oder Objekte von Klassen, die über keinen eigenen new-Operator verfügen, allokiert werden.
- Verfügt eine Klasse über ,,eigene`` new- oder delete-Operatoren (beziehungsweise new[] oder delete[]), so werden diese verwendet, wenn ein Objekt (beziehungsweise ein Vektor von Objekten) dieser Klasse angelegt oder zerstört wird.

Bei der Implementierung von new und delete (beziehungsweise new[] und delete[]) sind einige Dinge zu beachten:

- Wird new implementiert, so sollte auch delete implementiert werden.
- Ferner ist zu prüfen, ob das jeweilige Entwicklungssystem Vektoren von Objekten der Klasse tatsächlich über die Operatoren new[] und delete[] anlegt beziehungsweise zerstört. Gegebenenfalls sind auch diese zu implementieren.
- Bei der Implementierung von new und delete muß auch beachtet werden, daß abgeleitete Klassen (Kapitel 13) auf den Operator der Basisklasse zurückgreifen, falls dieser vorhanden ist.

Eine Implementierung von new und delete für TString wird hier nicht gezeigt, für entsprechende Beispiele sei auf die in Anhang E angeführte Literatur verwiesen.

Der Zuweisungsoperator =

Die folgenden Anweisungen vereinbaren zwei TString-Objekte a und b und weisen b an a zu:

```
TString     a("Hallo");
TString     b("Hello World!");
 ...
a = b;
```

Nach dieser Anweisung enthält a tatsächlich die Zeichenkette ,,a``, allerdings wurde die Zeichenkette nicht kopiert. Vielmehr verweist die Komponente str von a jetzt auf dieselbe Adresse wie die Komponente str von b. Der Grund dafür liegt darin, daß C++ im standardmäßig erzeugten Zuweisungsoperator ein *Shallow Copy* durchführt (vergleiche Abschnitt 11.7.1).

Abbildung 11.4 zeigt grafisch wie zwei Objekte (Zeile eins) zugewiesen werden (Ergebnis der Zuweisung in Zeile zwei) und das Löschen der Zeichenkette eines der beiden Objekte (Zeile drei). Wie im Fall von Standard-*Copy*-Konstruktoren (vergleiche Abbildung 11.2) bleibt auch hier ein Zeiger erhalten, der auf einen nicht mehr allokierten Speicherbereich verweist.

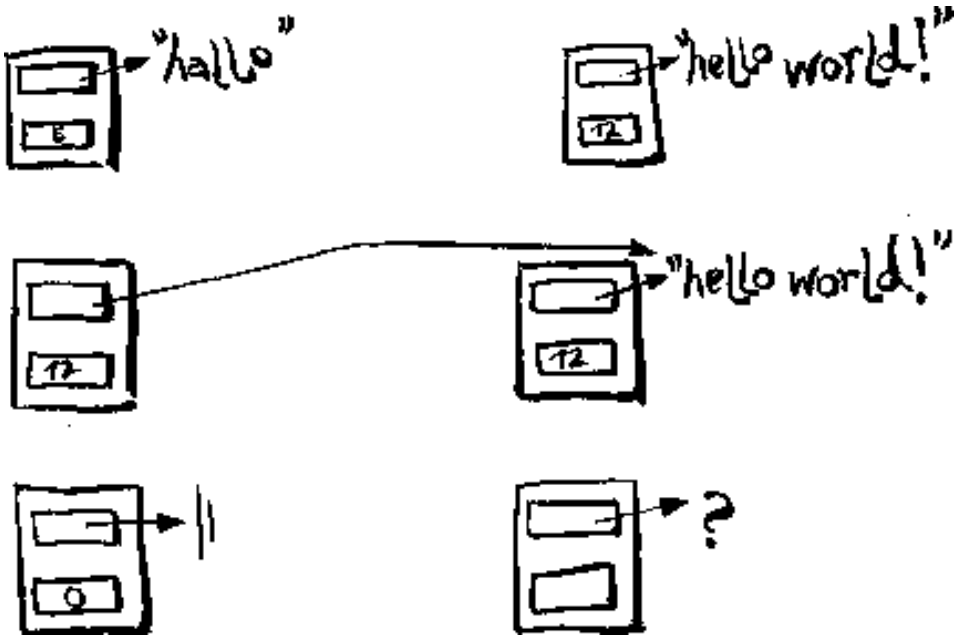


Abbildung 11.4: Problematik des Standard-Zuweisungsoperators anhand der Klasse `TString`

Um eine ,,echte`` Zuweisung, also ein *Deep Copy* , zu erreichen, muß ein eigener Zuweisungsoperator definiert werden.

Abbildung 11.5 zeigt das richtige Verhalten eines Zuweisungsoperators für Klassen mit dynamischen Datenstrukturen. Die einzelnen Zeilen der Abbildung im Detail:

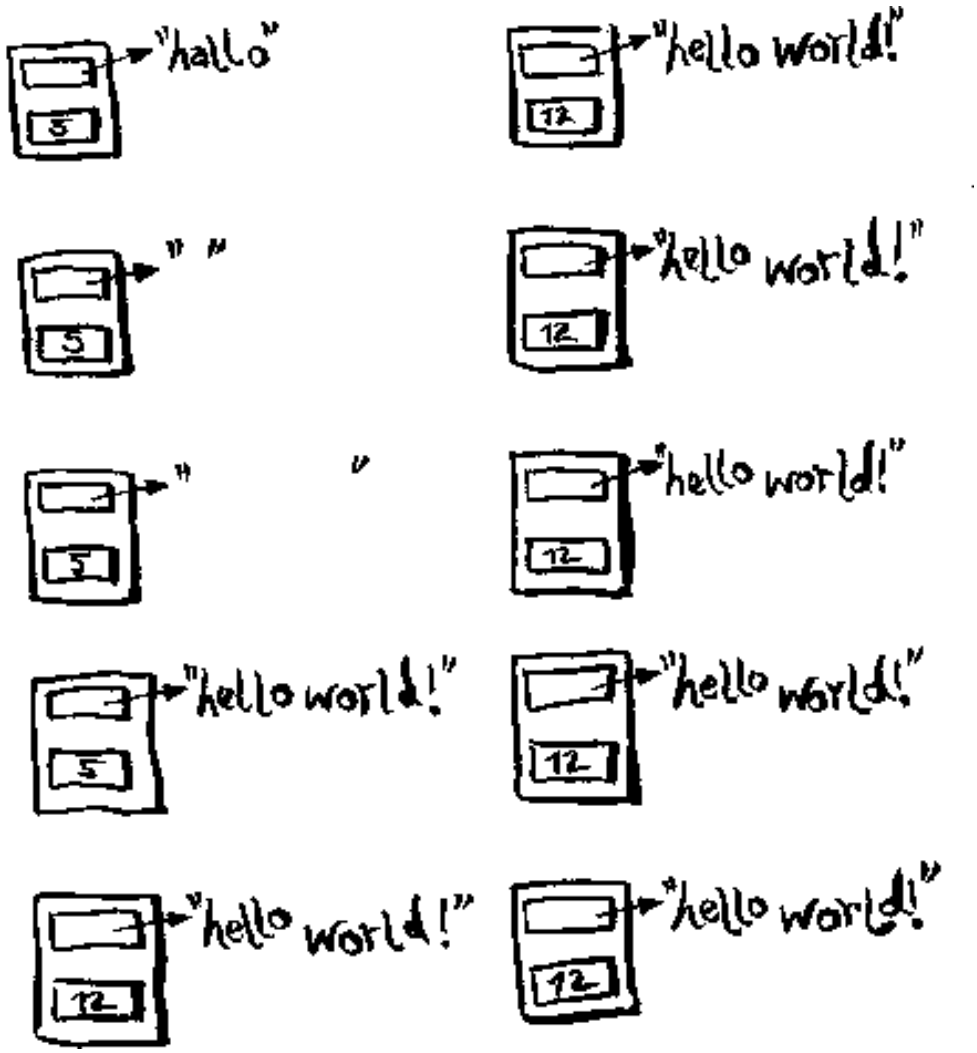


Abbildung 11.5: Das „richtige`` Verhalten des Zuweisungsoperators von `TString`

1. Als erstes muß geprüft werden, ob eine Zuweisung der Art `a = a` vorliegt. In diesem Fall dürfen keine Deallokierungen des Objekts auf der rechten Seite des Zuweisungsoperators erfolgen - sie würden das Objekt selbst zerstören. Eine Zuweisung eines Objekts an sich scheint auf den ersten Blick unsinnig, sie ist allerdings durchaus zulässig. Frei nach Murphy kann man davon ausgehen, daß das Vernachlässigen dieser Tatsache zu Problemen führen wird.
2. Die bereits allokierten Speicherbereiche des Objekts müssen de-allokiert werden.
3. Anschließend werden die Speicherbereiche mit der „richtigen`` Größe neu angefordert.
4. Dann werden die dynamisch allokierten Speicherbereiche kopiert.
5. Die restlichen Elemente werden zugewiesen.
6. Abschließend erfolgt die Rückgabe einer Referenz auf das eigene Objekt (`return *this`). Dies ist nötig, um Anweisungen der Art

   ```
   a=b=c=d;
   ```

   richtig abzuarbeiten.

Zwei Anmerkungen zum angeführten Schema:

- Die oben angeführte Art, auf Zuweisungen an sich selbst mittels Vergleich von Objektadressen zu prüfen, funktioniert nur in Verbindung mit Einfachvererbung! Bei Mehrfachvererbung (Kapitel 14) ist die Vorgangsweise wesentlich komplexer, siehe dazu [Mey92].
- Die Deallokierung „alter`` Speicherbereiche und ihre anschließende Neuanlage kann auch über eine sogennante „Re-Allokierung`` erfolgen. Dabei wird ein bereits allokierter Speicherbereich mit neuer Größe angelegt. Dem C++-System bleiben hier Möglichkeiten zur Effizienzverbesserung. Am grundsätzlichen Ablauf ändert dies allerdings nichts.

Der Zuweisungsoperator für `TString` sieht damit so aus:

```
TString& TString::operator=(const TString& s)
{
  if (this==&s) return *this; // Zuweisung der Art "a=a"
                              // abfangen
  delete [] str; str = 0;     // Alten Inhalt loeschen
  len = s.len;                // s.str nach s kopieren
  if (s.str != 0) {
    str = new char[len+1];
    memcpy(str, s.str,  len+1);
  }
  return *this;
}
```

Für Klassen mit dynamischen Datenstrukturen sollte neben dem *Copy*-Konstruktor stets auch der Zuweisungsoperator definiert werden.

- Bei der Implementierung von Zuweisungsoperatoren muß auch auf den speziellen Fall „`a = a`“ (Zuweisung eines Objekts an sich selbst) geachtet werden. Wichtig ist auch, daß Zuweisungsoperatoren eine Referenz auf das eigene Objekt zurückgeben.

Die Ein-/Ausgabeoperatoren >> und <<

Der Eingabeoperator >> und der Ausgabeoperator << können ebenfalls überladen werden. Damit ist es möglich, eigene Datentypen in bezug auf die Ein-/Ausgabe nahtlos in das C++-System zu integrieren: Alle Objekte, ob es sich nun um einfache Variablen (`int`, `char` etc.) oder um komplexe Objekte handelt, werden mit dem Operator >> eingelesen beziehungsweise mit << ausgegeben.

Anmerkung: Grundlegendes zu *Streams* von C++ ist in Anhang A angeführt.

Für die Klasse `TString` muß kein eigener Ein-/Ausgabeoperator vereinbart werden. Ein `TString` kann vom C++-System automatisch in eine Zeichenkette vom Typ `char*` umgewandelt werden und damit die entsprechenden Operatoren „mitbenutzen“ (siehe dazu Abschnitt 11.7.5).

Die Realisierung von Ein-/Ausgabeoperatoren soll daher an einem anderen Beispiel, der Klasse `Stack` aus Programm 11.7, gezeigt werden. Das Beispiel zeigt einen „sinnvollen“ Einsatz von `friend`-Beziehungen (mehr dazu unten).

Die Standard-Ausgabe soll um eine Möglichkeit erweitert werden, Objekte der Klasse `Stack` mit dem Operator << auszugeben.

Um Zugriff auf die `private`-Elemente eines `Stack`-Objekts zu haben, muß << ein Element dieser Klasse sein. Das bedeutet natürlich auch, daß << an das auszugebende Objekt geschickt wird:

```
myStack << cout;
```

Diese Möglichkeit ist sowohl verwirrend als auch unlogisch. Weshalb sollte die Methode *putTo* (Operator <<) an das auszugebende Objekt geschickt werden? Viel logischer ist es doch, diese Methode an den Ausgabestrom zu schicken und als Parameter das auszugebende Objekt zu übergeben. Das ist aber aufgrund des Zugriffsschutzes nicht möglich: „externe“ Funktionen/Element-Funktionen können nicht auf `private`- oder `protected`-Elemente der Klasse zugreifen.

Die Lösung hier lautet daher, den Operator << als eine „normale“ Funktion zu implementieren und diese als `friend` von `Stack` zu deklarieren. `friend`s von Klassen können auf alle Elemente, auch auf `private`-Elemente, zugreifen.

Der Ausgabeoperator << weist damit die Form

```
ostream& operator<<(ostream& s, const T& a);
```

auf. `ostream` steht dabei für einen Ausgabestrom, `T` für den Typ des zu lesenden Objekts.

Die Ausgabe erfolgt so, daß zunächst der Text `Stack` ausgegeben wird und dann die in < und > eingeschlossene Liste von `Stack`-Elementen:

Programm 11.10: Erweiterung von `Stack` um <<

```
class Stack
{
...
  friend ostream& operator<<(ostream& onTheOS,
                             const Stack& theStack);
...
};
...
ostream& operator<<(ostream& onTheOS, const Stack& theStack)
```

```
{
  Stack::Node*    p=theStack.topElem;
  onTheOS << "Stack: < ";
  while (p != 0) {
    onTheOS << p->data << " ";
    p = p->next;
  }
  onTheOS << ">";
  return onTheOS;
}
```

Die `friend`-Beziehung zwischen `Stack` und `<<` ist nötig, weil `<<` Internas von `Stack` verwendet. Das ist weder üblich noch guter Programmierstil (weshalb oben auch nur von einem ,,sinnvollen`` Einsatz von `friend` gesprochen wurde). Besser ist es, ein `public`-*Interface* zu vereinbaren, das auch den geordneten Zugriff auf alle Elemente eines Stacks zuläßt.

Der Eingabeoperator `>>` weist die Form

```
istream& operator>>(istream& s, T& a);
```

auf. `istream` steht für den Eingabestrom, der die Eingabedaten liefert, `T` für den Typ des zu lesenden Objekts. Der entsprechende Operator für die Klasse `Stack` liest zunächst die Anzahl der `Stack`-Werte ein und dann die einzelnen Werte. Sonderfälle und Status-Flags der *Stream*-Klasse werden hier *nicht* berücksichtigt!

Programm 11.11 erweitert die Klasse `Stack` aus Programm 11.7 um eine Eingabefunktion.

Programm 11.11: Erweiterung von `Stack` um `>>`

```
istream& operator>>(istream& fromTheIS, Stack& theStack)
{
  int stackElems;
  int dummy;

  theStack.removeAll();
  fromTheIS >> stackElems;
  for (int i=0; i<stackElems; ++i) {
    fromTheIS >> dummy;
    theStack.push(dummy);
  }
  return fromTheIS;
}
```

Anmerkung: In diesem Fall ist *keine* `friend`-Beziehung zwischen `Stack` und `>>` nötig. Die Operator-Funktion `>>` verwendet ausschließlich das `public`-*Interface* von `Stack` und benötigt keine Internas.

Vorige Seite: 11.7.2 Destruktoren **Eine Ebene höher:** 11.7 Spezielle Element-Funktionen und **Nächste Seite:** 11.7.4 Automatisch generierte Element-Funktionen

Vorige Seite: 11.7.3 Überladen von Operatoren **Eine Ebene höher:** 11.7 Spezielle Element-Funktionen und
Nächste Seite: 11.7.5 Benutzerdefinierte Typumwandlungen

11.7.4 Automatisch generierte Element-Funktionen und kanonische Form von Klassen

Wie bereits angeführt, erzeugt der C++-Compiler gewisse Element-Funktionen automatisch. Diese sind in Tabelle 11.2 aufgelistet:

Generierte Element-Funktion	Prototyp	Aktion
Default-Konstruktor	`T();`	-
Copy-Konstruktor	`T(const T&);`	Initialisiert Objekt mit Werten des Parameters (*Shallow Copy*)
Zuweisungsoperator	`T& operator=(const T&);`	Weist Datenelementen des Objekts jene des Parameters zu (*Shallow Copy*)
Adreßoperator	`T* operator&();` bzw. `const T* operator&() const;`	Gibt Adresse des Objekts zurück

Tabelle 11.2: Automatisch generierte Element-Funktion

Die in Tabelle 11.2 angeführten Element-Funktionen werden generiert, falls sie im Programm verwendet werden. Die Funktionen liefern nicht immer das gewünschte Resultat (beispielsweise wenn dynamische Datenstrukturen innerhalb einer Klasse verwendet werden) und können vom Programmierer selbst implementiert werden.

- Folgende Element-Funktionen werden vom Compiler generiert, wenn sie nicht vom Programmierer erstellt wurden: *Default*-Konstruktor, *Copy*-Konstruktor, Zuweisungs- und Adreßoperator.

Will man eine oder mehrere der oben angeführten Element-Funktionen nicht implementieren, und ihre (durch die automatische Generierung mögliche) Verwendung verbieten, so bleibt als Ausweg, sie als `private` zu deklarieren:

```
class TString
{
  ...
 private:
  TString& operator=(const TString& s);
  ...
};
```

Damit ist der Zuweisungsoperator für die Klasse `TString` als `private` deklariert und kann von außen nicht mehr benutzt werden.

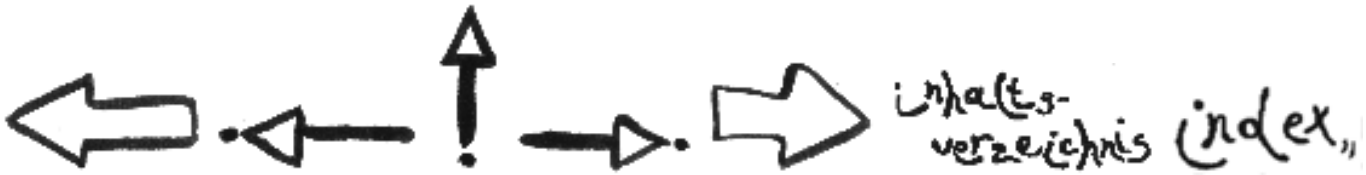
- Automatisch vom Compiler erzeugte Element-Funktionen können als `private` deklariert werden, um die Verwendung zu unterbinden.

Als *Kanonische Form* einer Klasse [Mey92, Cop92, Pap95] bezeichnet man jene Form, die es erlaubt, Objekte der Klasse *standardmäßig* (= so wie andere Objekte oder ,,normale`` Variablen) zu benutzen. Dazu müssen drei Bedingungen erfüllt sein:

- Ein korrekter *Default*-Konstruktor und (wenn nötig) verschiedene andere Konstruktoren müssen vorhanden sein.
- Enthält die Klasse dynamische Datenstrukturen, so sind der vordefinierte Zuweisungsoperator und der *Copy*-Konstruktor ,,nicht richtig`` und müssen neu implementiert werden.
- Ein - üblicherweise virtueller - Destruktor garantiert die korrekte De-Initialisierung von Objekten.

Nur wenn *alle* implementierten Klassen die kanonische Form aufweisen, können Objekte ohne Einschränkungen zugewiesen, kopiert, als Parameter übergeben und zerstört werden.

Die kanonische Form von Klassen gewährleistet eine einfache und sichere Verwendbarkeit von Klassen. Alle implementierten Klassen sollten diese Form aufweisen.



Vorige Seite: 11.7.3 Überladen von Operatoren **Eine Ebene höher:** 11.7 Spezielle Element-Funktionen und
Nächste Seite: 11.7.5 Benutzerdefinierte Typumwandlungen

Vorige Seite: 11.7.4 Automatisch generierte Element-Funktionen **Eine Ebene höher:** 11.7 Spezielle Element-Funktionen und

---

Teilabschnitte

11.7.5 Benutzerdefinierte Typumwandlungen

In C++ können für jede Klasse Typumwandlungen definiert werden. Diese Typumwandlungen stehen dem System und dem Benutzer dann zusätzlich zu den Standard-Typumwandlungen (vergleiche Abschnitt 9.4) zur Verfügung. Damit kann die Anzahl der für eine Klasse nötigen Funktionen/Operatoren drastisch reduziert werden.

Gegeben sei zum Beispiel eine Klasse `VeryLargeInt`, die sehr große ganze Zahlen aufnehmen kann. Diese Klasse enthält unter anderem die Operatoren `+`, `-`, `*` und `/`. Wünschenswert wäre es, daß Objekte der Klasse `VeryLargeInt` in beliebigen arithmetischen Ausdrücken - auch mit anderen Zahlen (`int`, `unsigned`, `long` etc.) - verwendet werden können.

Dazu müssen (nur für die Addition und Subtraktion mit `int`-Zahlen) zumindest folgende Operatoren implementiert werden:

```
class VeryLargeInt
{
  VeryLargeInt();
  virtual ~VeryLargeInt();
  ...
  friend VeryLargeInt operator+(const VeryLargeInt& v, int i);
  friend VeryLargeInt operator+(int i, const VeryLargeInt& v);
  friend VeryLargeInt operator-(const VeryLargeInt& v, int i);
  friend VeryLargeInt operator-(int i, const VeryLargeInt& v);
  ...
```

```
};
```

Werden auch noch die Operatoren `*`, `/`, `+=`, `-=`, `*=`, `/=` implementiert beziehungsweise auch noch andere Datentypen wie `float` oder `short` berücksichtigt, so steigt die Anzahl der benötigten Funktionen auf ein untragbares Maß.

Besser ist es, dem System von vornherein Möglichkeiten zu geben, `int`-Zahlen automatisch auf `VeryLargeInt`-Objekte zu konvertieren beziehungsweise umgekehrt. In diese Typumwandlungen können dann auch entsprechende Bereichsprüfungen eingebaut werden.

Dazu bestehen zwei Möglichkeiten: Typumwandlungen mittels Konstruktoren und Typumwandlungen mittels Umwandlungsoperatoren.

Typumwandlung mit Konstruktoren

Konstruktoren können dazu verwendet werden, Objekte einer bestimmten Klasse aus anderen Objekten oder Variablen von vordefinierten Datentypen zu erzeugen. So kann für die Klasse `VeryLargeInt` ein Konstruktor mit einem `int`-Argument definiert werden:

```
VeryLargeInt(int i);
```

Dieser Konstruktor initialisiert ein `VeryLargeInt`-Objekt mit dem angegebenen `int`-Argument. Der Compiler kann damit aus jeder `int`-Zahl automatisch ein entsprechendes `VeryLargeInt`-Objekt erzeugen. Für unsere Klasse `VeryLargeInt` bedeutet das, daß die Schnittstelle entsprechend kleiner wird: Es genügt nun, für die Addition, Subtraktion etc. einen einzigen Operator mit zwei Argumenten vom Typ `VeryLargeInt` zu implementieren. `int`-Zahlen können verwendet werden, da sie vom System automatisch mit dem entsprechenden Konstruktor in ein *temporäres* `VeryLargeInt`-Objekt umgewandelt werden.

In bezug auf die `TString`-Klasse wurde bereits ein derartiger Konstruktor definiert. Dieser Konstruktor kann `TString`-Objekte mit Werten vom Typ `char*` initialisieren.

In vielen Fällen ist es wohl erwünscht, verschiedene Konstruktoren zur Verfügung zu stellen, allerdings sollten diese *nicht* automatisch zur Typumwandlung verwendet werden. In diesem Fall muß der Konstruktor mit dem Schlüsselwort `explicit` gekennzeichnet werden. Das C++-System verwendet derartige Konstruktoren nicht, um Typumwandlungen durchzuführen:

```
explicit VeryLargeInt(int i); // Nur fuer Objekt-Konstruktion,
                              // nicht zur Typumwandlung!
```

`explicit` kann für sogenannte *Single-Argument*-Konstruktoren verwendet werden (Konstruktoren mit einem Parameter). Andere Operatoren können vom System nicht zur automatischen Typumwandlung herangezogen werden.

Typumwandlungen mit Umwandlungsoperatoren

Mit den verschiedenen Konstruktoren der Klasse `VeryLargeInt` ist es möglich, aus beliebigen Objekten ein `VeryLargeInt`-Objekt zu erzeugen. Es ist damit jedoch nicht möglich, ein `VeryLargeInt`-Objekt auf ein anderes Objekt zu konvertieren. Dazu müßte ein entsprechender Konstruktor in der jeweils anderen Klasse definiert werden, was nicht immer möglich ist.

Allerdings kann in der Klasse `VeryLargeInt` ein Umwandlungsoperator definiert werden, der als Ergebnis eine `int`-Variable liefert:

```
operator int ();
```

Dieser Operator wandelt ein Objekt der Klasse `VeryLargeInt` (falls möglich) in eine `int`-Zahl um. Das C++-System ruft diesen Operator immer dann automatisch auf, wenn eine `int`-Zahl erwartet wird und ein Objekt vom Typ `VeryLargeInt` vorhanden ist.

In bezug auf die Klasse `TString` kann eine Typumwandlung definiert werden, die `TString`-Objekte in `char*`-Zeiger umwandelt. Damit ist es möglich, `TString`-Objekte in Funktionen wie `memcpy`, `strcpy` zu verwenden. Allerdings verhält sich das Objekt dann wie ein ,,normaler`` `char*`-Zeiger - es werden keinerlei Prüfungen durchgeführt!

```
TString::operator char* () const
{
  return str;
}
```

Mit dieser Methode können TStrings synonym für `char*`-Zeiger verwendet werden:

```
char     cf[1024];
TString  s("Hello World!");
memcpy(cf, s, s.getLen());   // s wird automatisch in
                             // char* umgewandelt
```

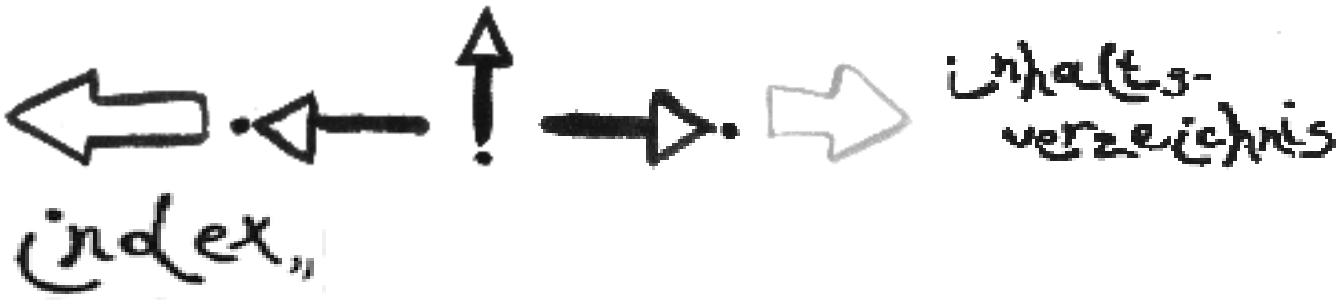
Typumwandlungen können auch explizit durch *Casts* ,,verlangt`` werden. Im Beispiel unten wird `s` durch den *Cast* in einen `char*` umgewandelt. In der zweiten Zeile wird für den Operator `+=` aber ein Argument der Klasse `TString` erwartet, also wird vom C++-System das Ergebnis des *Casts* wieder in einen `TString` konvertiert (durch den Konstruktor `TString(char*)`):

```
strcpy(cf, reinterpret_cast<char*>s); // Typumwandlung
s += reinterpret_cast<char*>s;        // erzwingen
```

Ebenso kann natürlich zum Beispiel ein Umwandlungsoperator der Form

```
operator int () const
```

implementiert werden, mit dem `TString`-Objekte automatisch in Zahlen umgewandelt werden können.



Vorige Seite: 11.7.4 Automatisch generierte Element-Funktionen **Eine Ebene höher:** 11.7 Spezielle Element-Funktionen und

Vorige Seite: Typumwandlungen mit Umwandlungsoperatoren **Eine Ebene höher:** 11 Das Klassenkonzept **Nächste Seite:** 11.8.1 Zeiger auf Klassen-Elemente

11.8 Weiterführende Themen

Vorige Seite: 11.8 Weiterführende Themen **Eine Ebene höher:** 11.8 Weiterführende Themen **Nächste Seite:** 11.8.2 Varianten

11.8.1 Zeiger auf Klassen-Elemente

Zeiger auf Klassen-Elemente unterscheiden sich von ,,normalen`` Zeigern. Klassen-Elemente sind Komponenten innerhalb eines Objekts und ihre Adressen sind Offsets zur Adresse der jeweiligen Objektinstanz. Zeiger auf Klassen-Elemente sind damit sowohl an den Datentyp, den sie referenzieren, gebunden, als auch an das Objekt, dem das referenzierte Element angehört.

In den folgenden Beispielen wird eine einfache Klasse `Node` vereinbart, anhand der die Zeiger auf Klassen-Elemente erklärt werden:

```
  // Schnittstelle
class Node
{
 public:
  int  getId();
  void setId(int theId);
  static int getNr();

  static int  nr;
  int         id;
};

                    // Implementierung:
int Node::nr = 0;   // static-Variable definieren kann
                    // auch in Schnittstelle erfolgen
int Node::getId()
{
  return id;
}

void Node::setId(int theId)
{
  id = theId;
}

static int Node::getNr()
```

```
{
  return nr;
}
```

Ein Zeiger auf ein `int`-Datenelement der Klasse `Node` wird wie folgt vereinbart:

```
  // Zeiger auf int-Element von Node
int Node::*pIntInNode;
```

`pIntInNode` ist wie bereits erwähnt ein Offset, sozusagen die Differenz zwischen der Basisadresse eines `Node`-Objekts und der Adresse eines `int`-Elements darin. Um mit einem derartigen Zeiger arbeiten zu können, sind daher ein konkretes Objekt sowie die Angabe, welches `int`-Element verwendet werden soll, nötig.

Im folgenden wird `pIntInNode` mit dem `int`-Element `id` der Klasse `Node` initialisiert. Verwendet wird dieser Offset dann im Kontext mit einem konkreten Objekt beziehungsweise einem Zeiger auf ein konkretes Objekt mit den sogenannten Element-Zeiger-Operatoren `.*` und `->*`.

```
Node  n;
Node* pN;
int   Node::*pIntInNode;
pN = &n;
pIntInNode = &Node::id;  // Zuweisung d. Offsets von
                         // id im Objekt Node
n.*pIntInNode = 10;      // Zuweisung v. 10 an n.id
                         // ueber Element-Zeiger
pN->*pIntInNode = 10;    // wie oben, nur ueber Zeiger
                         // auf Node-Objekt
```

Auch Zeiger auf Element-Funktionen sind an eine Klasse gebunden. Im Beispiel wird zunächst ein eigener Typ `PToSetIdFct` eingeführt. Variablen dieses Typs verweisen auf Element-Funktionen der Klasse `Node` mit einem `int`-Parameter und einem Rückgabewert `void`. Wiederum können Variablen dieses Zeigertyps nur in Verbindung mit Objekten verwendet werden.

```
  // Vereinbarung eines Typs "Zeiger auf Element-
  // Funktion v. Node, Return-Wert void, ein Parameter
  // vom Typ int"
typedef void  (Node::*PToSetIdFct)(int theId);

PToSetIdFct    pSetIdFct;

pSetIdFct = &Node::setId;  // Zuweisung des Offsets
                           // von setId in Node
(n.*pSetIdFct)(1);         // entspricht n.setId(1);
(pN->*pSetIdFct)(1);       // entspricht n.setId(1);
```

Statische Klassen-Elemente fallen nicht in dieses Schema, da sie in keinen Objekten einer Klasse abgelegt sind. Daher können statische Elemente auch ohne die Verwendung von Objekten referenziert werden:

```
  // Vereinbarung eines Zeigers auf ein static-
  // Datenelement:
  // Nicht: Node::*pInt2, da Zeiger auf stat. Element
int*     pInt2;
pInt2 = &Node::nr;
cout << "Nr: " << *pInt2 << "\n";

  // Vereinbarung eines Zeigers auf eine statische Funktion
typedef int (*PToStatFct)();
PToStatFct  pStatFct = &Node::getNr;

  // Dereferenzierung, kein Objekt noetig, da Zeiger auf
  // statische Fkt.
cout << "Ausgabe v. getNr: " << pStatFct() << "\n";
}
```

Vorige Seite: 11.8 Weiterführende Themen **Eine Ebene höher:** 11.8 Weiterführende Themen **Nächste Seite:** 11.8.2 Varianten

Vorige Seite: 11.8.1 Zeiger auf Klassen-Elemente **Eine Ebene höher:** 11.8 Weiterführende Themen
Nächste Seite: 11.8.3 Bitfelder

11.8.2 Varianten

`union` ist eine ,,Spezialität`` von C++, deren Nutzen im Bereich der objektorientierten Programmierung eher fraglich scheint. Lediglich für einige wenige, klar definierte Fälle scheint der Einsatz sinnvoll. `union` wird daher hier nur sehr kurz behandelt.

Eine `union` ist eine spezielle Klasse, die es erlaubt Speicherplatz einzusparen. Jede Instanz einer `union` verbraucht nur soviel Speicherplatz, wie nötig ist, um das größte Datenelement abzulegen. Dabei werden alle Datenelemente an derselben Speicheradresse abgelegt.

Ein Beispiel für den Einsatz einer `union` ist die Speicherung von Farbwerten wahlweise als Einzelwerte (`R`, `G`, `B`, `A`) und - aus Geschwindigkeitsgründen - als 32-Bit-Wort. Mit Hilfe einer `union` liegen beide Repräsentationen übereinander. Die Werte können wahlweise einzeln oder aber als ein Gesamtwert angesprochen werden:

```
union RGBA
{
 public:
  long getRGBA() const { return val; };
  char getR() const { return f[0]; };
  char getG() const { return f[1]; };
  char getB() const { return f[2]; };
  char getA() const { return f[3]; };
  void setRGBA(long l) { val = l; };
  void setR(char i) { f[0] = i; };
  void setG(char i) { f[1] = i; };
  void setB(char i) { f[2] = i; };
  void setA(char i) { f[3] = i; };
 private:
  long  val;   // 32-Bit-Wert, nicht portabel
  char  f[4];  // alternative Interpretation, 4 Bytes
};
```

Vorige Seite: 11.8.2 Varianten **Eine Ebene höher:** 11.8 Weiterführende Themen

11.8.3 Bitfelder

Bitfelder sind neben `union` eine weitere Möglichkeit, Speicherplatz einzusparen. Dabei wird ein Segment mit Integer-Größe in mehrere Datenelemente unterteilt, von denen jedes eine bestimmte Anzahl von Bits beschreibt. Im folgenden ist eine derartige Klasse definiert:

```
class FileState
{
 public:
  void setError();
  void clearError();
  int  wasError();
 private:
  unsigned int  error: 1;     // error belegt 1 bit
  unsigned int  modified: 1;  // modified belegt 1 bit
  unsigned int  openMode: 2;  // openMode belegt 2 bits
};
```

`error` und `modified` sind Variablen, die jeweils ein Bit belegen, `openMode` belegt 2 Bit und kann damit vier Zustände annehmen. Der Zugriff auf diese Elemente erfolgt ebenso wie der Zugriff auf Elemente einer ,,normalen`` Klasse. Der Vorteil von Bitfeldern liegt vor allem in der Möglichkeit der Hardware-Abstraktion.

Ebenso wie `union`'s erscheinen auch Bitfelder eher als C++-Spezifikum, das nur sehr begrenzt eingesetzt werden sollte.

Vorige Seite: 11.8.3 Bitfelder **Eine Ebene höher:** 11 Das Klassenkonzept **Nächste Seite:** 11.9.1 Klasse `Date`

11.9 Beispiele und Übungen

Die folgenden Übungsbeispiele sind nach aufsteigender Komplexität geordnet:

- Das erste Beispiel ist eine einfache Klasse `Date`, für die verschiedene Operatoren (Ein-/Ausgabe-, Vergleichsoperatoren) zu implementieren sind.
- Beispiel zwei zeigt eine einfache Anwendung von statischen Klassen-Elementen (Daten und Funktionen).
- Das nächste Beispiel ist ein „Theorie-Beispiel`` zum Themenbereich Gültigkeit und Sichtbarkeit.
- `Calculator` ist ein einfacher Rechner, der nach der umgekehrten polnischen Notation (UPN) arbeitet und aus zwei Klassen besteht.
- `List` ist eine *Collection*-Klasse, die ihre Elemente in einer verketteten Liste verwaltet. Themenbereiche des Beispiels sind die Problematik der kanonischen Form von Klassen sowie geschachtelte Klassen.
- `Rational` ist eine vollständige Klasse, die gebrochene Zahlen repräsentiert. `Rational` implementiert eine Reihe von Operatoren (`+`, `-`, `*`, `/`, Typumwandlungsoperator, `<<`, `>>`).

Vorige Seite: 11.9 Beispiele und Übungen **Eine Ebene höher:** 11.9 Beispiele und Übungen **Nächste Seite:** 11.9.2 Klasse `Counter`

Teilabschnitte

11.9.1 Klasse `Date`

Themenbereich

Einfache Klassen, Konstruktoren und Destruktoren, Vergleichsoperatoren, Ein-/Ausgabeoperatoren

Komplexität

Niedrig

Aufgabenstellung

Eine einfache Klasse `Date` ist zu implementieren. `Date` realisiert einen Datentyp zur Verwaltung von (Kalender-)Daten und enthält Instanzvariablen für Tag, Monat und Jahr. Die Daten sind verborgen, der Zugriff erfolgt über *Access*-Methoden. Ferner sind folgende Punkte zu beachten:

- Die Ein-/Ausgabeoperatoren >> und << sowie die verschiedenen Vergleichsoperatoren sind zu implementieren.
- Das Datum soll wahlweise im europäischen (Tag, Monat, Jahr) oder im amerikanischen Format (Monat, Tag, Jahr) ausgegeben und eingegeben werden können.

Vorige Seite: 11.9.1 Klasse `Date` **Eine Ebene höher:** 11.9 Beispiele und Übungen **Nächste Seite:** 11.9.3 Gültigkeitsbereich, Sichtbarkeit und

---

Teilabschnitte

11.9.2 Klasse `Counter`

Themenbereiche

Klassenvariablen

Komplexität

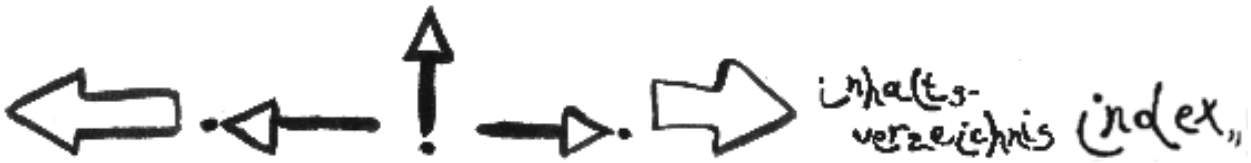
Niedrig

Aufgabenstellung

Zu implementieren ist eine Klasse `Counter`, die

- in einer Klassenvariable die Anzahl der existierenden `Counter`-Instanzen speichert und
- jedem Objekt eine eindeutige ,,Kennung`` zuordnet.

Die Kennung besteht aus einer (klassenweit) eindeutigen `int`-Zahl. Im Konstruktor/Destruktor ist eine Art ,,Debug-Protokoll`` zu realisieren, das entsprechende Meldungen beim Anlegen und Zerstören von Objekten ausgibt.



Vorige Seite: 11.9.2 Klasse `Counter` **Eine Ebene höher:** 11.9 Beispiele und Übungen **Nächste Seite:** 11.9.4 Klasse `Calculator`

Teilabschnitte

11.9.3 Gültigkeitsbereich, Sichtbarkeit und Lebensdauer

Themenbereiche

Gültigkeitsbereich, Sichtbarkeit, Lebensdauer

Komplexität

Mittel

Aufgabenstellung

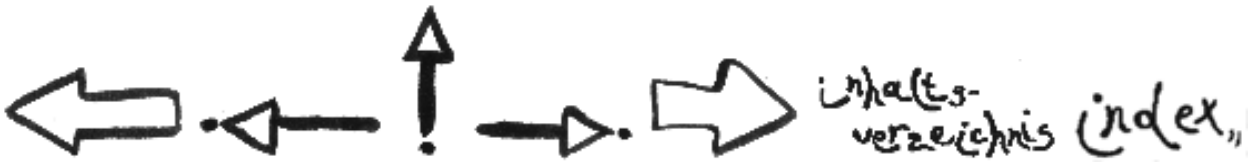
Um die Begriffe Gültigkeitsbereich, Sichtbarkeit und Lebensdauer zu illustrieren, ist ein Programmbeispiel angeführt, das eine Reihe von Ausgaben macht.

Stellen Sie fest, was das Programm ausgibt, und vergleichen Sie Ihr Ergebnis mit der Lösung!

Programm 11.12: Gültigkeitsbereich und Sichtbarkeit

```
#include <iostream>
int          a, b;

class A
{
 public:
  void foo(int b);
  static int a=12;
  int c;
};

int A::a;

void foo1(int a, int& b)
{
  int c;
  c = a; a = b; b = c+1;
}
```

```
void foo2(int& a, int& b)
{
  int c;
  {
    static int b;
    c = ::a+3; ::a = b; b = c;
  }
}

void foo3(int& a, int b)
{
  int c;
  c = A::a; A::a = 2*b; b = c;
}

void A::foo(int b)
{
  c = 12;
  a *= b++;
  foo1(a, a);
  cout << "A1: a = " << a << " b = " << b << endl;
  foo2(a, b);
  cout << "A2: a = " << a << " b = " << b << endl;
  foo3(b, b);
  cout << "A3: a = " << a << " b = " << b << endl;
}

void main()
{
  A    o;

  a = 2; b = 3;
  foo1(a, b);
  cout << "M1: a = " << a << " b = " << b << endl;
  foo2(a, a);
  cout << "M2: a = " << a << " b = " << b << endl;
  foo3(a, b);
  cout << "M3: a = " << a << " b = " << b << endl;
  o.foo(b);
  cout << "M4: a = " << a << " b = " << b << endl;
  o.foo(a);
  cout << "M5: a = " << a << " b = " << b << endl;
}
```



Vorige Seite: 11.9.3 Gültigkeitsbereich, Sichtbarkeit und **Eine Ebene höher:** 11.9 Beispiele und Übungen **Nächste Seite:** 11.9.5 Klasse `List`

Teilabschnitte

11.9.4 Klasse `Calculator`

Themenbereiche

Vollständige Klasse, Verwendung anderer Klassen

Komplexität

Niedrig

Aufgabenstellung

Ein einfacher Rechner, der mittels umgekehrter polnischer Notation (UPN) arbeitet, ist zu implementieren. Der Rechner beherrscht die Grundrechenarten, verfügt über einen Speicher und versteht einfache Ausdrücke, die nach folgender Grammatik aufgebaut sind:

```
CalcSeq:
                CalcSymbol
                CalcSeq CalcSymbol
CalcSymbol:
                Operator
                Number
                Fct
                Eos
Operator: one of
                + - / * n
Number:
                Digit
```

```
                Number Digit

Fct: one of

                S C R

Eos: one of

                 Leerzeichen =
```

Alle Eingaben bestehen aus derartigen Ausdrücken. Der Rechner selbst ist unabhängig von der jeweiligen Eingabe. Er stellt eine Methode zur Verfügung, mit der ein einzelnes Zeichen übergeben wird. Die eigentliche Eingabe (= das Lesen vom Eingabestrom) wird vom Hauptprogramm durchgeführt.

Der grundsätzliche Ablauf kann wie folgt beschrieben werden: Ein Symbol wird Zeichen für Zeichen eingelesen. Ist das gelesene Zeichen eine Ziffer, so muß diese vorerst in einem Puffer (eine Zeichenkette) zwischengespeichert werden. Jede Zahl besteht aus *x* Ziffern und wird durch das Auftreten eines Operators, eines Funktionssymbols oder eines *Eos*-Zeichens beendet.

Eos bezeichnet Zeichen ohne spezielle Bedeutung, sie trennen einzelne Zahlen. Die einzige Aktion, die daher bei ihrem Auftreten gesetzt werden muß, ist die Analyse der gespeicherten Zeichenkette (siehe unten).

Wird ein Operator eingegeben, so muß zunächst die abgelegte Zeichenkette analysiert werden: Es muß geprüft werden, ob die Zeichenkette eine gültige Zahl im Integer-Bereich enthält. Ist dies der Fall, so wird die Zeichenkette in eine Zahl umgewandelt und die Zahl auf den Stack gelegt. Dann wird der eigentliche Operator behandelt: Es gibt vier binäre (`+`, `-`, `*`, `/`) Operatoren sowie den unären Negationsoperator `n`. Je nach Operator werden zwei Operanden oder aber nur ein einzelner Operand vom Stack geholt. Die entsprechende Operation wird durchgeführt und das Ergebnis wieder auf den Stack gelegt.

Die Funktionssymbole des Rechners (`S`, `C`, `R`) bewirken ebenso wie die Operatoren eine Analyse der Zeichenkette und steuern den Rechner:

- `C` löscht den UPN-Stack, eventuelle Fehler-Flags, die gespeicherte Zahl und die abgelegte Zeichenkette.
- `S` speichert die oberste Zahl am Stack im internen Register.
- `R` holt die im Register abgelegte Zahl und legt sie auf den Stack.

Beim Berechnen der Ausdrücke können Fehler auftreten. So führt zum Beispiel der Ausdruck `2 3 * +` zu einem Fehler, da zuwenig Operanden vorliegen. Ein anderer Fehler ist etwa eine Division durch 0. Ist erst einmal ein Fehler aufgetreten, so führt der Rechner keine Operationen mehr durch. Durch die Eingabe des `C`-Symbols kann er allerdings in seinen Ausgangszustand zurückversetzt werden. Bereichsüberschreitungen wie sie zum Beispiel im Ausdruck 32767+2 auf einem 16-Bit-System auftreten, können vernachlässigt werden.

Die Schnittstelle ist in Programm 11.13 angegeben und weist folgende spezielle Methoden und Funktionen auf:

- `enterChar` übergibt ein gelesenes Zeichen an den Rechner.
- `getResult` liefert das aktuelle Ergebnis des im Rechner gespeicherten Ausdrucks.
- `wasError` gibt an, ob ein Fehler aufgetreten ist.
- `analyzeSym` ist eine Hilfsmethode, die das gelesene und in `buf` abgelegte Symbol analysiert und auf dem Stack `upnStack` legt.
- `clear` löscht den Stack und versetzt den Rechner in den Ausgangszustand.
- `store` und `recall` speichern ein Symbol beziehungsweise legen das gespeicherte Symbol auf den Stack.
- `addNum`, `subNum`, `mulNum`, `divNum` und `negateNum` implementieren die Addition, Subtraktion, Multiplikation, Division sowie die Negation.

Programm 11.13: Klasse `Calculator`, Schnittstelle

```
#ifndef calc__H
#define calc__H

  // Forward-Deklaration v. Stack
class Stack;
```

```
class Calculator
{
 public:
  Calculator();
  virtual ~Calculator();

  void enterChar(const char ch);

    // getResult liefert das aktuelle Ergebnis bzw. 0
    // falls ein Fehler aufgetreten ist
  int getResult();
    // wasError prueft, ob ein Fehler aufgetreten ist
  bool wasError();

 private:
    // analyzeSym analysiert das in buf gespeicherte Symbol
    // (eine Zahl in Textform), wandelt es in eine int-Zahl
    // um und legt es auf den Stack
  void analyzeSym();

    // clear loescht den UPN-Stack, Fehlerflag & Speicher
  void clear();

    // store speichert das aktuelle Ergebnis
  void store();

    // recall holt Ergebnis aus dem Speicher und legt es
    // auf den Stack
  void recall();

    // Arithmetische Operationen
  void addNum();
  void subNum();
  void mulNum();
  void divNum();
  void negateNum();

    // Konstante f. max. Zeichen pro Zahl
  static const int  MaxChars = 80;

  Stack    upnStack;          // UPN-Stack
  char     buf[MaxChars];     // Feld f. Zahl
  int      bufChars;
  int      memory;            // Gespeicherte Zahl
  bool     numStored;
  bool     calcErr;
};
#endif
```

Hintergrundwissen

In der UPN werden arithmetische Ausdrücke ,,umgekehrt`` notiert, das heißt zuerst die Operanden und dann der Operator. Der Ausdruck `2 3 +` entspricht damit dem Ausdruck `2 + 3` in der üblichen *Infix*-Notation.

Wesentlicher Vorteil dieser Notation ist das Wegfallen der Klammern. Einige Beispiele für UPN-Ausdrücke sind in Tabelle 11.3 angeführt.

UPN-Ausdruck	Infix
10 224 3 + *	10 * (224 + 3)
1 2 3 4 / * +	1 + (2 * (3 / 4))
30 14 18 2 / - /	30 / (14 - (18 / 2))

Tabelle 11.3: UPN-Beispiele

Vorige Seite: 11.9.3 Gültigkeitsbereich, Sichtbarkeit und **Eine Ebene höher:** 11.9 Beispiele und Übungen **Nächste Seite:** 11.9.5 Klasse `List`

Vorige Seite: 11.9.4 Klasse `Calculator` **Eine Ebene höher:** 11.9 Beispiele und Übungen **Nächste Seite:** 11.9.6 Klasse `Rational`

Teilabschnitte

11.9.5 Klasse `List`

Themenbereiche

Vollständige Klasse, dynamische Datenstrukturen, geschachtelte Klassen, Kanonische Form

Komplexität

Mittel

Aufgabenstellung

Eine Klasse `List` soll implementiert werden, die `int`-Elemente in einer einfach verketteten Liste verwaltet.

Eine einfach verkettete Liste besteht aus Elementen, die jeweils durch einen Zeiger verkettet sind. Jedes einzelne Element verweist auf das nächste Element. Die Elemente werden durch eine eigene Klasse `Node` dargestellt, die in `List` vereinbart ist.

`List` stellt folgende Zugriffsoperationen zur Verfügung:

- `add` fügt ein `int`-Element *am Ende* der Liste ein.
- `remove` entfernt das *erste* Element aus der Liste.
- `removeAll` entfernt alle Elemente der Liste.
- `wasError` gibt an, ob die letzte Operation fehlerhaft war.
- `isEmpty` gibt an, ob die Liste leer ist.

Zudem verfügen `List`-Objekte über eine Art ,,Gedächtnis`` und merken sich, welche Elemente zuletzt

gelesen wurden:

- `getFirst` liefert das erste Element der Liste, ohne es zu entfernen.
- `getNext` liefert das nächste Element nach dem zuletzt gelesenen.

Aufrufe von schreibenden Operationen (`add`, `remove`, `removeAll`) löschen das Gedächtnis. Ein Aufruf von `getNext` führt zu einem Fehler, wenn das ,,Gedächtnis`` nicht durch `getFirst` initialisiert wurde.

Vorige Seite: 11.9.4 Klasse `Calculator` **Eine Ebene höher:** 11.9 Beispiele und Übungen **Nächste Seite:** 11.9.6 Klasse `Rational`

Vorige Seite: 11.9.5 Klasse `List` **Eine Ebene höher:** 11.9 Beispiele und Übungen

---

Teilabschnitte

11.9.6 Klasse `Rational`

Themenbereiche

Vollständige Klasse, Ein-/Ausgabeoperator, Überladen von Operatoren, Typumwandlungsoperatoren

Komplexität

Mittel

Aufgabenstellung

Eine Klasse `Rational` ist zu implementieren. Die Klasse repräsentiert gebrochene Zahlen und implementiert die Operatoren `+`, `-`, `*`, `/`. Alle Grundrechenarten liefern als Ergebnis eine neue gebrochene Zahl. Die gebrochenen Zahlen müssen nach jeder Operation vereinfacht werden. Zusätzlich sind Operatoren zur Umwandlung in eine Zeichenkette beziehungsweise in eine Gleitkommazahl sowie die Ein-/Ausgabeoperatoren `>>` und `<<` zu codieren.

Bei der Eingabe ist zunächst ein `TString` (siehe Klasse `TString`) einzulesen und dieser anschließend in eine gebrochene Zahl umzuwandeln. Bei der Ausgabe ist ein führendes `+` im Fall von positiven Zahlen wegzulassen. Die Ein-/Ausgabe erfolgt nach der unten angegebenen Grammatik.

```
Rational:

Signopt IntNumber / IntNumber

IntNumber:

Digit

IntNumber Digit

Digit: one of

0 1 2 3 4 5 6 7 8 9

Sign: one of

+ -
```

Um unnötigen Codieraufwand zu ersparen, wird eine Datei `globs.h` verwendet. In dieser Datei sind Funktions-Templates für

die Operator-Funktionen <=, >= etc. abgelegt. Diese Operatoren führen alle Vergleiche auf die beiden Operatoren < und == zurück. Damit wird erreicht, daß in einer Klasse nur < und == definiert werden müssen, alle anderen Operatoren werden ,,automatisch generiert``.

Die Source-Codes für die Dateien sind im folgenden angeführt. Zu beachten ist, daß auch eventuelle Fehlerfälle (beim Einlesen, Rechnen etc.) zu berücksichtigen sind. Sollten zusätzliche Methoden benötigt werden, so können diese im `private`-Teil der Klasse vereinbart werden, der `public`-Teil darf nicht verändert werden.

Programm 11.14: Klasse `Rational`, Schnittstelle

```
#include <iostream>
#include "tstring.h"
#include "globs.h"

class Rational
{
 public:
  Rational(int n=0, int d=1);
  virtual ~Rational();

    // Accessors, inline-Funktionen!
  void set(int num, int denom=1) {
    num_d = num; denom_d = denom;
  };
  int num() const { return num_d; };
  int denom() const { return denom_d; };

    // *, / Operatoren, als Element-Funktionen realisiert
    // (das Ergebnis der Operationen ist vereinfacht)
  Rational operator*(const Rational& right) const;
  Rational operator/(const Rational& right) const;

    // Unaere Operatoren:
  Rational operator+() const;
  Rational operator-() const;

    // Typ-Umwandlung nach double
  operator double();

    // Typ-Umwandlung nach TString
  operator TString();

    // Gibt an, ob ein Fehler aufgetreten ist
  bool wasError();

 private:
    // Vereinfachen eines Bruchs
  void simplify();

  int    num_d;
  int    denom_d;
  bool   err_d;
};

  // Zwei Operatoren als "normale" Operator-Funktionen
```

```
Rational operator+(const Rational& left,
                   const Rational& right);
Rational operator-(const Rational& left,
                   const Rational& right);

  // Ein-/Ausgabeoperatoren
ostream& operator <<(ostream& os, const Rational& r);
istream& operator >>(istream& is, Rational& r);

  // Basis-Vergleichsoperatoren (==, <), alle anderen
  // werden durch Template-Funktionen realisiert
bool operator==(const Rational& left,
                const Rational& right);
bool operator<(const Rational& left,
               const Rational& right);
```

Programm 11.15: Datei `globs.h`

```
  // template functions:
template <class T>
inline bool operator!= (const T& left, const T& right)
{ return !(left == right); };

template <class T>
inline bool operator> (const T& left, const T& right)
{ return right < left; };

template <class T>
inline bool operator<= (const T& left, const T& right)
{ return !(right < left); };

template <class T>
inline bool operator>= (const T& left, const T& right)
{ return !(left < right); };
```

Vorige Seite: 11.9.5 Klasse `List` **Eine Ebene höher:** 11.9 Beispiele und Übungen

Vorige Seite: Aufgabenstellung **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** 12.1 Motivation

---

12 Templates

Der Template-Mechanismus von C++ erlaubt die Implementierung allgemeiner Klassen und Funktionen, die unabhängig von konkreten Datentypen arbeiten. Template-Klassen und -Funktionen können nicht nur mit konkreten Objekt-Werten (wie ,,normale`` Funktionen) parametrisiert werden, sondern auch mit Typen.

Dieses Kapitel beschreibt das Template-Konzept von C++ und ist wie folgt gegliedert: Nach einer kurzen Motivation wird die Realisierung von generischen Algorithmen (Funktions-Templates) gezeigt. Danach werden generische Klassen (Klassen-Templates) behandelt.

Vorige Seite: 12 Templates **Eine Ebene höher:** 12 Templates **Nächste Seite:** 12.2 Funktions-Templates

12.1 Motivation

In Kapitel 11 wurde eine Klasse `Stack` implementiert, die einen Kellerspeicher von `int`-Objekten darstellt. Wird ein `double`-Stack, ein `char`-Stack oder etwa ein `TString`-Stack benötigt, so muß der Source-Code der Klasse kopiert und abgeändert werden. Als Resultat erhält man eine Reihe von Klassen, die sich nur in den verwalteten Datentypen unterscheiden. Im Idealfall weisen die Klassen exakt dieselben Algorithmen auf. Allerdings ist nicht *eine* Version des Source-Codes zu verwalten, sondern *x* Ausführungen der (fast identischen) Klassen.

Ganz ähnlich verhält es sich mit Funktionen. Viele Funktionen unterscheiden sich lediglich in den Datentypen, auf denen sie operieren, weisen aber sonst exakt dieselben Aktionen auf. Ein Beispiel dafür ist etwa eine Funktion für eine binäre Suche in einem Vektor. Unabhängig vom Typ des Vektors ist der Algorithmus stets gleich, ebenso wie im Fall von Klassen müssen aber auch hier *x* verschiedene Versionen implementiert werden.

Als wesentliche Nachteile können angegeben werden:

- Die Vielzahl an möglichen Typen führt zu einer kaum zu verwaltenden Menge an (unnötigem) Code.
- Alle Klassen müssen unabhängig voneinander getestet werden.
- Wird in einer Klasse ein Fehler gefunden, so muß die Korrektur in den anderen Klassen nachgeführt werden. Werden derartige Änderungen unterlassen, so entstehen im Endeffekt verschiedene Versionen der an sich fast identischen Klassen.

All diese Nachteile können unter dem Schlagwort ,,Mißachtung des *Single Source*-Prinzips`` subsumiert werden und ergeben sich aus der Tatsache, daß verschiedene Source-Code-Versionen für ein und dieselbe Aufgabe existieren.

Mittels des Template-Konzepts können Klassen und Funktionen mit Datentypen parametrisiert und je nach Bedarf ausgeprägt werden. Wesentliche Vorteile von Templates sind:

- *Single Source*-Prinzip
  Für *x* Varianten derselben Datenstrukturen existiert genau eine Version des Source-Codes, der geändert und gewartet werden muß.
- Höhere Wiederverwendbarkeit
  Klassen-Templates sind bei geeigneter Wahl ihrer Parameter allgemein einsetzbar und einfach wiederverwendbar.
- Statische Bindung

Die Bindung zur Übersetzungszeit hat in bezug auf Typsicherheit und Fehlererkennung zweifellos große Vorteile gegenüber „generischen`` C-Lösungen mit `void*`-Zeigern, aber zum Teil auch gegenüber typisch „objektorientierten`` Varianten wie sie zum Beispiel in Smalltalk üblich sind.

Trotz dieser Vorteile können (und sollen) Templates Vererbung nicht ersetzen (siehe dazu auch [Mey88]) - vielmehr ergänzen sie die Palette der zur Verfügung stehenden Sprachkonzepte.

Vorige Seite: 12 Templates **Eine Ebene höher:** 12 Templates **Nächste Seite:** 12.2 Funktions-Templates

Vorige Seite: 12.1 Motivation **Eine Ebene höher:** 12 Templates **Nächste Seite:** 12.2.1 Ausprägung von Funktions-Templates

12.2 Funktions-Templates

Oft ist es nötig, ähnliche Funktionen für eine verschiedene Anzahl von Datentypen zu implementieren. Einfache Beispiele dafür sind etwa Funktionen zum Sortieren von Zahlenfeldern, zum Suchen von Minima oder Maxima oder das Vertauschen von Werten.

Immer dann wenn mehrere Funktionen sich nur in den Daten, auf denen sie operieren, unterscheiden, können sie in C++ durch eine generische Funktion realisiert werden. Generische Funktionen oder Funktions-Templates sind Funktionen, die mit Datentypen parametrisiert sind.

Diese Parametrisierung wird durch den Template-Mechanismus ermöglicht. Dazu wird dem Funktionsnamen das Schlüsselwort `template`, gefolgt von einer in spitzen Klammern eingeschlossenen Parameterliste, vorangestellt. Die Parameterliste enthält eine (nicht leere) Liste von Typ- oder Klassenparametern, die mit dem Schlüsselwort `class` oder `typename` beginnen.

`minimum` ist ein Funktions-Template, das das Minimum aus einem Vektor von beliebigen Werten suchen soll.

```
template<class ElemType>
ElemType minimum(ElemType elemField[], int fieldSize)
```

Anmerkung: `class` oder `typename` sind gleichwertig.

Die Implementierung des oben angeführten Funktions-Templates erfolgt analog der Implementierung „normaler`` Funktionen. Die Tatsache, daß der Typ `ElemType` noch nicht bekannt ist, ist für den Übersetzungsvorgang nicht entscheidend:

```
template <class ElemType>
ElemType minimum(ElemType elemField[], int fieldSize)
{
  int         min;

  min = 0;
  for (int i=1;i<fieldSize; ++i) {
    if (elemField[i]<elemField[min]) {
      min = i;
    }
  }
```

```
  return elemField[min];
}
```

Funktions-Templates stellen eine Art „Schablonen`` dar, die noch nicht ausführbar sind. Der Compiler erzeugt also bei der Übersetzung von Templates keinen ausführbaren Code. Der Grund dafür liegt darin, daß zu diesem Zeitpunkt der konkrete Template-Typ (hier: `ElemType`) noch nicht bekannt ist. Das oben angeführte Funktions-Template kann wie eine „normale`` Funktion verwendet werden:

```
#include <iostream>

int    iF[] = {1, 2, 3, 4, 5, 1, 0, 1, 10};
double dF[] = {1.0, 0.4, 0.3, 10.0};

main()
{
   int size1 = sizeof(iF)/sizeof(int);
   int i = minimum(iF, size1);
   int size2 = sizeof(dF)/sizeof(double);
   double d = minimum(dF, size2);

   cout << i << "\n" << d << "\n";
}
```

Funktions-Templates können beliebig viele Template-Parameter haben. Die einzelnen Parameter werden durch einen Beistrich voneinander getrennt. Funktions-Templates können auch als `inline`, `extern` oder `static` deklariert werden. Wichtig ist, daß in diesem Fall die entsprechenden Schlüsselwörter erst nach dem Schlüsselwort `template` und seiner formalen Parameterliste angeführt werden dürfen:

```
  // Prototyp eines inline-Funktions-Templates
template <class ElemType>
inline ElemType swap(ElemType& a, ElemType& b);
```

Vorige Seite: 12.1 Motivation **Eine Ebene höher:** 12 Templates **Nächste Seite:** 12.2.1 Ausprägung von Funktions-Templates

Vorige Seite: 12.2 Funktions-Templates **Eine Ebene höher:** 12.2 Funktions-Templates **Nächste Seite:** 12.2.2 Überladen von Funktions-Templates

12.2.1 Ausprägung von Funktions-Templates

Der Compiler erkennt automatisch, daß der Aufruf `minimum(iF, size1)` sich auf ein Funktions-Template bezieht und prägt dieses aus. Die Ausprägung eines Templates bewirkt, daß der Compiler die (bereits übersetzte) generische Schablone an konkrete Template-Parameter bindet und so eine ausführbare Einheit erzeugt.

Bei der Ausprägung ist aber zu beachten, daß für die Auflösung ausschließlich die Funktionssignatur (siehe Abschnitt 7.2) entscheidend ist. Der Rückgabewert einer Funktion spielt keine Rolle.

- Die Auflösung der Ausprägung geschieht nur durch Betrachtung der Funktionsparameter, der Rückgabewert wird nicht ausgewertet.

Erst durch das Ausprägen eines Funktions-Templates, also der erzeugten Schablone, wird vom Compiler eine ausführbare Funktion erstellt. Zu beachten ist in diesem Zusammenhang der Unterschied zwischen den Begriffen Template-Funktion und Funktions-Template:

- Funktions-Template
  Ein Funktions-Template ist eine generische Funktionsschablone, die mit verschiedenen Klassen-/Typ-Argumenten parametrisiert ist. Ein Funktions-Template ist keine ausführbare Einheit und dient nur als Deklaration beziehungsweise Definition.
- Template-Funktion
  Eine Template-Funktion ist die Ausprägung eines Funktions-Templates. Durch die Ausprägung wird eine Schablone an konkrete Datentypen/Klassen gebunden und damit zu einer ausführbaren Einheit.

Funktions-Templates können *explizit* (= ausdrücklich als Funktions-Template) qualifiziert werden:

```
int i = minimum<int>(iF, size1);
```

Mit expliziter Qualifikation ist es auch möglich, Template-Funktionen zu vereinbaren, deren Parameter nicht über die Parameterliste bestimmbar sind:

```
template<typename T, typename X>
T anyFoo(const X& x)
```

```
{ ... }

int  help;
char ch;
help = anyFoo(ch);  // Fehler! Template-Parameter T
                    // nicht aufloesbar, da Rueckgabe-
                    // wert nicht beachtet wird
help = anyFoo<int>(ch); // OK, explizite Qualifikation
```

Zudem gibt es die Möglichkeit, den Compiler zu einer expliziten Ausprägung einzelner ,,Instanzen`` von Funktions-Templates zu zwingen. Folgende Anweisungen zwingen den Compiler, zwei spezielle Instanzen der Funktions-Templates `minimum` und `swap` zu erzeugen:

```
template int minimum<int>(int);
template char swap<char>(char&, char&);
```

Explizite Ausprägungen wie die oben angeführten scheinen auf den ersten Blick unnötig. Wozu Funktions-Templates explizit ausprägen, wenn dies der Compiler ohnehin bei der ersten Verwendung übernimmt?

Die Gründe dafür hängen mit dem Zeitpunkt der möglichen automatischen Ausprägung zusammen: Eine sichere automatische Ausprägung kann (ohne aufwendige Prüfungen) erst stattfinden, wenn alle Programmdateien übersetzt sind. Das bedingt, daß Templates (im einfachsten Fall) beim Binden erzeugt werden. Dazu sind eine aufwendige Compiler-Implementierung beziehungsweise ein erneuter Compiler-Lauf nötig, oder aber die Ausprägung wird dem Programmierer überlassen. In C++ können daher Templates explizit (das heißt vom Programmierer gesteuert) ausgeprägt werden.

Ferner können mit Hilfe von expliziter Ausprägung vom Programmierer bestimmte Ausprägungen ,,angefordert`` werden.

Zu beachten ist aber der Zeitpunkt *aller* Ausprägungen:

- Die Ausprägung von Funktions-Templates findet zur Übersetzungszeit statt, nicht zur Laufzeit.

Vorige Seite: 12.2 Funktions-Templates **Eine Ebene höher:** 12.2 Funktions-Templates **Nächste Seite:** 12.2.2 Überladen von Funktions-Templates

Vorige Seite: 12.2.1 Ausprägung von Funktions-Templates **Eine Ebene höher:** 12.2 Funktions-Templates

12.2.2 Überladen von Funktions-Templates

Funktions-Templates können sowohl mit anderen Funktions-Templates als auch mit ,,normalen`` Funktionen überladen werden. Die Namensauflösung erfolgt nach folgendem Schema:

- Der Compiler geht die Liste der möglicherweise passenden Funktions-Templates durch und erzeugt die entsprechenden Template-Funktionen.
- Das Ergebnis ist eine Reihe von (möglicherweise) passenden Template-Funktionen ergänzt durch die vorhandenen ,,normalen`` Funktionen. Diese Liste wird wie im Fall von ,,normalen`` überladenen Funktionen nach der am besten passenden durchsucht, die dann ausgewählt wird.

Vorige Seite: 12.2.2 Überladen von Funktions-Templates **Eine Ebene höher:** 12 Templates **Nächste Seite:** 12.3.1 Definition von Klassen-Templates

12.3 Klassen-Templates

Klassen-Templates sind mit Typen oder Konstanten parametrisierte Klassen. Sie sind eine Art Schablonen, die vor ihrer Verwendung mit konkreten Parametern ausgeprägt werden. Die Schablone gibt dem Compiler an, wie die Template-Klassen aufgebaut sind, welche Struktur und Daten sie haben und welche Methoden zur Verfügung gestellt werden.

Bei der Ausprägung mit konkreten Werten und Typen bindet der Compiler die Schablone mit den aktuellen Parametern zu einer ausführbaren Einheit. Ab diesem Zeitpunkt kann die Klasse verwendet werden.

Vorige Seite: 12.3 Klassen-Templates **Eine Ebene höher:** 12.3 Klassen-Templates **Nächste Seite:** 12.3.2 Ausprägung von Klassen-Templates

12.3.1 Definition von Klassen-Templates

Die Definition von Klassen-Templates erfolgt fast analog der Syntax von Funktions-Templates. Dazu wird vor der eigentlichen Klassendefinition das Schlüsselwort `template`, gefolgt von der Template-Parameterliste, angeführt.

Die Template-Parameterliste enthält - eingeschlossen in die eckigen Klammern `< >` - die Parameter, von denen die Klasse abhängig ist. Dabei kann ein Klassen-Template sowohl von Typen oder Klassen als auch von Ausdrücken abhängen. Alle Ausdrücke in Template-Parameterlisten müssen aber zur Übersetzungszeit aufgelöst werden können.

Wie bei Funktions-Templates muß auch hier zwischen Klassen-Templates und Template-Klassen unterschieden werden:

- Klassen-Template
  Ein Klassen-Template ist eine allgemeine Schablone einer Klasse, die mit verschiedenen Typ- oder Ausdrucksargumenten parametrisiert ist.
- Template-Klasse
  Eine Template-Klasse ist die Ausprägung eines Klassen-Templates. Durch die Ausprägung wird eine Schablone an konkrete Datentypen/Klassen gebunden und damit zu einer ausführbaren Einheit.

Im folgenden Programm wird ein `Stack`-Template vereinbart, das `Size` Elemente in einem `Stack` beliebigen Typs verwaltet:

Programm 12.1: Klassen-Template `Stack` (Schnittstelle)

```
#ifndef ARRTMPL__H
#define ARRTMPL__H

template <typename ElemType, int Size=100>
class Stack
{
public:
  Stack();
  ~Stack();
  void push(const ElemType& anElem);
  void pop(ElemType& anElem);
  bool wasError() const;
  bool isEmpty() const;
private:
  ElemType    elems[Size];
  int         top;
  bool        errorOccd;
};
#endif
```

Das Klassen-Template ist von zwei Template-Parametern abhängig: `ElemType` und `Size`. `ElemType` ist ein Typ oder eine Klasse (`typename`), `Size` ist ein „normaler`` Ausdrucksparameter, der mit dem Standardwert 100 vorbelegt ist.

Die Implementierung der Klasse `Stack` kann im wesentlichen wie eine ,,normale`` Klassenimplementierung erfolgen. Dabei muß aber eine etwas umständliche Notation befolgt werden:

- Ist ein Klassen-Template definiert, so kann sein Name benutzt werden wie der einer ,,normalen`` Klasse. Die Einschränkung dabei ist, daß hinter dem Namen die Template-Parameter in spitzen Klammern angeführt werden müssen.
- Innerhalb des eigenen Klassen-Templates kann der Klassenname ohne Parameterliste verwendet werden.
- Vor jeder Element-Funktion eines Klassen-Templates muß spezifiziert werden, daß es sich um eine Element-Funktion eines Templates handelt. Dazu wird vor der Methode das Schlüsselwort `template` zusammen mit den in `< >` eingeschlossenen Template-Parametern angegeben:

```
template <class ElemType, int size>
```

Die Implementierung des Klassen-Templates `Stack` sieht so aus:

Programm 12.2: Klassen-Template `Stack` (Implementierung)

```
#include "arrtmpl.h"

template <class ElemType, int Size>
Stack<ElemType, Size>::Stack()
 : top(0), errorOccd(false)
{}

template <class ElemType, int Size>
Stack<ElemType, Size>::~Stack()
{}

template <class ElemType, int Size>
void Stack<ElemType, Size>::push(const ElemType& anElem)
{
   errorOccd = (top == Size);
   if (!errorOccd) {
      elems[top++] = anElem;
   }
}

template <class ElemType, int Size>
void Stack<ElemType, Size>::pop(ElemType& anElem)
{
   errorOccd = (top == 0);
   if (!errorOccd) {
      anElem = elems[--top];
   }
}

template <class ElemType, int Size>
bool Stack<ElemType, Size>::wasError() const
{
   return errorOccd;
}

template <class ElemType, int Size>
bool Stack<ElemType, Size>::isEmpty() const
{
   return (top==0);
}
```

Anmerkung: Bei der Implementierung des Konstruktors und des Destruktors wird die Template-Parameterliste nur am Zeilenanfang angeführt. Nach dem *Scope*-Operator muß die Parameterliste nicht verwendet werden, da sich in diesem Fall der Name `Stack` auf die Element-Funktion (Konstruktor) und nicht auf den Klassennamen bezieht.

Vorige Seite: 12.3 Klassen-Templates **Eine Ebene höher:** 12.3 Klassen-Templates **Nächste Seite:** 12.3.2 Ausprägung von Klassen-Templates

Vorige Seite: 12.3.1 Definition von Klassen-Templates **Eine Ebene höher:** 12.3 Klassen-Templates **Nächste Seite:** 12.3.3 Geschachtelte Template-Klassen und `friend`

12.3.2 Ausprägung von Klassen-Templates

Das oben angeführte Klassen-Template kann nun verwendet werden, indem es übersetzt und dann mit konkreten Werten für die Template-Parameter ausgeprägt wird. Die Ausprägung erfolgt durch Angabe des Template-Namens gefolgt von den jeweiligen aktuellen Parametern in spitzen Klammern. Die Anweisung

```
Stack<int, 10>    s;
```

prägt das `Stack`-Template mit den Parametern `int` und `10` aus und erstellt damit einen konkreten `int`-Stack, der 10 Elemente aufnehmen kann. `s` ist ein Objekt dieser Klasse.

Da `Size` vorbelegt ist, reicht es aus, einen Typ bei der Ausprägung anzugeben:

```
Stack<int>    s;
```

In Programm 12.3 ist ein sehr einfaches Testprogramm für die Klasse `Stack` angeführt.

Programm 12.3: Testprogramm für Klassen-Template `Stack`

```
#include "arrtmpl.h"

void main()
{
   Stack<int, 10>    s;
   char              ch;
   int               i;

   cout << "\n\nOperation (Quit, pUsh, pOp, isEmpty) ";
   cin >> ch;

   while (ch != 'Q') {
      switch (ch) {
       case 'U':
         cout << "\n Element to push? ";
         cin >> i;
         s.push(i);
         if (s.wasError()) {
            cout << "Error!";
         } else {
            cout << "\n pushed element " << i;
         }
         break;
      case 'O':
         s.pop(i);
         if (s.wasError()) {
            cout << "Error!";
         } else {
            cout << "\n popped element " << i;
```

```
         }
         break;
      case 'E':
         if (s.isEmpty()) {
            cout << "\n Stack is empty.";
         } else {
            cout << "\n Stack contains elements.";
         }
         break;
      default:
         cout << "\n Wrong operation.";
      }
      cout << "\n\nOperation (Quit, pUsh, pOp, isEmpty) ";
      cin >> ch;
   }
}
```

Vorige Seite: 12.3.2 Ausprägung von Klassen-Templates **Eine Ebene höher:** 12.3 Klassen-Templates
Nächste Seite: 12.3.4 Explizite Ausprägung und

12.3.3 Geschachtelte Template-Klassen und `friend`

Klassen-Templates können nicht nur einfache Klassen wie die oben angeführte Klasse `Stack` sein, sie können andere ,,normale`` Klassen oder andere Templates enthalten, sowie beliebige Klassen als `friend` aufweisen.

Enthält ein Klassen-Template eine zweite Klasse, so gibt es zwei Möglichkeiten:

- Die zweite,

  innere Klasse kann eine ,,normale`` Klasse sein. Damit ist die zweite Klasse ebenfalls abhängig von den Template-Parametern, da sie ja intern zu dieser definiert wurde. Sie wird bei der Ausprägung der äußeren Klasse automatisch ausgeprägt (= gebundenes Template).
- Die innere

  Klasse ist wieder ein Klassen-Template. Damit enthält die äußere Klasse ein Template, das sowohl von der äußeren Klasse als auch von seinen eigenen Template-Parametern abhängig ist. Die innere Klasse kann erst verwendet werden, wenn sowohl die äußere als auch die innere Klasse ausgeprägt wurde (= ungebundenes Template).

Ein Beispiel für die erste Möglichkeit ist bei der Listenverarbeitung gegeben. Ein Klassen-Template `Tree` ist mit dem zu verwaltenden Datentyp parametrisiert und enthält eine Klasse `Node`. Diese stellt einen Knoten der Liste dar und enthält Datenteil und Verweisteil:

```
template <class ElemType>
class Tree
{
  ...
public:
  class Node
  {               // Umgebendes Template ist friend:
    friend Tree<ElemType>;
    ...
  }
};
```

In diesem Beispiel ist die innere Klasse `Node` von `Tree` abhängig und damit ebenfalls parametrisiert.

Die innere Klasse wird automatisch mit der äußeren ausgeprägt. Die äußere Klasse ist als ein `friend` von `Node` vereinbart, wobei die Parameterliste in spitzen Klammern angegeben werden muß. Damit ist für eine Ausprägung von `Tree` mit `int` genau die `int`-Ausprägung `friend` der inneren Klasse `Node` und keine andere.

Bei der zweiten Möglichkeit (innere Klasse = Klassen-Template) ist zwischen drei verschiedenen Arten von `friend`s zu unterscheiden:

- Ein Template hat einen speziellen `friend`. Dieses Beispiel wurde bereits angeführt.
- Ein Template hat *eine Ausprägung* eines anderen Templates als `friend`:

```
template <class T>
class X
{
  ...
  friend class AnotherTemplate<T>;
};
```

- Ein Template hat *alle Ausprägungen* eines anderen Templates als `friend`:

```
template <class T>
class X
{
  ...
  template<class TT>
  friend class AnotherTemplate;
};
```

Vorige Seite: 12.3.2 Ausprägung von Klassen-Templates **Eine Ebene höher:** 12.3 Klassen-Templates
Nächste Seite: 12.3.4 Explizite Ausprägung und

Vorige Seite: 12.3.3 Geschachtelte Template-Klassen und `friend` **Eine Ebene höher:** 12.3 Klassen-Templates **Nächste Seite:** 12.3.5 Template-Typen

12.3.4 Explizite Ausprägung und Spezialisierung

Ebenso wie Funktions-Templates können auch Klassen-Templates explizit ausgeprägt werden. Der Compiler erzeugt dann Code für genau diese Instanz und behandelt sie so, als ob eine „normale`` Klasse vorliegt.

```
template <typename KeyType, typename ElemType>
class Map {
...
};
...
class Map<int, TString>;
```

Eine Spezialisierung eines Templates ist die Deklaration eines alternativen Namens mit einer eingeschränkten Parameterliste:

```
template <typename KeyType, typename ElemType>
class Map {
...
};
...
template<typename T>  // Spezielle Variante v. Map
class Map<T, T>;      // mit zwei gleichen Template-
                      // Parametern

template<typename T>  // Spezielle Variante v. Map mit
class Map<T*, T>;     // einem Zeiger und einem Typ

template<> class Map<int, TString>;
```

Die oben angegebenen teilweisen Spezialisierungen (`Map<T,T>` beziehungsweise `Map<T*, T>`) stellen neue, speziellere Varianten desselben Templates zur Verfügung.

Alle Spezialisierungen müssen mit `template<>` beginnen. Daher beginnt auch die Spezialisierung von `Map` mit `int` und `TString` so.

Vorige Seite: 12.3.4 Explizite Ausprägung und **Eine Ebene höher:** 12.3 Klassen-Templates

12.3.5 Template-Typen

Werden Typen verwendet, die im Sichtbarkeitsbereich von Template-Parametern vereinbart sind, so müssen diese explizit als solche gekennzeichnet werden, indem das Schlüsselwort `typename` vorangestellt wird:

```
template <typename T>
class X {
  ...
  typename T::X o;  // T::X ist Typ!
  ...
};
...
class Tree
{
  ...
  class X { ... };
};

X<Tree>  x;
```

Wird das Schlüsselwort `typename` bei der Deklaration von `o` nicht verwendet, so geht der Compiler davon aus, daß `T::X` ein konkretes Element (Variable, Konstante) ist und liefert einen Fehler!

Vorige Seite: 12.3.5 Template-Typen **Eine Ebene höher:** 12 Templates **Nächste Seite:** 12.5 Beispiele und Übungen

12.4 Element-Templates

Auch Element-Funktionen können als Templates vereinbart werden. Damit ist es zum Beispiel möglich, eine allgemeine Klasse `Builder` zu erstellen, deren Aufgabe das Erzeugen von Objekten ist. Neben anderen, speziellen Aufgaben stellt die Klasse eine Element-Funktion zum Allokieren von beliebigen Speicherbereichen zur Verfügung.

Diese Element-Funktion wird als Template-Element-Funktion (kurz: Element-Template) implementiert und kann damit für beliebige Typen verwendet werden:

```
class Builder
{
  ...
  template<class T> static T* allocateMem();
};
```

Die Ausprägung der Element-Funktion erfordert die Angabe des Schlüsselworts `template`:

```
test = Builder::template allocateMem<int>();
```

Zu beachten ist allerdings eine Einschränkung, die die in Kapitel 14 besprochenen `virtual`-Methoden betrifft:

Der Template-Mechanismus nimmt Ausprägungen ausschließlich zur Übersetzungszeit vor. Template-Element-Funktionen können daher nicht als `virtual` vereinbart werden.

Vorige Seite: 12.4 Element-Templates **Eine Ebene höher:** 12 Templates **Nächste Seite:** 12.5.1 Funktions-Template `binSearch`

12.5 Beispiele und Übungen

Die folgenden Übungsbeispiele umfassen ein (einfaches) Funktions-Template sowie zwei Klassen-Templates:

- `binSearch` ist ein Funktions-Template, das in einem Vektor binär nach einem bestimmten Element sucht.
- `List` ist ein Klassen-Template, das die Verwaltung von beliebigen Elementen in einer Liste erlaubt. Themenbereiche sind Klassen-Templates, dynamische Datenstrukturen, geschachtelte Klassen sowie Iteratoren.
- `Tree` ist ein ähnliches (wenn auch algorithmisch schwierigeres) Beispiel. Zu erstellen ist ein Klassen-Template `Tree`, das Elemente in einem binären Baum verwaltet, sowie ein passender Iterator.

- Hintergrundwissen

Vorige Seite: 12.5 Beispiele und Übungen **Eine Ebene höher:** 12.5 Beispiele und Übungen **Nächste Seite:** 12.5.2 `List`-Template

---

Teilabschnitte

12.5.1 Funktions-Template `binSearch`

Themenbereich

Funktions-Templates

Komplexität

Einfach

Aufgabenstellung

Im folgenden ist der Prototyp einer universellen generischen Funktion `binSearch` angegeben. Sie sucht in (fast) beliebigen eindimensionalen, aufsteigend sortierten Vektoren binär nach `theElem`. Als Resultat wird der Index des ersten Vektor-Elements, das größer oder gleich dem zu suchenden Element (`theElem`) ist, zurückgegeben. Der Parameter `found` gibt an, ob das Element gefunden wurde.

```
template <typename T>
void binSearch(T itemField[],
               const T theElem,
               int size,
               int& index,
               bool& found);
```

Implementieren Sie die Funktion und testen Sie sie mit zumindest zwei Vektoren verschiedenen Typs. Beantworten Sie folgende Fragen:

- Kann die Funktion jeweils mit den Typen `char*`, `int`, `char` ausgeprägt werden?

- Arbeitet die Funktion in jedem dieser Fälle korrekt?

Hintergundwissen

Eine binäre Suche kann nur in sortierten Bereichen stattfinden und läuft (bei aufsteigender Sortierung) nach folgendem Schema ab:

- Die Suche nach dem Element beginnt in der Mitte des Bereichs.
- Das aktuelle Element wird mit dem zu suchenden verglichen:
  - Ist das aktuelle Element gleich dem zu suchenden, so wird die Suche abgebrochen.
  - Ist das aktuelle Element kleiner als das zu suchende, so wird die Suche in der linken Hälfte des Vektors nach demselben Schema fortgesetzt.
  - Ist das aktuelle Element größer als das zu suchende, so wird die Suche in der rechten Hälfte des Vektors nach demselben Schema fortgesetzt.

Vorige Seite: 12.5 Beispiele und Übungen **Eine Ebene höher:** 12.5 Beispiele und Übungen **Nächste Seite:** 12.5.2 `List`-Template

Vorige Seite: 12.5.1 Funktions-Template `binSearch` **Eine Ebene höher:** 12.5 Beispiele und Übungen
Nächste Seite: 12.5.3 `Tree`-Template

Teilabschnitte

12.5.2 `List`-Template

Themenbereich

Klassen-Template, dynamische Datenstrukturen, Iteratoren

Komplexität

Hoch

Aufgabenstellung

Implementiert werden soll ein Klassen-Template `List`, das beliebige Daten in einer verketteten Liste verwaltet. Die Klasse stellt die ,,üblichen`` Zugriffsfunktionen zur Verfügung:

- `add` fügt ein Element am Listenende ein,
- `remove` entfernt ein Element vom Listenanfang,
- `wasError` gibt an, ob letzte Operation fehlerhaft war, und
- `isEmpty` gibt an, ob die Liste leer ist.

Die einzelnen Elemente werden durch eine interne Klasse (`Node`) repräsentiert. `Node` enthält sowohl die eigentlichen Daten als auch einen Verweis auf das nächste Element.

Zur einfacheren Verwaltung von Listen wird ferner ein *Iterator* zur Verfügung gestellt, der über eine interne Klasse `Iterator` realisiert ist. `Iterator` stellt folgende Methoden zur Verfügung:

- `reset` setzt den Iterator an den Listenanfang (er verweist dann auf das erste Listenelement),

- `++` rückt den Iterator in der Liste um eine Position vorwärts (er verweist dann auf das *nächste* Element),
- `--` rückt den Iterator um eine Position rückwärts (er verweist dann auf das *vorige* Element),
- `element` liefert eine Referenz auf das Listenelement, auf das der Iterator zeigt, und
- `isValid` prüft, ob der Iterator gültig ist (ein Iterator ist dann gültig, wenn er auf ein Listenelement verweist).

Spezifizieren Sie eine entsprechende Schnittstelle und implementieren Sie die Klasse `List`.

Hintergrundwissen

Ein Iterator ist eine Datenstruktur, die den einfachen Zugriff auf die gespeicherten Elemente in einer definierten Reihenfolge erlaubt. Man unterscheidet grundsätzlich zwischen aktiven und passiven Iteratoren:

- Passive Iteratoren ,,besuchen`` alle Elemente einer Datenstruktur in einer definierten Reihenfolge und führen mit jedem Element eine Operation durch.
- Aktive Iteratoren erlauben mehr Kontrolle und stellen es dem Benutzer frei, in welcher Reihenfolge die einzelnen Elemente der Datenstruktur verarbeitet werden. Üblicherweise können aktive Iteratoren innerhalb der assoziierten Datenstruktur vorwärts oder rückwärts bewegt und absolut positioniert werden. Außerdem kann (über eine spezielle Operation) auf das Element, auf das er verweist, zugegriffen werden.

Im Fall des angegebenen Beispiels enthält der Iterator zwei Zeiger. Beim Initialisieren werden die Zeiger so gesetzt, daß sie auf den Listenanfang und das aktuelle Element verweisen. Das aktuelle Element ist das Element, das über die Operation `element()` zurückgegeben wird und dessen Position durch die Operatoren `++` und `--` verändert wird.

Mit einem aktiven Iterator sind zum Beispiel Konstrukte der folgenden Art möglich:

```
Iterator  it(aList);      // Iterator initialisieren
while (it.isValid()) {    // Verweist Iterator auf ein
                          // gueltiges Element?
  it.element()->print();  // Zugriff auf Element
  ...
  ++it;  // Eine Position nach vorne verschieben
}
```

Ein Problem im Umfeld von Iteratoren ist ihre ,,Robustheit``. Ein Beispiel soll die Problematik aufzeigen:

1. Eine Liste mit den Elementen `1`, `10` und `45` wird angelegt.
2. Ein Iterator wird mit der Liste assoziiert. Er zeigt damit auf das erste Listenelement (`1`).
3. Die Liste wird verändert: Die Elemente `1` und `10` werden gelöscht, neue Elemente werden

eingefügt.

4.

Nun wird mit dem Iterator auf das aktuelle Listenelement (Operation `element()` zugegriffen.

Der Iterator verweist auf ein mittlerweile nicht mehr vorhandenes Element, ein Fehler ist die Folge.

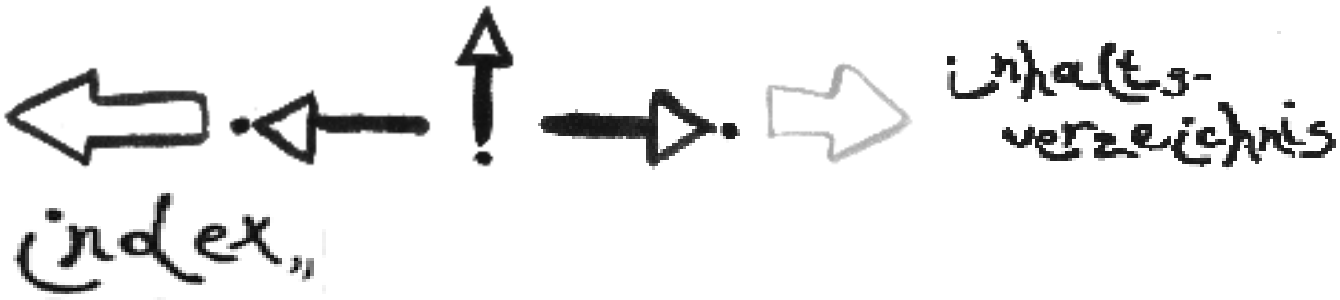
Iteratoren, die es erlauben, daß assoziierte Listen verändert werden, werden als ,,robust`` bezeichnet.

Will man einen nicht-robusten Iterator implementieren, so empfiehlt es sich, zumindest Fehler der oben angeführten Art zu vermeiden. Dazu wird in der Liste ein ,,Schlüssel`` mitgeführt, der bei jeder Schreiboperation (`add`, `remove`, `removeAll`) inkrementiert wird. Der Iterator enthält ebenfalls einen Schlüssel, der bei der Assoziierung mit einer Liste mit deren Schlüssel initialisiert wird.

In der Folge kann die ,,Gültigkeit`` eines Iterators überprüft werden, indem der Schlüssel des Iterators mit dem der assoziierten Liste verglichen wird. Nur wenn beide Schlüssel übereinstimmen, wurde die Liste seit der Initialisierung des Iterators nicht mehr verändert.

Iteratoren sind ein wesentliches Konzept der *Standard Template Library* (STL), die in der Standard C++-Bibliothek enthalten ist. Für eine Diskussion der STL siehe zum Beispiel [MS96]. Für nähere Betrachtungen der allgemeinen Thematik der Iteratoren sei auf [Boo87, Mur93, GHJV95] verwiesen.

Vorige Seite: 12.5.1 Funktions-Template `binSearch` **Eine Ebene höher:** 12.5 Beispiele und Übungen
Nächste Seite: 12.5.3 `Tree`-Template



Vorige Seite: 12.5.2 `List`-Template **Eine Ebene höher:** 12.5 Beispiele und Übungen

---

Teilabschnitte

12.5.3 `Tree`-Template

Themenbereich

Klassen-Templates, Iteratoren

Komplexität

Hoch

Aufgabenstellung

Zu entwerfen und zu implementieren ist ein Klassen-Template `Tree`. Die Klasse kann für die Verwaltung beliebiger Daten in einem binären Suchbaum ausgeprägt werden und stellt zumindest folgende Operationen zur Verfügung:

- Einfügen von neuen Elementen
- Suchen von Elementen im Binärbaum
- Löschen von Elementen

Entwerfen und implementieren Sie zudem einen Iterator für die Klasse `Tree`. Der Iterator ist innerhalb der Klasse `Tree` zu vereinbaren. Stellen Sie zumindest folgende Operationen zur Verfügung:

- Initialisieren und De-Initialisieren
- Zugriff auf das aktuelle Element
- Relative Bewegung um eine Position nach vorne beziehungsweise zurück (der Iterator verweist

dann auf das logisch nächste/vorherige Element)

- Absolute Positionierung (logisch erstes/letztes Element im Baum)
- Gültigkeitstest

Hintergrundwissen

Ein binärer Suchbaum ist ein Sonderfall einer verketteten Liste mit folgenden Eigenschaften:

- Jedes Element hat 0, 1 oder 2 Nachfolger, einen linken und einen rechten, im folgenden als *Söhne* bezeichnet.
- Jeder der Nachfolger kann ebenfalls wieder 0, 1 oder 2 Nachfolger haben.
- Für alle linken Söhne eines Knotens gilt, daß ihre Werte kleiner als der Wert des aktuellen Knotens sind. Alle rechten Söhne weisen größere Werte auf.

Ein Beispiel für einen binären Suchbaum ist in Abbildung 12.1 angeführt. Der abgebildete Baum enthält das ,,Wurzel-Element`` `10`, links davon sind die Elemente `5` und `7` angeordnet. `17` ist größer als `10` und daher im rechten Teilbaum abgelegt.

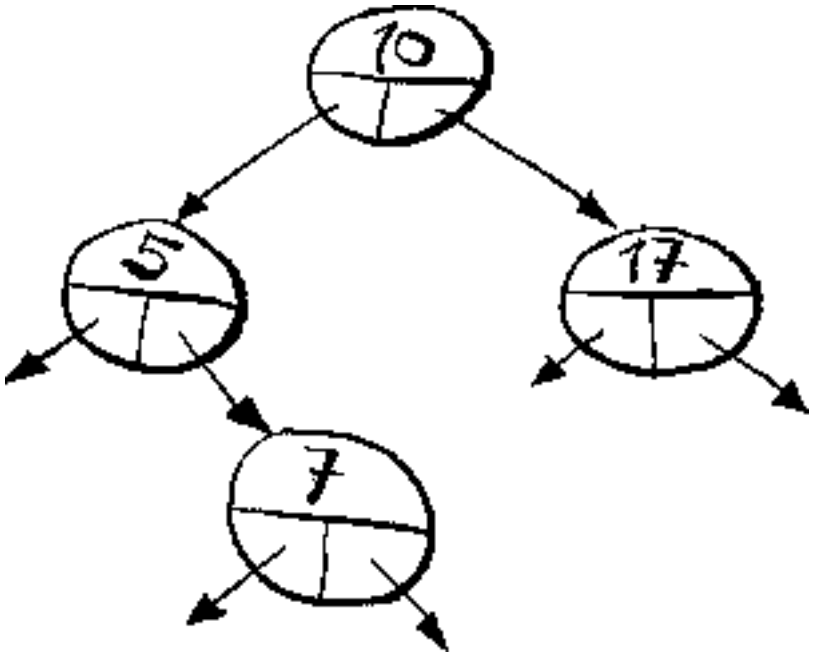


Abbildung 12.1: Beispiel für einen binären Suchbaum

Mit dieser Struktur kann relativ schnell nach gespeicherten Elementen gesucht werden. Die Suche nach einem Element *n* in einem Baum läuft wie folgt ab:

1. Ist der Wert *n* im aktuellen Knoten gespeichert, so ist die Suche beendet.
2. Ist der Wert *n* kleiner dem Wert des aktuellen Knotens, so wird die Suche im linken Teilbaum fortgesetzt.

3. Ist der Wert *n* größer als der Wert des aktuellen Knotens, so wird die Suche im rechten Teilbaum fortgesetzt.

4. Hat der aktuelle Knoten keine Nachfolger, so ist das Element *n* nicht im Baum enthalten.

Für eine nähere Betrachtung der Algorithmen im Umfeld von binären Suchbäumen sei auf [Sed92] verwiesen.

Vorige Seite: 12.5.2 `List`-Template **Eine Ebene höher:** 12.5 Beispiele und Übungen

Vorige Seite: Hintergrundwissen **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** 13.1 Einführung

13 Vererbung

Vererbung ist ein Konzept, das es erlaubt neue Klassen auf Basis von alten Klassen zu definieren. Dieses Kapitel behandelt die *statischen* Aspekte der Vererbung und ist wie folgt gegliedert:

- Nach einer kurzen Einführung wird besprochen, wie eine neue Klasse von einer bestehenden abgeleitet wird und was diese Ableitung bedeutet.
- Die darauffolgenden Abschnitte behandeln Gültigkeitsbereiche bei abgeleiteten Klassen, spezielle Element-Funktionen sowie das Überschreiben von ererbten Methoden.
- Den Abschluß bildet ein Abschnitt, der die verschiedenen Möglichkeiten beschreibt, wie eine Klasse von einer anderen abzuleiten ist.

- Themenbereich
- Aufgabenstellung

Vorige Seite: 13 Vererbung **Eine Ebene höher:** 13 Vererbung **Nächste Seite:** 13.1.1 Was heißt Vererbung?

13.1 Einführung

Vorige Seite: 13.1 Einführung **Eine Ebene höher:** 13.1 Einführung **Nächste Seite:** 13.1.2 Begriffe im Zusammenhang

13.1.1 Was heißt Vererbung?

Vererbung erlaubt die Vereinbarung von Klassen auf Basis von bereits bestehenden Klassen. So können Einheiten vereinbart werden, die auf bereits bestehenden aufbauen. Die neuen Einheiten besitzen, ohne Eingriffe in den Source-Code der bereits bestehenden Klassen, all deren Eigenschaften, sie ,,erben`` deren Verhalten und Daten. Zusätzlich kann

- ihr Verhalten durch das Hinzufügen von neuen Methoden erweitert werden,
- ihr Verhalten durch das Abändern (= Überschreiben) von ,,ererbten`` Methoden geändert werden, und
- sie können um neue Datenelemente erweitert werden.

Aber Vererbung ist mehr als die geschilderte, einfache Wiederverwendung von bestehenden Klassen. Vererbung definiert eine Beziehung zwischen der Basisklasse und der abgeleiteten Klasse, die sogenannte Vererbungsbeziehung. In der Regel handelt es sich dabei um die sogenannte *Is-A*-Beziehung (vergleiche Abschnitt 13.3): Die neue Klasse stellt eine ,,Spezialisierung`` der Basisklasse dar, da sie alle Eigenschaften der Basisklasse ererbt und zusätzlich neue Eigenschaften aufweisen kann.

Durch diese Beziehung werden Abstraktionen (Klassen) in Hierarchien eingeordnet. Allgemeine Klassen sind in diesen Hierarchien weiter oben angesiedelt. Von diesen allgemeinen Klassen werden speziellere abgeleitet. Die speziellen Klassen implementieren allerdings nur jenes Verhalten und jene Daten, die sie von ihren Basisklassen unterscheiden, alles andere wird ererbt.

Zwei Abbildungen zeigen Beispiele für Vererbungsbeziehungen aus dem ,,täglichen Leben``:

- Die Vererbungshierarchie *Organisation* (Abbildung 13.1) beschreibt verschiedene Organisationen. Basis ist die allgemeine *Organisation*. Von dieser sind verschiedene spezielle Organisationen abgeleitet: *Firma*, *Abteilung* und *Team*. *Personalabteilung*, *Verwaltung* und *Marketing* wiederum sind spezielle *Abteilungen*.
- Auch Lebewesen (Abbildung 13.2) können durch Abstraktionen modelliert werden, die in Hierarchien angeordnet sind: *Säugetier* und *Vogel* sind *Lebewesen*. Ein *Mensch* wiederum ist ein *Säugetier*, während *Strauß* und *Adler* spezielle *Vögel* sind.

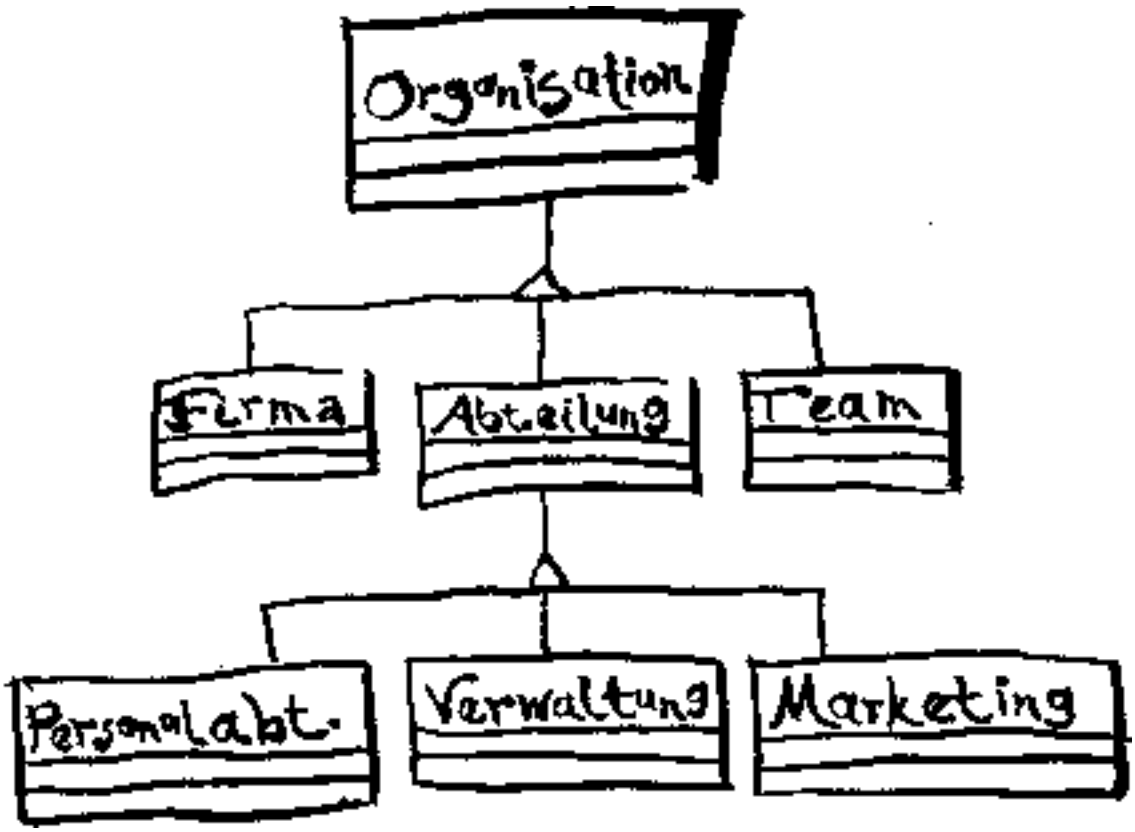


Abbildung 13.1: Vererbungshierarchie Organisation

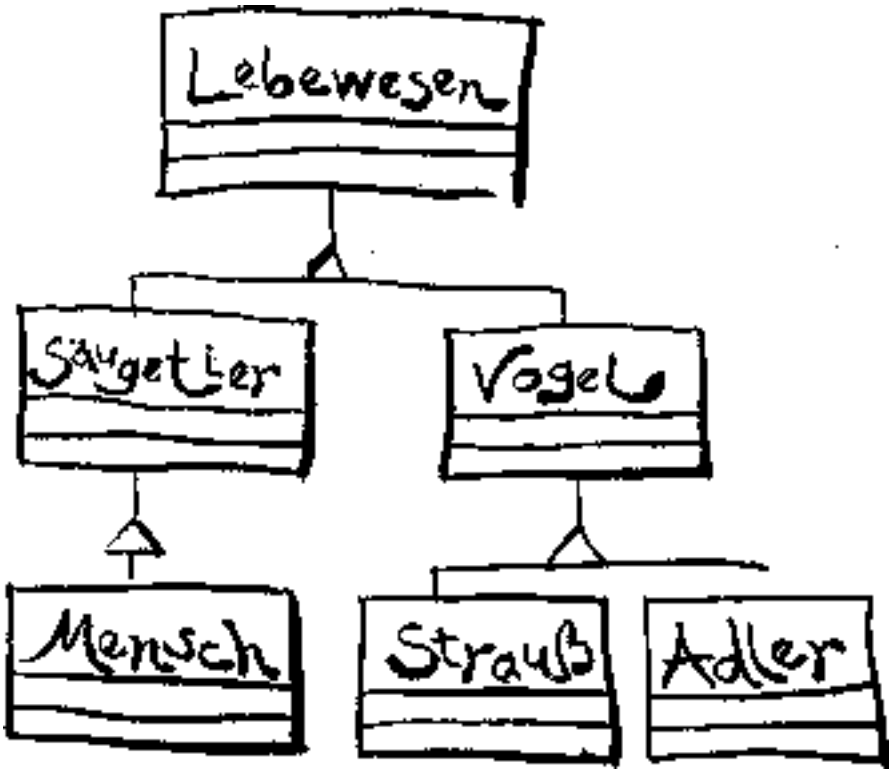


Abbildung 13.2: Vererbungshierarchie Lebewesen

Vorige Seite: 13.1 Einführung **Eine Ebene höher:** 13.1 Einführung **Nächste Seite:** 13.1.2 Begriffe im Zusammenhang

Vorige Seite: 13.1.1 Was heißt Vererbung? **Eine Ebene höher:** 13.1 Einführung

---

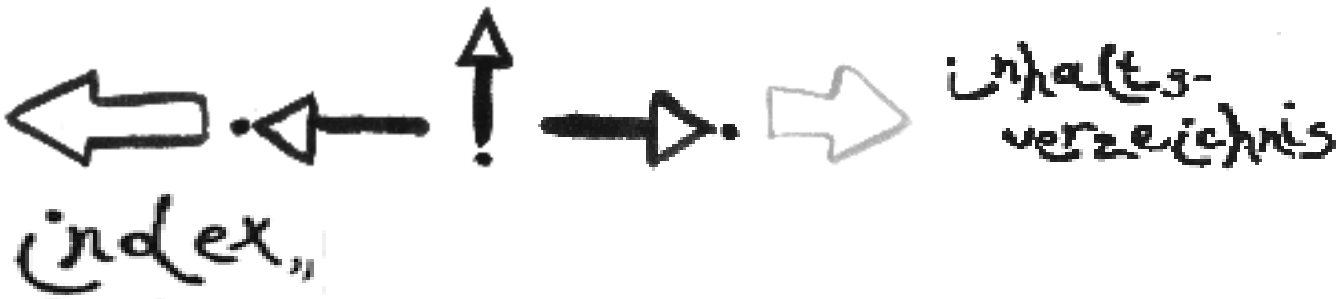
13.1.2 Begriffe im Zusammenhang mit Vererbung

Den Vorgang der Vererbung nennt man *Ableiten* . Eine Klasse von einer vorhandenen abzuleiten, heißt, sie auf der Basis dieser anderen Klasse zu definieren. Die vorhandene Klasse wird oft auch *Vaterklasse* , *Basisklasse* oder *Superklasse* genannt. Analog dazu nennt man Klassen, die von der Basisklasse abgeleitet sind, *Söhne* , *Unterklassen* oder *Subklassen* .

Die verschiedenen Klassen eines Systems stehen in Beziehung zueinander. Bisher kennen wir die sogenannte *Uses*- oder ,,Benutzt``- Beziehung und die *Has* - oder ,,Enthält`` -Beziehung: Eine Klasse benutzt eine andere, wenn sie die Klasse für Operationen oder als Parameter verwendet. Sie enthält eine andere Klasse, wenn sie Objekte vom Typ der anderen Klasse als Datenelemente direkt oder über Zeiger enthält. Durch die Vererbung wird zusätzlich die Vererbungs- oder *Is-A*-Beziehung eingeführt .

Ordnet man Klassen nach ihren Vererbungsbeziehungen, so erhält man die *Vererbungshierarchie* . Sie zeigt, welche Klasse von welcher abgeleitet ist. Die Klasse von der eine andere abgeleitet ist, wird auch als *direkte Basisklasse* bezeichnet. Klassen, die keine Basisklassen haben, nennt man auch *Wurzelklasse* oder kurz *Wurzel*.

Neben der Vererbungshierarchie ist vor allem eine gesamte *Klassentopologie* von Bedeutung. Eine Klassentopologie zeigt alle Arten von Beziehungen zwischen Klassen auf und ist damit wesentlich umfassender als eine reine Vererbungshierarchie.



Vorige Seite: 13.1.1 Was heißt Vererbung? **Eine Ebene höher:** 13.1 Einführung

Vorige Seite: 13.1.2 Begriffe im Zusammenhang **Eine Ebene höher:** 13 Vererbung **Nächste Seite:** 13.3 Die *Is-A*-Beziehung

13.2 Ableiten einer Klasse

Um eine Klasse ableiten zu können, benötigen wir zunächst eine Basisklasse. Unsere Basisklasse heißt `ComicCharacter` und ist die ,,softwaretechnische Umsetzung`` einer Comic-Figur. Von dieser Klasse ausgehend werden wir später eine Reihe von speziellen Comic-Figuren ableiten. Die Klasse `ComicCharacter` sieht wie folgt aus:

Programm 13.1: Klasse `ComicCharacter`, Basis

```
#include <iostream>
#include "tstring.h" // Klasse TString aus Kapitel 11
class ComicCharacter
{
 public:
  ComicCharacter() {};
  ComicCharacter(const TString& aStr)
   : name(aStr) {};
  virtual ~ComicCharacter() {};
  virtual void print() const { cout << "Name "
    << "of ComicChar.: " << name << "\n";
  };
  virtual void dance() const { cout << name
    << " dances... \n";
  };
  virtual void sing() const { cout << name
    << " sings... \n";
  };
  void setName(const TString& aStr) { name = aStr; };
  TString getName() const { return name; };
 private:
  TString   name;
};
```

Ein `ComicCharacter` hat einen Namen, kann tanzen und singen und sich auf dem Bildschirm ausgeben. Der Name ist vom Typ `TString` (Klasse `TString` aus Kapitel 11). Die Methoden sind nur rudimentär implementiert: `print` zeichnet keine Figur und auch `dance` und `sing` geben nur Text am Bildschirm aus. Zur Veranschaulichung der wesentlichen Konzepte genügt dies aber. Die Umsetzung der Konzepte auf ,,echte`` Probleme wird dann später anhand der Übungsbeispiele demonstriert.

An der Klasse fällt auf, daß die Methoden `print`, `dance`, `sing` und der Destruktor als `virtual` vereinbart sind. Das soll uns zunächst nicht kümmern, der Grund dafür wird später erläutert.

Ein `ComicCharacter` ist eine nicht näher bestimmte Comic-Figur, die die Eigenschaften enthält, die allen Comic-Figuren gemein sind. Ausgehend von dieser Klasse soll nun eine neue, spezielle Comic-Figur vereinbart werden, ein Superheld. Superhelden sind Comic-Figuren, die zusätzlich über Superkräfte verfügen und kämpfen können. Um die neue Klasse von `ComicCharacter` abzuleiten, muß hinter dem Klassennamen ein Doppelpunkt, gefolgt vom Schlüsselwort `public` und

dem Namen der Basisklasse, angegeben werden.

`SuperHero` sieht wie folgt aus:

Programm 13.2: Klasse `SuperHero` (1)

```
class SuperHero : public ComicCharacter
{
 public:
  SuperHero(const TString& aSuperPower = "")
   : superPower(aSuperPower) {};
  virtual ~SuperHero() {};
  virtual void fight() {
    cout << getName() << " fights ... \n";
  };
 private:
  TString superPower;
};
```

Eine abgeleitete Klasse erbt von ihrer Basisklasse *alle* Elemente, das heißt alle Methoden und Daten. Vereinfacht gesagt kann `SuperHero` alles, was auch `ComicCharacter` kann. Klassenspezifische Element-Funktionen wie Konstruktoren, Destruktoren, Zuweisungsoperatoren etc. müssen aber für jede Klasse neu vereinbart werden, da sie speziell für eine Klasse ausgelegt sind.

Allerdings können nicht alle ererbten Elemente auch tatsächlich verwendet werden. Grundsätzlich kann man zwischen drei Arten von Klienten einer Klasse `T` unterscheiden:

- `friend`s (Funktionen und Klassen) haben vollen Zugriff auf alle Klasseninternas.
- Abgeleitete Klassen haben ebenfalls gewisse ,,Sonderrechte`` und werden bevorzugt behandelt.
- Zugriffe von anderen Klienten (Funktionen, andere Klassen) werden gleich behandelt und im folgenden als ,,Zugriffe von außen`` bezeichnet.

Auf `private`-Elemente einer Klasse kann weder von außen noch von einer abgeleiteten Klasse aus zugegriffen werden. Auf `protected`-Elemente ist kein Zugriff von außerhalb möglich, sie können nur in der eigenen Klasse beziehungsweise von abgeleiteten Klassen aus verwendet werden.

Bezogen auf die `SuperHero`-Klasse können von außen also die Methoden `print`, `sing`, `dance`, `fight`, `getName` und `setName` aufgerufen werden. Die Komponente `name` ist in der Basisklasse `ComicCharacter` als `private` vereinbart und kann daher *nicht* verwendet werden. Auch innerhalb der Klasse `SuperHero` führt jede Verwendung von `name` zu einem Fehler beim Übersetzen des Programms.

```
SuperHero     sh("Speed");
SuperHero*    p = new SuperHero("Power");

sh.fight();
p->print();
p->dance();
sh.name = "X";  // Fehler! name ist private in
                // der Basisklasse
```

Wird eine Klasse `public` abgeleitet (Schlüsselwort `public` vor dem Namen der Basisklasse bei der Vereinbarung), so gelten die in Tabelle 13.1 angegebenen Zugriffsregeln. *Anmerkung*: `private` und `protected`-Ableitungen werden später erläutert.

	Sichtbar von		
Zugriffsschutz	eigener Klasse aus	abgeleiteter Klasse aus	außerhalb
public	ja	ja	ja

protected	ja	ja	nein
private	ja	nein	nein

Tabelle 13.1: Zugriffsmöglichkeiten auf Elemente von `public`-Basisklassen

Ist `name` als `protected` vereinbart, so kann darauf auch von abgeleiteten Klassen aus zugegriffen werden (vergleiche Abschnitt 11.4.1):

```
class ProtComicCharacter
{
   ...
 protected:
  TString   name;
};

class ProtSuperHero : public ProtComicCharacter
{
 public:
  void foo1() { cout << "Name: " << getName(); };
  void foo2(const ProtSuperHero& s) {
    cout << "Names: " << getName() << ", " s.getName();
  };
};
```

`foo1` und `foo2` verwenden das `protected`-Element `name` der Basisklasse. Dennoch weigert sich der Compiler, die Funktion `foo2` zu übersetzen. Der Grund liegt darin, daß ausschließlich `protected`-Elemente des *eigenen* Objekts verwendet werden dürfen. `s.name` referenziert aber ein `protected`-Element eines *anderen* Objekts (`s`) und ist damit ungültig.

`friend`-Beziehungen werden nicht vererbt. Im folgenden Beispiel wird dies deutlich:

```
class A
{
  friend B;
  ...
};
class B { ... };
class C : public B { ... };
```

`B` ist als `friend` von `A` vereinbart, kann also auf alle Komponenten von `A` zugreifen. Wird nun von `B` eine Klasse `C` abgeleitet, so gilt diese `friend`-Beziehung für die neue Klasse nicht mehr. Um auch `C` `friend`-Zugriff auf `A` zu gestatten, müßte `C` explizit als `friend` von `A` vereinbart werden.

Abschließend sollen einige ,,Richtlinien`` zur Verwendung von `public`, `protected` und `private` angegeben werden.

Daten werden bis auf wenige Ausnahmen immer in der `private`-Sektion einer Klasse vereinbart. Der Zugriff erfolgt über `public`- oder `protected`-Methoden.

Auch wenn es auf den ersten Blick ,,einfacher`` und ,,effizienter`` scheint, Daten in einer Klasse als `public` oder zumindest `protected` zu vereinbaren, sollten diese immer als `private` vereinbart werden. Trotz der Ersparnis beim Codieren (weniger *Access*-Methoden) und des vermeintlichen Performance-Gewinns (direkter Zugriff) überwiegen bei weitem die Nachteile durch die stärkere Kopplung der einzelnen Klassen, die wesentlich erschwerte Testbarkeit und andere negative Folgen des nicht eingehaltenen Geheimnisprinzips. Eine derartige Vorgangsweise kann in den meisten Fällen als kurzsichtig bezeichnet werden.

Die Erfahrung zeigt, daß `protected`-Datenelemente nur in wenigen Fällen sinnvoll sind und in den meisten Fällen zu einer unnötig hohen Bindung innerhalb der Ableitungshierarchie und zu einer Verkomplizierung der Schnittstelle führen. Längerfristig sind die Vorteile einer einfach und klar aufgebauten Schnittstelle allemal höher einzustufen als die eher kleinen Vorteile, die durch Einsatz von `protected` entstehen.

Selbst Bjarne Stroustrup erachtet die Einführung von `protected` inzwischen als einen ,,Fehler``:

> ,,In my experience, there have always been alternatives to placing significant amounts of information in a common base class for derived classes to use directly. In fact, one of my concerns about protected is exactly that it makes it too easy to use a common base the way one might sloppily have used global data
>
> In retrospect, I think that protected is a case where 'good arguments' and fashion overcame my better judgement and my rules of thumb for accepting new features.`` [Str94, S. 302].

`protected` sollte lediglich für ganz spezielle Zwecke eingesetzt werden. So ist es zum Beispiel mit `protected` möglich, daß bestimmte Konstruktoren, Zuweisungsoperatoren oder andere Element-Funktionen nur innerhalb des ,,eigenen`` Objekts verwendet werden können.

Die Verwendung von `protected`-Datenelementen sollte weitgehend vermieden werden.

Vorige Seite: 13.1.2 Begriffe im Zusammenhang **Eine Ebene höher:** 13 Vererbung **Nächste Seite:** 13.3 Die *Is-A*-Beziehung

Vorige Seite: 13.2 Ableiten einer Klasse **Eine Ebene höher:** 13 Vererbung **Nächste Seite:** 13.4 Vererbung und Gültigkeitsbereiche

13.3 Die *Is-A*-Beziehung

Ein Objekt wird vereinfacht gesagt als eine Ansammlung der Elemente im Speicher abgelegt. Wird von einer Klasse `A` eine Klasse `B` abgeleitet, so ,,erbt`` `B` alle Elemente von `A`. Im Speicher enthält ein `B`-Objekt damit alle Komponenten von `A` und eventuell zusätzliche, eigene Komponenten. Abbildung 13.3 zeigt eine vereinfachte Darstellung von einem `ComicCharacter`- und einem `SuperHero`-Objekt. Das `ComicCharacter`-Objekt enthält ein Datum (`name`). Die Klasse `SuperHero` ist von `ComicCharacter` abgeleitet und enthält damit ein *komplettes* `ComicCharacter`-Objekt *und* die eigenen Daten (`superPower`).

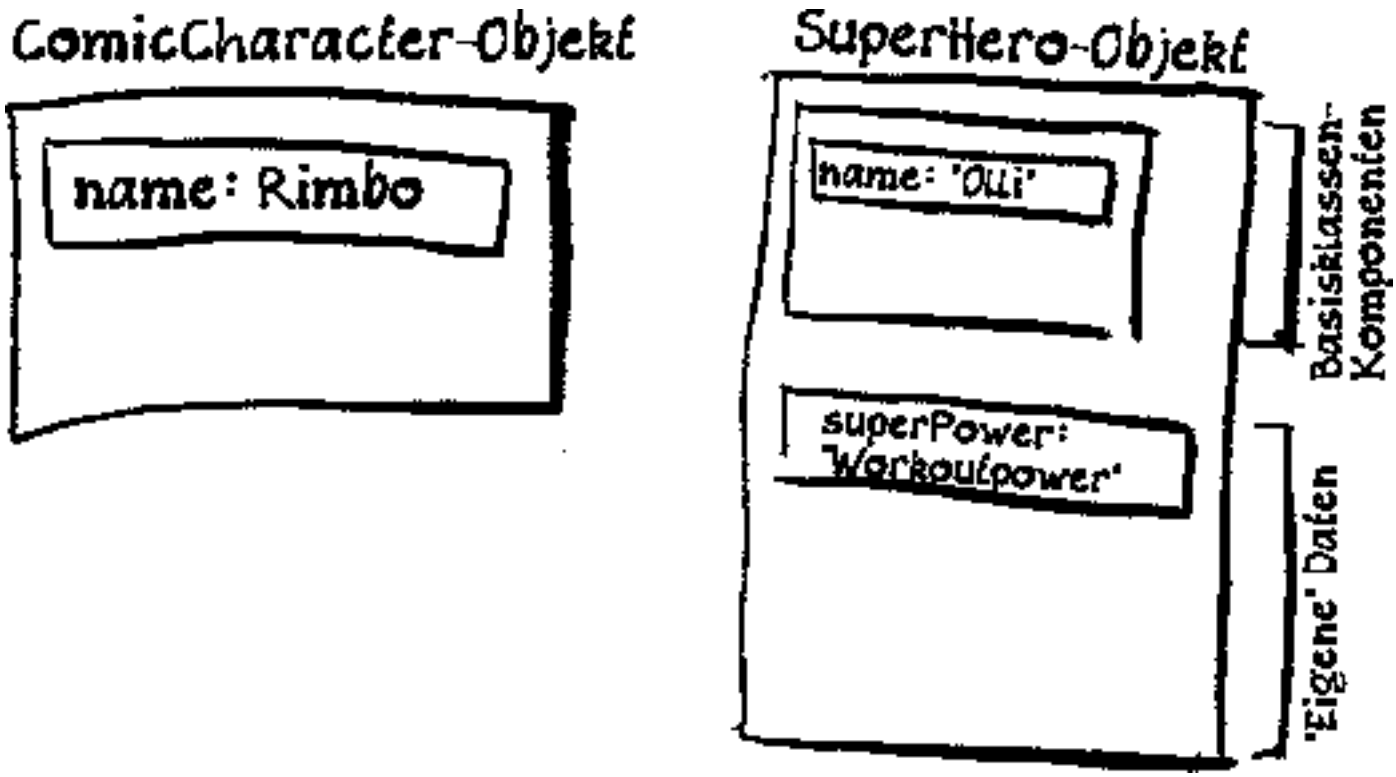


Abbildung 13.3: Repräsentation eines Objekts im Speicher

Jedes B-Objekt enthält alle Elemente von A und ist damit auch ein vollständiges A. Zudem können in B neue Datenelemente und neue Methoden vereinbart, bestehende Methoden überschrieben (= verändert) und die Sichtbarkeit ererbter Methoden auch verändert werden.

Übertragen auf unser `ComicCharacter`-Beispiel heißt dies, daß jeder `SuperHero` ein `ComicCharacter` ist. Ein `SuperHero`-Objekt weist zusätzliche Daten (`superPower`) und Methoden (`fight`) auf. Später werden verschiedene ererbte Methoden verändert werden.

Die Beziehung zwischen Basisklasse und abgeleiteter Klasse wird wie bereits oben erwähnt als *Is-A*-Beziehung bezeichnet. Sie wirkt sich nicht nur darin aus, daß `SuperHero`-Objekte singen und tanzen können. Alle `SuperHero`-Objekte können auch anstelle von `ComicCharacter`-Objekten verwendet werden. Überall dort, wo das System ein `ComicCharacter`-Objekt erwartet, kann auch ein `Superhero`-Objekt stehen:

```
void fooVal(const ComicCharacter o)
{
  o.print();
}

ComicCharacter  c("OlliBaer");
SuperHero       s("Rim van Bert");

fooVal(c);
fooVal(s);
```

Die *Is-A*-Beziehung gilt natürlich auch über mehrere Ebenen. Wird also von `SuperHero` eine Klasse `Superman` abgeleitet, so ist `Superman` sowohl ein `SuperHero` als auch ein `ComicCharacter`. Auch ein `Superman`-Objekt kann damit als Aktualparameter beim Aufruf von `fooVal` verwendet werden.

Die *Is-A*-Beziehung gilt allerdings nur in einer Richtung. Kein `ComicCharacter`-Objekt kann (ohne *Cast*) anstelle eines `SuperHero`-Objekts verwendet werden. Der Grund dafür ist der, daß ein `ComicCharacter` kein `SuperHero` ist - ein `ComicCharacter` kann weder kämpfen, noch enthält er eine Komponente `superPower`.

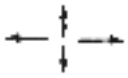
Als Vergleich dazu soll ein Beispiel aus dem Alltagsleben dienen: Von einer Basisklasse `Säugetier` könnten die verschiedenen Säugetiere wie etwa `Wal`, `Mensch` abgeleitet werden. Jeder `Wal` ist auch ein `Säugetier`. Umgekehrt gilt aber nicht, daß jedes `Säugetier` ein `Wal` ist. Folgendes Codestück liefert daher Fehler beim Übersetzen:

```
void fooVal(const SuperHero o)
{
  o.print();
}
...
ComicCharacter  c("Speedy Gonzalez");

fooVal(c);  // Fehler!
```

Ebenso wie Objekte einer abgeleiteten Klasse anstelle von Objekten ihrer Basisklasse verwendet werden können, ist dies auch mit Zeigern auf Objekte möglich (siehe auch Abschnitt 9.4.1). Ein Zeiger auf `SuperHero` wird vom System automatisch als Zeiger auf `ComicCharacter` aufgefaßt, wenn dies nötig ist. Ebenso wie bei Klassen gilt auch hier, daß der umgekehrte Weg nicht möglich ist. Um einen Zeiger auf `ComicCharacter` in einen Zeiger auf `SuperHero` umzuwandeln, muß ein *Cast* verwendet werden.

Im folgenden Beispiel wird über eine Variable vom Typ „Zeiger auf `ComicCharacter``` auch auf ein `SuperHero`-Objekt zugegriffen. Das ist möglich, weil `SuperHero` von `ComicCharacter` abgeleitet ist und zwischen den beiden Objekten daher eine *Is-A*-Beziehung besteht.

```
ComicCharacter  cc("Speedy Gonzales");
ComicCharacter* pCC;
SuperHero       sh("Speed");

pCC = &cc;
cc.print();
pCC->sing();
pCC = &sh;
pCC->print();
sh.fight();
pCC->fight();  // Fehler! Unmoeglich, da pCC ein
               // Zeiger auf ComicCharacter ist
```

Zwischen der Funktion `fooVal` auf Seite  und folgender Funktion `fooRef` besteht ein wesentlicher Unterschied:

```
void fooRef(const ComicCharacter& o)
{
  o.print();
}
```

`fooVal` weist einen *Call by Value*-Parameter auf, das heißt, der Aktualparameter wird bei der Übergabe *kopiert*. Beim Kopieren eines Objekts wird vom System entsprechend Speicher angefordert und dann das neue Objekt mit dem *Copy*-Konstruktor initialisiert. Der Typ des neuen Objekts ist durch den Formalparameter bestimmt.

Unabhängig vom konkreten Typ des jeweiligen Aktualparameters ist der Formalparameter `o` ein Objekt der Klasse `ComicCharacter`. Das System erzeugt automatisch Code zum Kopieren eines `ComicCharacter`-Objekts. Beim Kopieren werden nur die `ComicCharacter`-Teile berücksichtigt. Dieser Umstand wird auch als *Slicing Problem* bezeichnet: Durch das Kopieren wird alles „Überflüssige`` weggeschnitten, übrig bleibt ein reines `ComicCharacter`-Objekt. Das Daten-Element `superPower` und die `SuperHero`-Methode `fight` sind „verloren``.

Will man das vermeiden und das (meist unnötige) Kopieren von Objekten umgehen, so muß der Parameter mit einem Zeiger oder einer Referenz übergeben werden. Wie bereits in Abschnitt 8.5

angegeben, werden Parameter hier grundsätzlich als Referenzen deklariert (Eingangsparameter als Referenzen auf `const`-Typen).

Der Aufruf

```
fooRef(s);
```

hilft damit nicht nur, Laufzeit einzusparen. Er führt auch dazu, daß das übergebene Objekt auch in der Funktion ein „echter`` `SuperHero` ist und nicht zu einem „normalen`` `ComicCharacter` verkümmert.

Vorige Seite: 13.2 Ableiten einer Klasse **Eine Ebene höher:** 13 Vererbung **Nächste Seite:** 13.4 Vererbung und Gültigkeitsbereiche

Vorige Seite: 13.3 Die *Is-A*-Beziehung **Eine Ebene höher:** 13 Vererbung **Nächste Seite:** 13.5 Element-Funktionen bei abgeleiteten

13.4 Vererbung und Gültigkeitsbereiche

Abbildung 13.3 zeigt, daß jedes Objekt alle Komponenten der Basisklassen enthält. Jede Klasse vereinbart einen eigenen Gültigkeitsbereich. Der Gültigkeitsbereich `ComicCharacter` zum Beispiel enthält unter anderem die dort vereinbarten Elemente `name`, `sing` und `print`. `SuperHero` umfaßt die Elemente `superPower` und `fight` und alle ererbten Elemente.

Auch in abgeleiteten Klassen können die Gültigkeitsbereiche für Bezeichner explizit angegeben werden. Folgendes Beispiel vereinbart drei Klassen `A`, `B` und `C` und referenziert verschiedene Elemente über Angabe des Gültigkeitsbereichs:

```
class A
{
public:
  int j;
};

class B : public A
{
public:
  int i;
};

class C : public B
{
public:
  void foo();
  int i, j;
};

void C::foo()
{
  cout << i;      // Ausgabe C::i
  cout << j;      // Ausgabe C::j
  cout << B::i;   // Ausgabe B::i
  cout << B::j;   // Ausgabe A::j, B erbt j und
```

```
                  // ueberdeckt es nicht
  cout << A::i;   // Fehler, A hat kein i und erbt
                  // auch keines
  cout << A::j;   // Ausgabe A::j
}
```

A hat ein Datenelement `j`, B ein Datenelement `i`. Damit sind im Gültigkeitsbereich B zwei Datenelemente sichtbar: `j` (ererbt) und `i`. In der Methode `foo` wird bei der Ausgabe von `j` daher das ererbte Element ausgegeben. In der Klasse C werden zwei Datenelemente `i` und `j` vereinbart. Diese überdecken die ererbten Elemente, die Ausgabe von `C::i` und `C::j` führt daher zur Ausgabe der in C vereinbarten Elemente.

Die Anweisung `cout << A::i` hingegen führt zu einem Fehler beim Übersetzen, da A ein Element `i` weder hat noch erbt.

Vorige Seite: 13.4 Vererbung und Gültigkeitsbereiche **Eine Ebene höher:** 13 Vererbung **Nächste Seite:** 13.5.1 Konstruktoren

13.5 Element-Funktionen bei abgeleiteten Klassen

Abgeleitete Klassen sind, wie bereits erwähnt, eigenständige Einheiten, die Methoden und Daten enthalten können. Allerdings gilt es, einige Besonderheiten vor allem im Zusammenhang mit Konstruktoren, Destruktoren und Zuweisungsoperatoren sowie beim Überschreiben von ererbten Methoden zu beachten.

Vorige Seite: 13.5 Element-Funktionen bei abgeleiteten **Eine Ebene höher:** 13.5 Element-Funktionen bei abgeleiteten **Nächste Seite:** 13.5.2 *Copy*-Konstruktor

13.5.1 Konstruktoren

Wird für die Klasse `SuperHero` ein Konstruktor definiert, so stellt sich die Frage nach der Initialisierung der ererbten Datenelemente.

In einem Konstruktor müssen alle Elemente eines Objekts (also auch die ererbten) initialisiert werden. Die direkte Initialisierung oder Zuweisung aller Elemente in einem Konstruktor ist aber nicht unbedingt zielführend. Man stelle sich den Aufwand vor, in jedem Konstruktor jeder neuen Klasse alle (!) ererbten Elemente zu initialisieren.

Statt dessen läuft die Initialisierung nach einem auch *Chaining* genannten Schema ab: Jede Klasse erledigt ,,nur`` die eigenen Aufgaben. Aufgaben, die ererbte Methoden übernehmen können, werden an diese delegiert. Da jede Klasse ohnehin Konstruktoren zur Initialisierung der eigenen Datenelemente aufweist, werden diese verwendet und in einem neuen Konstruktor nur die eigenen Datenelemente initialisiert.

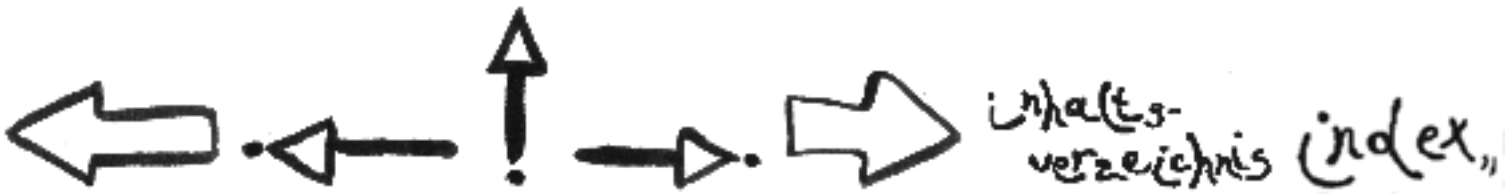
Zu beachten ist, daß die Datenelemente der Basisklassen *vor* den eigenen Datenelementen initialisiert werden müssen, da abgeleitete Klassen auf ihren Basisklassen aufbauen. Jeder Konstruktor ruft daher zuerst den Konstruktor der direkten Basisklasse auf und initialisiert dann die eigenen Daten. Der Konstruktor der Basisklasse arbeitet nach demselben Schema: Gibt es eine Basisklasse, so werden zuerst deren Konstruktor und erst dann die eigenen Daten initialisiert.

Der Aufruf des Basisklassen-Konstruktors erfolgt nicht im Anweisungsteil, sondern in der Initialisierungsliste eines Konstruktors. Ein Konstruktor für die Klasse `SuperHero` kann damit wie folgt angegeben werden:

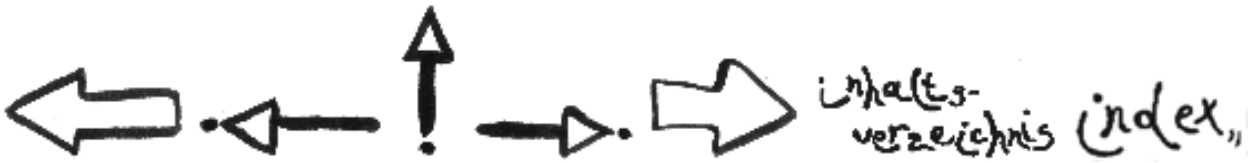
```
...
SuperHero(const TString& aName="",
          const TString& aSuperPower = "")
 : ComicCharacter(aName), superPower(aSuperPower) {};
...
```

Falls kein Aufruf eines Basisklassen-Konstruktors in der Initialisierungsliste eines Konstruktors erscheint, so fügt der Compiler automatisch einen Aufruf des *Default*-Konstruktors der Basisklasse ein. Auch der vom System erzeugte *Default*-Konstruktor weist einen derartigen Aufruf auf. Ein automatisch erzeugter *Default*-Konstruktor für `SuperHero` sieht damit wie folgt aus:

```
SuperHero()
    : ComicCharacter() {};
```



Vorige Seite: 13.5 Element-Funktionen bei abgeleiteten **Eine Ebene höher:** 13.5 Element-Funktionen bei abgeleiteten **Nächste Seite:** 13.5.2 *Copy*-Konstruktor



Vorige Seite: 13.5.1 Konstruktoren **Eine Ebene höher:** 13.5 Element-Funktionen bei abgeleiteten **Nächste Seite:** 13.5.3 Destruktor

---

13.5.2 *Copy*-Konstruktor

Copy-Konstruktoren werden auch für abgeleitete Klassen automatisch erzeugt. Die automatisch erzeugten *Copy*-Konstruktoren initialisieren zunächst die Basisklassenkomponenten durch einen Aufruf des *Copy*-Konstruktors der direkten Basisklasse und dann die eigenen Elemente durch Zuweisung.

Wie bei ,,normalen`` Klassen gilt also auch in diesem Fall, daß der erzeugte Konstruktor dann ,,falsch`` ist, wenn *in der eigenen Klasse* dynamische Datenstrukturen verwendet werden. Die Basisklassen sind für dieses Kriterium belanglos - jede Klasse ist ausschließlich für ihre eigenen Elemente zuständig. Sind alle Klassen in kanonischer Form, so kann davon ausgegangen werden, daß die Basisklassen ohnehin korrekt initialisiert werden!

Die automatisch erzeugten *Copy*-Konstruktoren für `SuperHero` und `SuperMan` sehen wie folgt aus:

```
class SuperHero : public ComicCharacter
{
  SuperHero(const SuperHero& o)
   : ComicCharacter(o), superPower(o.superPower) {};
  ...
};

class Superman : public SuperHero
{
  Superman(const Superman& o)
   : SuperHero(o) {};
  ...
};
```

Beide *Copy*-Konstruktoren sind ,,richtig``. `TString`-Objekte aus Kapitel 11 sind zwar (wie wir wissen) Objekte mit dynamischen Datenstrukturen, allerdings ist die Klasse `TString` in kanonischer Form. Damit erzeugt die Initialisierung von `superPower` im *Copy*-Konstruktor der Klasse `SuperHero` eine echte Kopie. Bereits hier wird klar, wie wichtig die konsequente Befolgung von Richtlinien wie die zur Realisierung von Klassen in kanonischer Form ist.

Im folgenden Programm werden von `SuperHero` zwei Klassen `Batman` und `Superman` abgeleitet. Alle drei Klassen sind mit entsprechenden Konstruktoren (und Destruktoren) versehen:

Programm 13.3: Klassen `Superhero`, `Superman` und `Batman`

```
class SuperHero : public ComicCharacter
{
 public:
  SuperHero(const TString& aName,
            const TString& aSuperPower = "")
   : ComicCharacter(aName), superPower(aSuperPower) {};
```

```
  virtual ~SuperHero() {};
  virtual void fight() { cout << name
    << " fights... \n";
  };
 private:
  TString superPower;
};

class Batman : public SuperHero
{
 public:
  Batman()
   : SuperHero("Batman", "") {};
  virtual ~Batman() {};
};

class Superman : public SuperHero
{
 public:
  Superman()
   : SuperHero("Superman", "Flying, ...") {};
  virtual ~Superman() {};
};
```

Die Deklaration `Superman s;` vereinbart ein `Superman`-Objekt, dessen Initialisierung wie in Tabelle 13.2 abläuft.

in Objekt	Aktion
Superman	Im Konstruktor wird der Konstruktor der Basisklasse `SuperHero` aufgerufen.
Superhero	Der Konstruktor der Basisklasse `ComicCharacter` wird aufgerufen.
ComicCharacter	Der `ComicCharacter`-Konstruktor hat keine Basisklasse und initialisiert daher „nur`` die eigenen Daten (`name`).
Superhero	Die eigenen Daten werden initialisiert (`superPower`).
Superman	`Superman` hat keine eigenen Daten, die Initialisierung ist beendet.

Tabelle 13.2: Ablauf einer Objekt-Initialisierung bei abgeleiteten Klassen

Vorige Seite: 13.5.1 Konstruktoren **Eine Ebene höher:** 13.5 Element-Funktionen bei abgeleiteten **Nächste Seite:** 13.5.3 Destruktor

Vorige Seite: 13.5.2 *Copy*-Konstruktor **Eine Ebene höher:** 13.5 Element-Funktionen bei abgeleiteten
Nächste Seite: 13.5.4 Zuweisungsoperator

13.5.3 Destruktor

Destruktoren werden im Fall von abgeleiteten Klassen ebenfalls nach dem bereits bekannten *Chaining*-Prinzip realisiert: jede Klasse kümmert sich um die eigenen Datenelemente und überläßt die ererbten Elemente den Destruktoren der jeweiligen Basisklassen. Im Unterschied zu Konstruktoren werden Aufrufe der Basisklassen-Destruktoren aber nie explizit angeführt, das System führt sie automatisch durch.

Ein ,,leerer`` Destruktor der Art

```
~SuperHero() {};
```

ruft also automatisch den Basisklassen-Destruktor auf.

Zudem ist die Reihenfolge hier umgekehrt: *Erst* werden die eigenen dynamischen Speicherbereiche bereinigt und *dann* erfolgt der ,,Aufruf`` des Basisklassen-Destruktors. Der Grund dafür ist klar: Klassen bauen auf ihren Basisklassen auf, eigene Funktionalität bezieht sich damit auch auf die Funktionalität der Basisklasse. Erzeugt werden solche Objekte von innen nach außen: zuerst muß der Kern (= eine Instanz der obersten Basisklasse) existieren, dann können die darauf aufbauenden Schichten erzeugt werden. Die Zerstörung erfolgt natürlich in der umgekehrten Reihenfolge: zuerst die äußeren Schichten, dann die inneren.

Der Ablauf bei der Zerstörung eines `Superman`-Objekts (Programm 13.3) verläuft ähnlich wie bei der Konstruktion (vergleiche Tabelle 13.2), nur daß hier die Aufrufreihenfolge umgekehrt ist: Erst wird der Destruktor der Klasse `Superman` aufgerufen, dann der der Klasse `SuperHero` und anschließend der der Klasse `ComicCharacter`.

Vorige Seite: 13.5.2 *Copy*-Konstruktor **Eine Ebene höher:** 13.5 Element-Funktionen bei abgeleiteten
Nächste Seite: 13.5.4 Zuweisungsoperator

Vorige Seite: 13.5.3 Destruktor **Eine Ebene höher:** 13.5 Element-Funktionen bei abgeleiteten **Nächste Seite:** 13.5.5 Überschreiben von ererbten

13.5.4 Zuweisungsoperator

Die Implementierung des Zuweisungsoperators in abgeleiteten Klassen erfolgt ebenfalls nach dem *Chaining*-Schema. Wie im Fall des *Copy*-Konstruktors werden bei der eigentlichen Zuweisung die Elemente der Basisklassen durch einen Aufruf des Zuweisungsoperators der direkten Basisklasse zugewiesen.

Der vom System automatisch erzeugte Zuweisungsoperator für die Klassen `SuperHero` und `Superman` sieht wie folgt aus:

```
SuperHero& SuperHero::operator=(const SuperHero& o)
{
  ComicCharacter::operator=(o);  // Aufruf = d. Basisklasse
  superPower = o.superPower;     // Zuweisung der eigenen
                                 // Datenelemente
  return *this;                  // Rueckgabe eigenes Objekt
}

Superman& Superman::operator=(const Superman& o)
{
    // alternative Syntax:
  (reinterpret_cast<SuperHero&>*this) = o;
  ...
}
```

Anmerkung: Beide angeführten Arten, den Zuweisungsoperator der Basisklasse aufzurufen, führen zum selben Ergebnis. Im ersten Fall wird der Zuweisungsoperator aus dem Gültigkeitsbereich `ComicCharacter` (also der Basisklasse) aufgerufen, im zweiten Fall wird das eigene Objekt (`*this`) als ein `SuperHero`-Objekt betrachtet und dann der Zuweisungsoperator aufgerufen.

Wie auch schon beim *Copy*-Konstruktor gilt auch hier, daß beide automatisch erzeugten Zuweisungsoperatoren „richtig`` sind, da beide Klassen keine dynamischen Datenstrukturen enthalten und alle verwendeten Klassen in kanonischer Form sind. Enthält allerdings eine Klasse dynamische Datenstrukturen, so muß der Zuweisungsoperator neu implementiert werden.

Das in Abschnitt 11.7.3 angegebene Schema wird dazu wie folgt abgeändert:

1.

Test auf Zuweisung an sich selbst (`a = a`)

2. Aufruf des Zuweisungsoperators der Basisklasse
3. Zuweisung der eigenen Elemente:
   - „Normale`` Elemente zuweisen
   - Dynamische Speicherbereiche kopieren
4. Rückgabe einer Referenz auf eigenes Objekt

Ein Beispiel: Von einer Klasse A wird die Klasse B und von dieser die Klasse C abgeleitet. Der Ablauf einer Zuweisung eines Objekts der Klasse C an ein anderes ist in Tabelle 13.3 angegeben.

in Klasse	Aktion
C	Test auf Zuweisung an sich selbst, eventuell Abbruch
C	Zuweisung der Basisklassenkomponente (B)
B	Test auf Zuweisung an sich selbst, eventuell Abbruch
B	Zuweisung der Basisklassenkomponente (A)
A	Test auf Zuweisung an sich selbst, eventuell Abbruch
A	Keine Basisklasse, daher: Zuweisen der eigenen Elemente über = beziehungsweise Kopieren im Fall von dynamischen Datenstrukturen
A	Rückgabe einer Referenz auf eigenes Objekt (= A-Teil des Objekts)
B	Zuweisung der Basisklassenkomponente beendet, Zuweisen bzw. Kopieren der eigenen Elemente
B	Rückgabe einer Referenz auf eigenes Objekt (= B-Teil des Objekts)
C	Zuweisung der Basisklassenkomponente beendet, Zuweisen bzw. Kopieren der eigenen Elemente
C	Rückgabe einer Referenz auf eigenes Objekt (= gesamtes Objekt)

Tabelle 13.3: Ablauf einer Zuweisung von abgeleiteten Klassen

Vorige Seite: 13.5.3 Destruktor **Eine Ebene höher:** 13.5 Element-Funktionen bei abgeleiteten **Nächste Seite:** 13.5.5 Überschreiben von ererbten

Vorige Seite: 13.5.4 Zuweisungsoperator **Eine Ebene höher:** 13.5 Element-Funktionen bei abgeleiteten

13.5.5 Überschreiben von ererbten Methoden

Superhelden tanzen nicht und singen auch nicht, sie sind mit dem Kampf gegen das Böse ausreichend beschäftigt. Die Methoden `sing` und `dance` für Superhelden müssen daher geändert werden.

Dazu werden diese in der entsprechenden Klasse überschrieben (Überschreiben = Neu-Definieren von ererbten Methoden). Zu beachten ist hier, daß das Überschreiben von ererbten Methoden zu erheblichen Problemen führen kann, falls diese nicht als `virtual` vereinbart sind.

Auf die Bedeutung von `virtual` wird erst später eingegangen, vorerst aber soll als Grundregel festgehalten werden:

Alle Methoden einer Klasse, die in abgeleiteten Klassen überschrieben werden können, werden als virtual vereinbart, oder es werden in abgeleiteten Klassen nur Methoden überschrieben, die in den Basisklassen als `virtual` vereinbart sind.

Die Klasse `SuperHero` muß also wie unten angeführt neu definiert werden.

Programm 13.4: Klasse `SuperHero` (2)

```
class SuperHero : public ComicCharacter
{
public:
  SuperHero(const TString& aName,
            const TString& aSuperPower = "")
   : ComicCharacter(aName), superPower(aSuperPower) {};
  virtual ~SuperHero() {};
  virtual void fight() {
    cout << getName() << " fights... \n";
  };
  virtual void print() const {
    cout << "Superhero, name: " << getName()
    << ", special abilities: " << superPower << "\n";
  };
  virtual void dance() const {
    cout << getName() << " won't dance. \n";
  };
  virtual void sing() const {
    cout << getName() << " won't sing. \n";
  };
private:
  TString superPower;
```

```
};
```

Das folgende Codestück liefert in Verbindung mit den vorher angeführten Basisklassen die darunter angeführte Ausgabe:

```
void main()
{
  Batman   batm;
  Superman su;

  su.print();
  su.sing();
  batm.dance();
}
```

Ausgabe:

```
Superhero, name: Superman, special abilities: Flying, ...
Superman won't sing.
Batman won't dance.
```

Zusätzlich soll eine weitere Klasse `Duck` implementiert werden, die von `ComicCharacter` abgeleitet wird. Enten können schnattern und erhalten eine etwas abgeänderte Routine für die Ausgabe (`print`) (siehe Programm 13.5).

Programm 13.5: Klasse `Duck`

```
class Duck : public ComicCharacter
{
 public:
  Duck(const TString& aName = "")
   : ComicCharacter(aName+" Duck") {};
  virtual ~Duck() {};
  virtual void cackle { cout << name << " cackles... \n"; };
  virtual void print() { cout << "Duck, " << name
                              << " Duck.\n";
  };
};
```

Die Klassenhierarchie sieht damit wie in Abbildung 13.4 dargestellt aus.

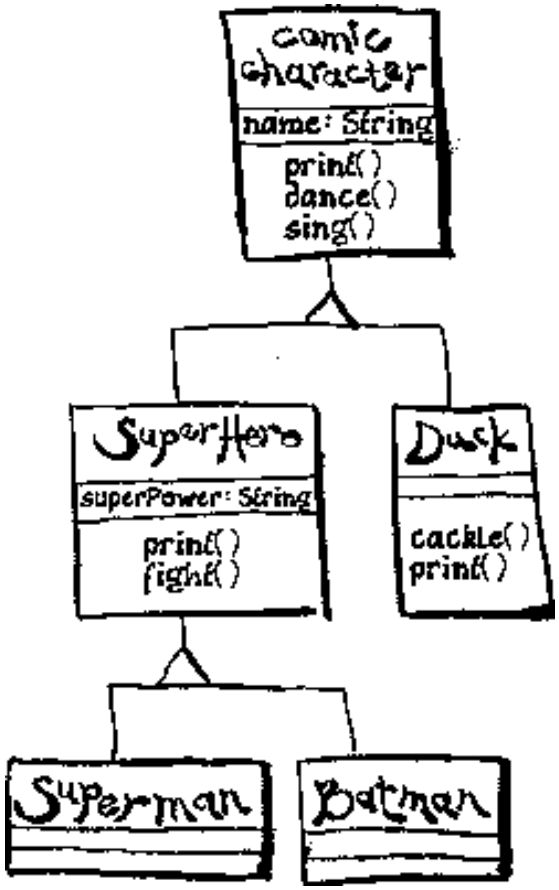


Abbildung 13.4: Klassenhierarchie `ComicCharacter`

Anmerkung: Das Überschreiben von Methoden und virtuelle Methoden im allgemeinen werden in Kapitel 14 noch gesondert behandelt.

Vorige Seite: 13.5.5 Überschreiben von ererbten **Eine Ebene höher:** 13 Vererbung **Nächste Seite:** 13.7 Beispiele und Übungen

13.6 Spezifikation von Basisklassen

Wird eine Klasse von einer Basisklasse abgeleitet, so kann die Basisklasse als `public`, `protected` oder `private` spezifiziert werden.

Bei der Ableitung einer Klasse von einer `public`-Basisklasse bleibt der Zugriffsschutz der ererbten Elemente erhalten: `public`- und `protected`-Elemente sind auch in der neuen Klasse `public` oder `protected` und auf `private`-Elemente der Basisklasse kann nicht zugegriffen werden. Wird eine Basisklasse als `protected` spezifiziert, so werden alle `public`-Elemente der Basisklasse zu `protected`-Elementen, bei einer `private`-Basisklasse werden alle ererbten Elemente zu `private`-Elementen.

Die Konsequenzen der Spezifikation von Basisklassen als `protected` oder `private` sind:

- Die ererbten `public`-Elemente können nicht mehr von außen angesprochen werden (`protected`-Basisklasse) beziehungsweise auch nicht mehr von abgeleiteten Klassen (`private`-Basisklasse).
- Es findet keine implizite Konversion zwischen abgeleiteten Klassen und ihren Basisklassen mehr statt. Würde zum Beispiel `ComicCharacter` bei der Ableitung von `SuperHero` als `protected`-Basisklasse definiert, so könnten in einer Funktion

  ```
  draw(const ComicCharacter& aComicChar);
  ```

  `SuperHero`-Objekte nur mittels explizitem *Cast* als Argument verwendet werden.

Die *Is-A*-Beziehung zwischen Basisklasse und abgeleiteter Klasse ist damit zerstört. Das ist auch der Hauptgrund, warum derartige Ableitungen eher selten sind. Eine Ableitung, die zu inkompatiblen Typen führt, bezeichnet man als *Implementierungs-Vererbung*. Ihr einziger Zweck ist eine Vereinfachung der Implementierung und nicht der, eine Hierarchie von Typen aufzubauen (*Interface-Vererbung*).

Basisklassen werden als `protected` vereinbart, wenn das `public`- und `protected`-*Interface* der Basisklasse für alle abgeleiteten Klassen (also innerhalb der Hierarchie) erhalten bleiben soll, nach außen hin aber nur das `public`-*Interface* der abgeleiteten Klasse sichtbar sein soll. Die *Is-A*-Beziehung gilt damit nur mehr innerhalb der Ableitungshierarchie.

Basisklassen werden als `private` vereinbart, wenn die *Is-A*-Beziehung, die bei der Ableitung einer Klasse von einer `public`-Basisklasse gilt, vollständig aufgehoben werden soll. Die abgeleitete Klasse dient nach außen hin und innerhalb der Hierarchie als eine vollständig neue Klasse, es kann außerhalb

der abgeleiteten Klasse auf keinerlei ererbte Elemente zugegriffen werden.

Ein Anwendungsbeispiel für `private`- oder `protected`-Basisklassen ist etwa die Ableitung einer Klasse `Stack` von einer Klasse `Vector`: `Vector` bietet für einen `Stack` alle Möglichkeiten, die Operationen `push` und `pop` einfach zu implementieren. Das Problem ist, daß auf einen `Stack` nur Zugriffe mittels `push` und `pop` erwünscht sind, die Basisklasse `Vector` aber wesentlich mehr Operationen zur Verfügung stellt (`[]`, `firstElem`, `lastElem` etc.). Wird `Stack` `public` abgeleitet, so stehen alle diese Operationen sowohl von außen als auch in abgeleiteten Klassen zur Verfügung. Bei der Definition von `Vector` als `protected`-Basisklasse können die Zugriffe `push` und `pop` nur mehr innerhalb der Hierarchie umgangen werden. Wird `Vector` als `private`-Basisklasse vereinbart, so können auch abgeleitete Klassen nur über das *Interface* von `Stack` zugreifen.

Derselbe Effekt kann allerdings auch erzielt werden, indem `Stack` nicht von `Vector` abgeleitet wird, sondern ein `Vector`-Objekt als Instanzvariable enthält. In diesem Fall werden nur die Operationen `push` und `pop` definiert, die die `Stack`-Funktionalität mittels der `privaten` Instanzvariable realisieren.

Die Frage, die sich hier stellt, ist, ob ein `Stack` ein `Vector` ist (*Is-A*-Beziehung) oder nur einen `Vector` benutzt (`Uses`-Beziehung) oder enthält (*Has*-Beziehung). Vererbung ist ein sehr mächtiges Konzept in der objektorientierten Programmierung, aber kein Wundermittel. Es sollte nicht benutzt werden, um ,,Benutzt``- oder ,,Enthält``- Beziehungen zu simulieren.

Vorige Seite: 13.5.5 Überschreiben von ererbten **Eine Ebene höher:** 13 Vererbung **Nächste Seite:** 13.7 Beispiele und Übungen

Vorige Seite: 13.6 Spezifikation von Basisklassen **Eine Ebene höher:** 13 Vererbung **Nächste Seite:** 13.7.1 Klasse `Word`

13.7 Beispiele und Übungen

Da Vererbung ohne Polymorphismus nur bedingt von Interesse ist, wird in diesem Abschnitt nur ein einzelnes Übungsbeispiel (`Word`) angeführt.

Vorige Seite: 13.7 Beispiele und Übungen **Eine Ebene höher:** 13.7 Beispiele und Übungen

---

Teilabschnitte

13.7.1 Klasse `Word`

Themenbereich

Vererbung

Aufgabenstellung

Von der Klasse `TString` ist eine Klasse `Word` abzuleiten, die über folgende Funktionalität verfügt:

- `Word` repräsentiert die Stammform und die verschiedenen Wortformen eines Worts (zum Beispiel Buch, Bücher, Buchs). Die verschiedenen Wortformen speichert jedes Wort in einer verketteten Liste.
- `Word` kann Zeichenketten (`TString`-Objekte) zu dieser Liste hinzufügen und entfernen.
- `Word` verfügt über eine Vergleichsfunktion `match`, die angibt, ob eine vorgegebene Zeichenkette mit einer der Wortformen übereinstimmt.
- Der Eingabeoperator von `Word` liest die Stammform des Worts und beliebig viele (auch 0) weitere Formen ein. Das Eingabeformat ist unten angeführt. Der Ausgabeoperator gibt das Wort und seine verschiedenen Formen aus.

Entwerfen und implementieren Sie die Klasse `Word`. Beantworten Sie folgende Fragen:

- Welche Instanzvariablen enthält `Word`? Welche Sichtbarkeit (`public`, `protected`, `private`) besitzen diese? Warum?
- Wie wird `Word` von `TString` abgeleitetet (`public`, `protected`, `private`)? Warum?
- Ist `Word` in kanonischer Form? Wenn nein, welche Methoden müssen implementiert werden?

Testen Sie die Klasse `Word`, indem Sie eine Klasse `SpellChecker` erstellen, die von einer Wortdatei verschiedene Wörter (inklusive Wortformen) einliest und mit diesen Daten einen Text (der von einer

zweiten Datei einzulesen ist) auf unbekannte Wörter hin überprüft. Der zu überprüfende Text kann auch Satzzeichen oder andere Sonderzeichen enthalten. `SpellChecker` ignoriert alle derartigen Zeichen.

Die Klasse `SpellChecker` weist unter anderem folgende Methoden auf:

```
  // Wort-DB einlesen
int initDatabase(const TString& fName);
  // Text pruefen
int checkString(const TString& w);
```

`SpellChecker` verwaltet die eingelesenen Wörter in einer Liste oder einer ähnlichen dynamischen Datenstruktur. Auch bei `SpellChecker` sind die Fragen bezüglich kanonischer Form und den daraus resultierenden Konsequenzen zu beantworten.

Dem eigentlichen Testprogramm `main` werden die Namen der beiden Dateien als Kommandozeilenargumente (siehe Anhang C) übergeben. Der erste Parameter gibt den Namen der Textdatei, der zweite den der Wortdatei an. Sie können von der Korrektheit der beiden Dateien ausgehen. Der konkrete Aufbau der Dateien ist unten angeführt.

Aufbau einer Wortdatei:

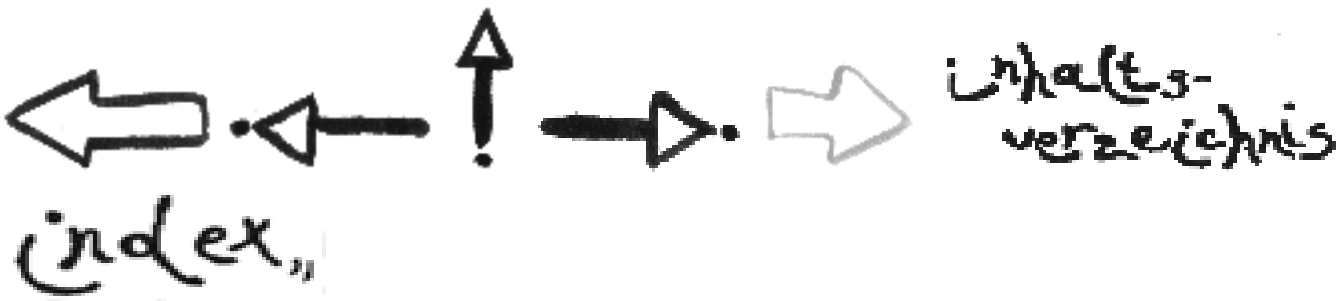
```
Wort ' ' Form1 ' ' Form2 EndOfLine
Wort ' ' Form1 ' ' Form2 ' ' Form3 ... EndOfLine
...
Wort EndOfFile
```

Beispiel für eine konkrete Wortdatei:

```
Test Tests ~
dort ~
Gang Gange G"ange G"angen ~
```

Dateiaufbau einer Textdatei:

```
Wort Wort Wort
```



Vorige Seite: 13.7 Beispiele und Übungen **Eine Ebene höher:** 13.7 Beispiele und Übungen

Vorige Seite: Aufgabenstellung **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** 14.1 Polymorphismus

---

14 Polymorphismus und spezielle Aspekte der Vererbung

In den bisherigen Kapiteln wurden die ,,statischen`` Aspekte von C++ behandelt. Dieses Kapitel beschreibt die dynamischen objektorientierten Sprachmerkmale von C++. Erst durch diese wird C++ zu einer ,,echten`` objektorientierten Programmiersprache. Das Kapitel ist wie folgt gegliedert:

- Zunächst erfolgt eine kurze Erklärung des Begriffs *Polymorphismus* und des Konzepts der virtuellen Element-Funktionen, die Polymorphismus in C++ realisieren.
- Klassen, die nur als Mittel der Abstraktion und nicht zur Vereinbarung von Objekten verwendet werden, sind im nächsten Abschnitt beschrieben.
- Im Anschluß daran wird Mehrfachvererbung, die Ableitung einer Klasse von mehr als einer Basisklasse, dargelegt.
- Abschnitt 14.5 behandelt virtuelle Basisklassen.
- Abschnitt 14.6 bespricht einen weiteren Aspekt von polymorphen Klassen, die Laufzeit-Typinformation.

Vorige Seite: 14 Polymorphismus und spezielle **Eine Ebene höher:** 14 Polymorphismus und spezielle
Nächste Seite: 14.2 Virtuelle Element-Funktionen

14.1 Polymorphismus

Werden von einer Klasse `A` die Klassen `B` und `C` abgeleitet, so können Objekte vom Typ ,,Zeiger auf `A``` auch auf `B`- oder `C`-Objekte verweisen. Implementieren alle drei Klassen eine Operation `foo` jeweils verschieden, so bewirkt die Anweisung

```
anAPointer->foo();
```

in ,,normalen`` Programmiersprachen den Aufruf von `A::foo`: Zur Übersetzungszeit bindet der Compiler die Funktion `foo` der Klasse `A`, weil die Variable `anAPointer` vom Typ `A` ist.

Man nennt diese Art des Bindens *Statische Bindung* , da sie statisch (das heißt unveränderbar) ist. In diesem Zusammenhang wird oft auch der Begriff *Early Binding* (Frühes Binden) verwendet. *Early Binding* heißt, daß die Bindung zum frühest möglichen Zeitpunkt, also zur Übersetzungszeit, erfolgt.

Die Variable `anAPointer` kann in einer objektorientierten Programmiersprache auch andere ,,Gestalten`` annehmen, indem sie auf Objekte der abgeleiteten Klassen verweist und somit eigentlich ein `B`- oder `C`-Objekt darstellt.

In ,,echten`` objektorientierten Programmiersprachen werden derartige Aufrufe nicht zur Übersetzungszeit, sondern erst zur Laufzeit gebunden: Beim Aufruf untersucht das System den jeweiligen Typ des Objekts, auf das die Methode angewandt werden soll. In Abhängigkeit von diesem Typ wird dann die aufzurufende Methode (`A::foo`, `B::foo` oder `C::foo`) selektiert. Diese Art der Bindung wird als *Dynamische Bindung* bezeichnet. Da die Methoden erst bei der Ausführung gebunden werden, spricht man auch von *Late Binding* (Spätes Binden).

Der Begriff *Polymorphismus* bedeutet ,,Vielgestaltigkeit`` und umschreibt das oben angeführte Verhalten. Er kann als die wesentliche Eigenschaft gesehen werden, die echte objektorientierte Systeme von modulbasierten oder objektbasierten unterscheidet.

Polymorphismus ist ein Konzept der Typtheorie in der Informatik, das in Verbindung mit dynamischer Bindung hilft, Softwaresysteme flexibel und einfach erweiterbar zu gestalten. So können etwa (vereinfacht gesagt) in einem Grafikprogramm alle grafischen Objekte von einer gemeinsamen Klasse abgeleitet werden. Diese gemeinsame Basisklasse definiert die Methoden, die auf alle Grafik-Objekte angewandt werden können (`draw`, `move`, `setPos`). Die einzelnen Klassen überschreiben die ererbten Methoden entsprechend. Ein Aufruf von `draw` (über eine ,,polymorphe Variable``) bewirkt damit, daß in Abhängigkeit vom aktuellen Objekt (oder besser: von seiner Klasse) ein Dreieck, Viereck, Polygon etc. gezeichnet wird.

Damit werden alle sonst nötigen `CASE`- beziehungsweise `switch`-Anweisungen „überflüssig``, und das Programmsystem ist dadurch flexibler und einfacher erweiterbar. Jede neue Grafik-Klasse wird ebenfalls von der gemeinsamen Basisklasse abgeleitet und kann damit ohne weitere Änderungen in das Programmsystem eingebunden werden.

Vorige Seite: 14 Polymorphismus und spezielle **Eine Ebene höher:** 14 Polymorphismus und spezielle
Nächste Seite: 14.2 Virtuelle Element-Funktionen

Vorige Seite: 14.1 Polymorphismus **Eine Ebene höher:** 14 Polymorphismus und spezielle **Nächste Seite:** 14.2.1 Deklaration von virtuellen

14.2 Virtuelle Element-Funktionen

Virtuelle Element-Funktionen sind spezielle Funktionen, die nicht zur Übersetzungs-, sondern zur Laufzeit gebunden werden. Das heißt, daß erst beim Aufruf einer virtuellen Element-Funktion entschieden wird, welche Funktion tatsächlich ausgeführt wird.

Dazu ein Beispiel: In Kapitel 13 wurde eine Reihe von Klassen eingeführt, die alle über eine Element-Funktion `print` verfügen. Diese Funktion druckt den Namen der entsprechenden Comic-Figur sowie eventuell vorhandene spezielle Fähigkeiten aus.

In einem Programm zur Verwaltung aller Comic-Figuren soll eine Liste verwendet werden, in der alle Objekte dieser Klassen abgelegt sind. Dazu genügt es, eine Liste von `ComicCharacter`-Objekten anzulegen. Da alle Objekte Ausprägungen der Klasse `ComicCharacter` oder davon abgeleiteter Klassen sind, sind sie auch `ComicCharacter` (*Is-A*-Beziehung). In einer Funktion `printAll` soll die gesamte Liste ausgegeben werden. Zur Implementierung der Liste werden die Klassen aus den obigen Beispielen, das Klassen-Template `List` (siehe Kapitel 12) und die Klasse `TString` (siehe Kapitel 11) verwendet.

Programm 14.1: Polymorphe Comic-Figurenliste

```
#include <iostream>
#include "tstring.h"
#include "inheri2.h"
#include "listmpl2.h"

typedef List<ComicCharacter*>  CCharPtrList;

void printAll(const CCharPtrList& theCList)
{
   CCharPtrList::Iterator  it(theCList);
   while (!it.isAtEnd()) {
      it.element()->print();
      ++it;
   }
}

void main()
{
   Duck*         donald= new Duck("Donald");
   Duck*         dagobert= new Duck("Dagobert");
   SuperHero*    redlight= new SuperHero("Red Lightning",
                                         "Speed");
   Batman*       batman = new Batman;
   Duck*         lascher= new  Duck("Lascher");
   CCharPtrList  comicPtrList;

   comicPtrList.add(dagobert);
   comicPtrList.add(redlight);
```

```
comicPtrList.add(batman);
comicPtrList.add(donald);
comicPtrList.add(lascher);
printAll(comicPtrList);
...
```

In einem „normalen`` Programm würde mit dem oben angeführten Code immer der Name der Comic-Figuren in der Liste ausgegeben werden. Die Liste wurde als `List<ComicCharacter*>` definiert und beim Aufruf von `print` müßte daher eigentlich die entsprechende Element-Funktion der Klasse `ComicCharacter` ausgeführt werden. Die Ausgabe des Programms sieht aber so aus:

```
Duck, Dagobert Duck.
Superhero, name: Red Lightning, special abilities: Speed
Superhero, name: Batman, special abilities:
Duck, Donald Duck.
Duck, Lascher Duck.
```

Die Erklärung dazu ist einfach. Da die Element-Funktionen `print` als virtuell deklariert sind, wird ein entsprechender Aufruf erst zur Laufzeit aufgelöst. Das heißt in unserem Fall, daß `printAll` wie folgt abläuft: Die Methoden `isAtEnd`, `++` und `element` des vereinbarten Iterators sind statisch (zur Übersetzungszeit) gebunden und werden beim Aufruf sofort ausgeführt. Das Ergebnis des Aufrufs von `element()` ist ein Zeiger auf das aktuelle Element in der Liste. An dieses Objekt wird die virtuelle Methode `print` geschickt. Erst jetzt (also nicht zum Übersetzungszeitpunkt, sondern beim Methodenaufruf) wird entschieden, welche Methode aktiviert wird. Je nachdem, ob das Objekt eine Instanz der Klasse `ComicCharacter`, `Duck` oder `SuperHero` ist, wird `ComicCharacter::print()`, `Duck::print()` oder `SuperHero::print()` aufgerufen.

Vorige Seite: 14.1 Polymorphismus **Eine Ebene höher:** 14 Polymorphismus und spezielle **Nächste Seite:** 14.2.1 Deklaration von virtuellen

Vorige Seite: 14.2 Virtuelle Element-Funktionen **Eine Ebene höher:** 14.2 Virtuelle Element-Funktionen
Nächste Seite: 14.2.2 Aufruf von virtuellen

14.2.1 Deklaration von virtuellen Element-Funktionen

Um eine Element-Funktion als virtuell zu definieren, wird das Funktionsattribut `virtual` (siehe Abschnitt 9.2.3) verwendet. Klassen, die virtuelle Element-Funktionen enthalten, werden als *polymorphe Klassen* bezeichnet.

```
class A
{
public:
  virtual A* foo() { ... };
};

class B : public A
{
public:
  virtual A* foo() { ... };  // ueberschreibt A::foo,
                             // virtuell
};
```

Beim Überschreiben von (virtuellen) Methoden ist wesentlich, daß die neue Methode dieselbe Signatur wie die Methode der Basisklasse aufweist. Liegen verschiedene Signaturen vor, so wird eine neue Methode eingeführt und nicht die Methode der Basisklasse überschrieben!

In bezug auf den Rückgabewert besteht allerdings eine gewisse Freiheit. So kann die Methode `foo` der Klasse `B` so wie die Methode `foo` der Basisklasse `A` ein Objekt vom Typ `A*` zurückgeben oder aber einen Zeiger auf eine von `A` abgeleitete Klasse:

```
class B : public A
{
public:
  virtual B* foo() { ... };  // Covarianter Rueckgabewert
};
```

Zu beachten ist, daß beim Überschreiben von virtuellen Methoden das Schlüsselwort `virtual` nicht nötig ist:

```
class B : public A
```

```
{
public:
  B* foo() { ... };  // Virtuelle Methode, da die Basis-
};                   // klassen-Methode virtuell ist
```

Nur die oberste Element-Funktion (= erste in der Hierarchie) muß als `virtual` gekennzeichnet werden. „Überflüssige`` `virtual`s werden vom Compiler ignoriert.

- Überschriebene virtuelle Element-Funktionen müssen nicht explizit als `virtual` gekennzeichnet sein. Das Weglassen von `virtual` hat allerdings keine Vorteile und führt zu schlechterer Lesbarkeit.

Vorige Seite: 14.2 Virtuelle Element-Funktionen **Eine Ebene höher:** 14.2 Virtuelle Element-Funktionen
Nächste Seite: 14.2.2 Aufruf von virtuellen

Vorige Seite: 14.2.1 Deklaration von virtuellen **Eine Ebene höher:** 14.2 Virtuelle Element-Funktionen
Nächste Seite: 14.2.3 Virtuelle Destruktoren

14.2.2 Aufruf von virtuellen Element-Funktionen

Virtuelle Element-Funktionen können dynamisch oder statisch aufgerufen werden. Dynamisch bedeutet, daß die Methoden unter Berücksichtigung des tatsächlichen Typs eines Objekts *zur Laufzeit* aufgerufen werden. Eine statische Methodenauflösung ist eine Auflösung zur Übersetzungszeit.

Eine dynamische Methodenauflösung erfolgt entweder über Zeiger oder aber über Referenzen:

```
void printCC1(const ComicCharacter& c)
{
  c.print();  // dynamische Aufloesung
}

void printCC2(const ComicCharacter* p)
{
  p->print();  // dynamische Aufloesung
}
```

Im Gegensatz dazu wird ein Aufruf mit einem Objekt und der Punktnotation statisch aufgelöst. Der Grund liegt darin, daß ein ,,echtes`` Objekt seinen Typ nicht verändern kann (da es nicht polymorph ist) und der Compiler so schon zur Übersetzungszeit entscheiden kann, welche Element-Funktion aufzurufen ist.

```
Duck      dagobert("Dagobert");
SuperHero redlight("Red Lightning", "Speed");
Batman    batman;

batman.print();   // Statische Aufloesung
redlight.print();
dagobert.print();
```

Eine statische Auflösung wird auch dann erzwungen, wenn der Gültigkeitsbereich einer Methode explizit angegeben wird. Auch in diesem Fall löst der Compiler den Aufruf bereits zur Übersetzungszeit auf:

```
void printCC(const ComicCharacter* p)
{
  p->ComicCharacter::print();  // Aufruf von
}                              // ComicCharacter::print
```

Wichtig ist zu wissen, daß innerhalb von Konstruktoren und Destruktoren *alle* Methodenaufrufe statisch aufgelöst werden. Der Grund dafür ist, daß in Konstruktoren (beziehungsweise Destruktoren) Klassen noch nicht (beziehungsweise nicht mehr) vollständig definiert sind.

```
class A
{
public:
  A() { foo(); };
  virtual A* foo() { cout << "<A>"; };
};

class B : public A
{
public:
  B() { foo(); };
  virtual B* foo() { cout << "<B>"; };
};

B  b;
```

Im Beispiel oben wird `<A>  <B>` ausgegeben, das heißt der Konstruktor der Klasse `A` ruft `A::foo` auf, der von `B` ruft `B::foo`. Beide Aufrufe werden zum Übersetzungszeitpunkt gebunden.

Vorige Seite: 14.2.1 Deklaration von virtuellen **Eine Ebene höher:** 14.2 Virtuelle Element-Funktionen
Nächste Seite: 14.2.3 Virtuelle Destruktoren

Vorige Seite: 14.2.2 Aufruf von virtuellen **Eine Ebene höher:** 14.2 Virtuelle Element-Funktionen
Nächste Seite: 14.2.4 Überschreiben von nicht

14.2.3 Virtuelle Destruktoren

Bereits in Kapitel 11 wurden alle Destruktoren als `virtual` vereinbart. Der Grund dafür soll nun anhand eines Beispiels erläutert werden.

In Programm 14.1 wird eine Reihe von `ComicCharacter`-Objekten in einer Liste verwaltet. Wird diese Liste zerstört, so müssen auch alle enthaltenen Objekte zerstört (= deallokiert) werden. Dabei werden die Destruktoren der einzelnen Listenelemente und in weiterer Folge auch die Destruktoren der verwalteten Objekte aufgerufen.

Sind die Destruktoren der `ComicCharacter`-Klassenhierarchie nicht als `virtual` deklariert, so wird immer der Destruktor der Klasse `ComicCharacter` aufgerufen. Die korrekte De-Initialisierung ist damit nicht gewährleistet.

Nur wenn die einzelnen Destruktoren virtuell sind, wird beim Zerstören der Objekte der tatsächliche Typ des jeweiligen Objekts in Betracht gezogen und der ,,richtige`` Destruktor aktiviert.

- In allen polymorphen Klassen sind die Destruktoren als virtuell zu vereinbaren.

Vorige Seite: 14.2.3 Virtuelle Destruktoren **Eine Ebene höher:** 14.2 Virtuelle Element-Funktionen
Nächste Seite: 14.2.5 *Is-A*-Vererbung als *Programming by Contract*

14.2.4 Überschreiben von nicht virtuellen Element-Funktionen

In Kapitel 13 wurde bereits angeführt, daß nur virtuelle Element-Funktionen überschrieben werden sollen. Syntaktisch können natürlich auch ,,normale`` Funktionen überschrieben werden. Das ist allerdings problematisch. Im folgenden Beispiel werden zwei Klassen `A` und `B` vereinbart, die beide eine Element-Funktion `foo` aufweisen. Da `B` von `A` abgeleitet ist, überschreibt `B::foo` die Methode `foo` der Basisklasse `A`.

```
class A {
public:
  void foo() { ... };
};

class B : public A {
public:
  void foo() { ... };
};
```

Für jedes `B`-Objekt sollte beim Aufruf von `foo` die Methode `B::foo` aktiviert werden. Im folgenden Beispiel ist dies allerdings nicht so:

```
B      o;
A*     pa=&o;
B*     pb=&o;

pa->foo();   // A::foo
pb->foo();   // B::foo
```

Beide Zeiger `pa` und `pb` verweisen auf dasselbe Objekt `o`. In beiden Fällen *sollte* die Methode `B::foo` aktiviert werden. Da `foo` aber nicht als virtuell vereinbart ist, werden beide Aufrufe statisch gebunden. Im Fall von `pa` heißt das, daß die Methode `A::foo` gebunden wird!

Da derartige Auswirkungen kaum mehr zu durchschauen sind, gilt allgemein:

Nicht virtuelle Methoden dürfen nicht überschrieben werden.

Vorige Seite: 14.2.4 Überschreiben von nicht **Eine Ebene höher:** 14.2 Virtuelle Element-Funktionen
Nächste Seite: 14.2.6 Repräsentation polymorpher Objekte

14.2.5 *Is-A*-Vererbung als *Programming by Contract*

Vererbung wird oft als Konzept zum Ersparen von Codieraufwand mißverstanden. Die wesentliche Eigenschaft der Vererbung ist aber die *Is-A*-Beziehung. Diese Beziehung darf bei der *Interface*-Vererbung nicht zerstört werden. Auch beim Überschreiben von Methoden muß dies beachtet werden. Ein Ansatz dafür kann mit Bertrand Meyers *Programming by Contract* umschrieben werden:

Jede Klasse stellt eine Abstraktion dar, die eine Reihe von Aufgaben übernimmt. Diese Aufgaben sind in Form von Methoden implementiert. Jede Methode erfüllt einen ,,Kontrakt``:

- Sie ,,übernimmt`` unter bestimmten Umständen eine Aufgabe. Das heißt, daß bei der Aktivierung einer Methode gewisse Bedingungen (*Pre-Conditions*) gelten müssen.

  Ein Beispiel dafür ist, daß bei der Aktivierung der Methode `pop` eines `Stack`-Objekts zumindest ein Element im Stack vorhanden sein muß.
- Die Methode führt ihre Aktionen aus. Nach der Beendigung der Methode gelten bestimmte Abschlußbedingungen (*Post-Conditions*).

  Die *Post-Condition* im Fall der Methode `pop` ist, daß das `Stack`-Objekt ein Element weniger als vorher enthält.

Der Kontrakt jeder einzelnen Methode ist also, unter bestimmten Bedingungen (*Pre-Conditions*) einen Auftrag anzunehmen, auszuführen und dann zu garantieren, daß danach andere Bedingungen (*Post-Conditions*) gelten.

Jede abgeleitete Klasse ,,steht mit ihrer Basisklasse in Konkurrenz``: Sie darf einen Kontrakt der Basisklasse (= eine Methode) nur dann abändern, wenn sie ,,bessere Bedingungen bietet``. Das heißt im Detail:

- Die Eingangsbedingungen der Methode der Basisklasse dürfen nur erweitert (= abgeschwächt), nicht aber eingeschränkt werden. Jede Einschränkung würde bedeuten, daß die Methode der abgeleiteten Klasse den Kontrakt der Basisklassen-Methode nur mehr teilweise erfüllt und damit ,,verschlechtert``.
- Die Ausgangsbedingung kann beliebig eingeschränkt (= verstärkt) werden. Ein Kontrakt ist auch dann erfüllt, wenn ,,mehr`` erfüllt wird, als ursprünglich vereinbart wurde.

Kann eine Methode der abgeleiteten Klasse den Kontrakt der Basisklassen-Methode nicht erfüllen, so ist sie *kein* Kandidat für das Überschreiben. In diesem Fall ist es besser, eine neue Methode zu vereinbaren.

Vorige Seite: 14.2.4 Überschreiben von nicht **Eine Ebene höher:** 14.2 Virtuelle Element-Funktionen
Nächste Seite: 14.2.6 Repräsentation polymorpher Objekte

Vorige Seite: 14.2.5 *Is-A*-Vererbung als *Programming by Contract* **Eine Ebene höher:** 14.2 Virtuelle Element-Funktionen

14.2.6 Repräsentation polymorpher Objekte im Speicher

Abbildung 13.3 zeigt eine schematische Darstellung eines Objekts im Speicher. Diese Darstellung stimmt aber nur, wenn keine polymorphen Element-Funktionen enthalten sind. Im Fall von polymorphen Objekten sind mehr Informationen nötig.

Für jede polymorphe Klasse legt das System eine spezielle Tabelle, die sogenannte *vtbl* (*Virtual Function Table*) an. In dieser Tabelle vermerkt das System der Reihe nach die Adressen der für eine Klasse gültigen virtuellen Element-Funktionen. Für die folgende Klasse `A` enthält die *vtbl* der Klasse die Adressen der Element-Funktionen `foo1` und `foo2`.

```
class A
{
public:
  virtual void foo1() { ... };
  virtual void foo2() { ... };
  void foo3() { ... };
};
```

Eine von `A` abgeleitete Klasse `B` enthält eine *vtbl*, in der unter anderem die Adressen aller ererbten virtuellen Element-Funktionen stehen.

```
class B : public A
{
public:
  virtual void foo1() { ... };
  virtual void foo4() { ... };
};
```

Die Adressen aller neu vereinbarten virtuellen Element-Funktionen (`foo4`) werden ebenfalls in der *vtbl* abgelegt. Die virtuelle Element-Funktion `foo1` wird in `B` überschrieben. Die Adresse von `B::foo1` ersetzt daher den entsprechenden Eintrag von `A::foo1`.

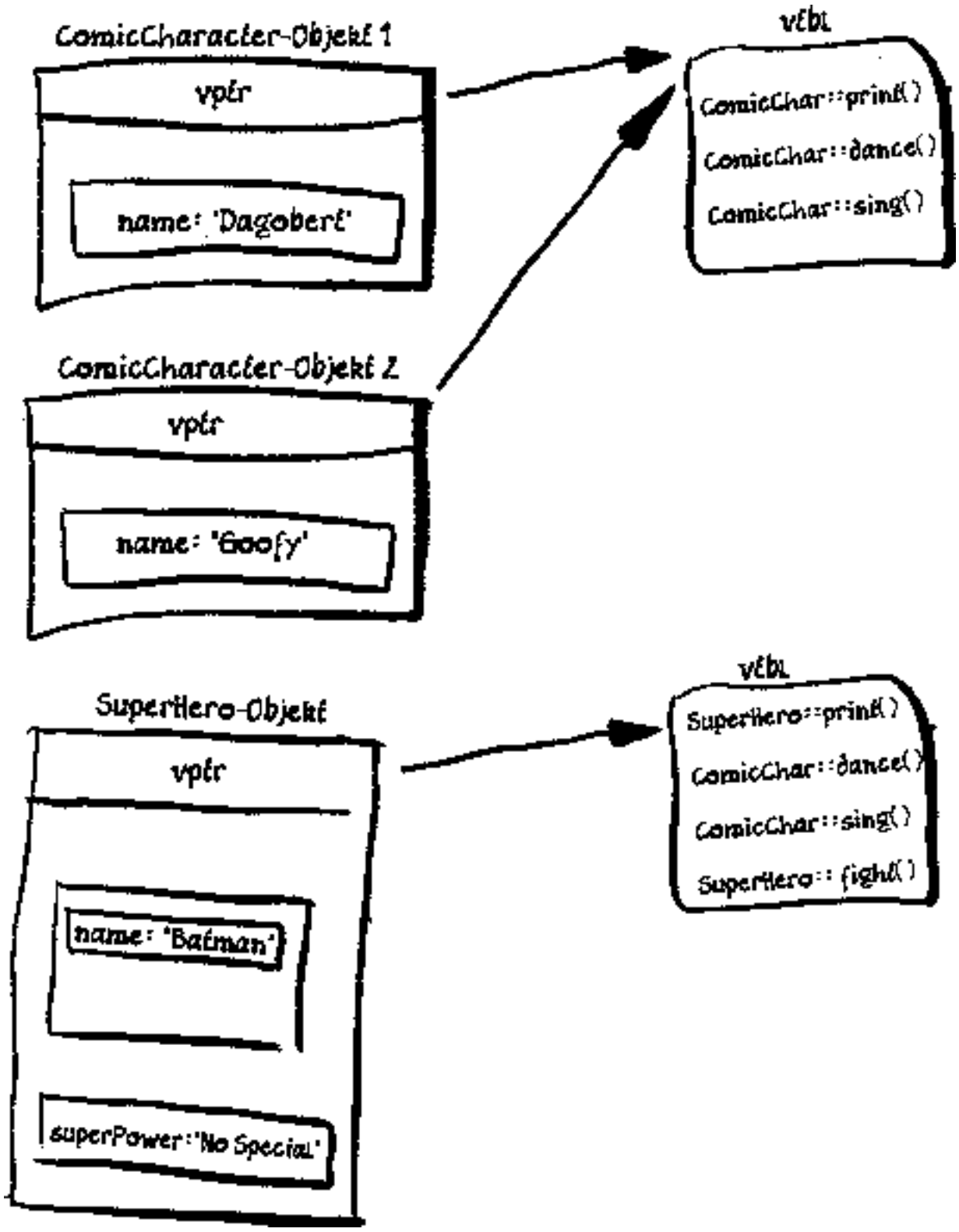


Abbildung 14.1: Repräsentation von `ComicCharacter`-Objekten mit *vtbl*

Abbildung 14.1 zeigt die Repräsentation von `ComicCharacter`-Objekten mit ihrer *vtbl* im Speicher. Die beiden ersten Objekte sind `ComicCharacter`-Objekte, die einen Verweis auf dieselbe *vtbl* haben. Die *vtbl* enthält die Adressen der Element-Funktionen `print`, `dance` und `sing`.

Das `SuperHero`-Objekt in der Abbildung weist eine andere *vtbl* auf. `SuperHero` überschreibt die

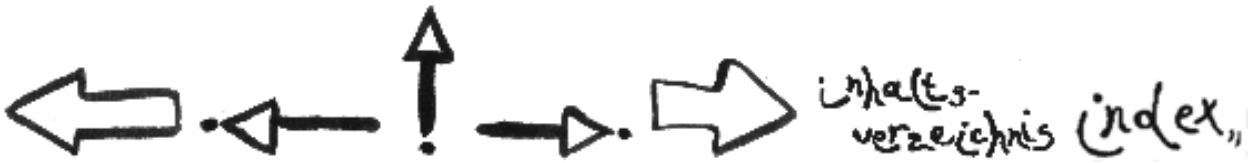
Methode `print`, daher ist in der `SuperHero`-*vtbl* der entsprechende Eintrag abgeändert (er verweist auf die `SuperHero`- und nicht mehr auf die `ComicCharacter`-Element-Funktion). Zudem weist die Klasse `SuperHero` die virtuelle Methode `fight` auf.

Der Aufrufeiner virtuellen Element-Funktion erfolgt (technisch gesehen) so:

- Der Funktionsname wird beim Compilieren in einen Index umgewandelt.
- Beim Aufruf selbst wird mit diesem Index in der *vtbl* des jeweiligen Objekts die Adresse der „richtigen`` Funktion gefunden.

Auf diese Art können Aufrufe von virtuellen Element-Funktionen (fast) so schnell durchgeführt werden wie „normale`` Funktionsaufrufe.

Vorige Seite: 14.2.5 *Is-A*-Vererbung als *Programming by Contract* **Eine Ebene höher:** 14.2 Virtuelle Element-Funktionen



Vorige Seite: 14.2.6 Repräsentation polymorpher Objekte **Eine Ebene höher:** 14 Polymorphismus und spezielle **Nächste Seite:** 14.4 Mehrfachvererbung

14.3 Abstrakte Basisklassen und rein virtuelle Element-Funktionen

Die Klasse `ComicCharacter` dient im implementierten Klassensystem als eine Art abstrakte Basisklasse. Eine abstrakte Basisklasse ist eine Klasse, die mehr oder weniger vollständig ist und dazu dient, Gemeinsamkeiten der abgeleiteten Klassen festzuhalten.

`ComicCharacter` legt fest, daß alle Comic-Figuren (also alle abgeleiteten Klassen) die Methoden `print`, `dance` und `sing` verstehen. In der ersten Version gibt `print` nur den Namen eines Objekts aus, die abgeleiteten Klassen sind dann dafür verantwortlich, genauere Informationen auszugeben. Die Methoden, die alle Klassen einer Hierarchie verstehen, werden auch als das *Protokoll* der Gruppe bezeichnet.

Definieren Methoden einer Basisklasse nur ein Protokoll, so sind sie oft nicht dazu gedacht, ausgeführt zu werden. Sie dienen dazu, eine gemeinsame Schnittstelle zu schaffen und werden rein virtuelle Methoden genannt. Die konkrete Implementierung einer rein virtuellen Methode bleibt den abgeleiteten Klassen vorbehalten. Die Methode `print` der Basisklasse `ComicCharacter` zum Beispiel ist Kandidat für eine derartige Methode, ebenso `sing` und `dance`.

Abstrakte Basisklassen sind Klassen, die ein oder mehrere rein virtuelle Methoden *(Pure Virtual Method*) enthalten. Rein virtuelle Methoden sind Methoden ohne Rumpf. Sie vereinbaren eine Schnittstelle, können aber nicht aufgerufen werden. Sie werden in C++ definiert, indem eine virtuelle Methode mit dem Wert 0 initialisiert wird:

```
virtual void print() = 0;
```

Eine Klasse mit einer oder mehreren virtuellen Methoden kann auch nicht dazu verwendet werden, Objekte zu erzeugen. Sie dient ausschließlich als abstrakte Basisklasse und zwingt den Programmierer dazu, die rein virtuellen Methoden in abgeleiteten Klassen zu implementieren.

In der Comic-Klassenhierarchie wäre es besser, `ComicCharacter` zu einer abstrakten Basisklasse zu machen und nur jene Methoden anzuführen, die alle abgeleiteten Klassen enthalten. Das ist die Methode `print`, die als rein virtuelle Methode implementiert wird. Alle abgeleiteten Klassen, die nicht als abstrakte Basisklassen dienen, müssen `print` nun selbst definieren.

Im Fall von `sing` und `dance` muß zudem angemerkt werden, daß nicht alle Comic-Figuren singen und tanzen können oder wollen. Die Angabe der Methoden `sing` und `dance` in der Basisklasse hat dazu geführt, in der Klasse `SuperHero` eine Art Fehlermeldung auszugeben. Damit kann aber ein „fehlerhafter`` Aufruf einer `sing`- oder `dance`-Methode für Superhelden erst zur Laufzeit entdeckt werden.

Von `ComicCharacter` wird daher eine Klasse `SingingComicChar` abgeleitet, die die Methoden `sing` und `dance` implementiert. `SingingComicChar` ist ebenfalls eine abstrakte Basisklasse, da `print` nicht vollständig ausgeführt ist. `sing` und `dance` sind nicht mehr in `ComicCharacter` angeführt, daher müssen `SuperHero`-Objekte nicht singen und tanzen. Alle Versuche, sie dazu zu zwingen, werden bereits vom Compiler entdeckt und verhindert (nicht erst zur Laufzeit). Das führt zu der in Abbildung 14.2 angegebenen Klassenhierarchie.

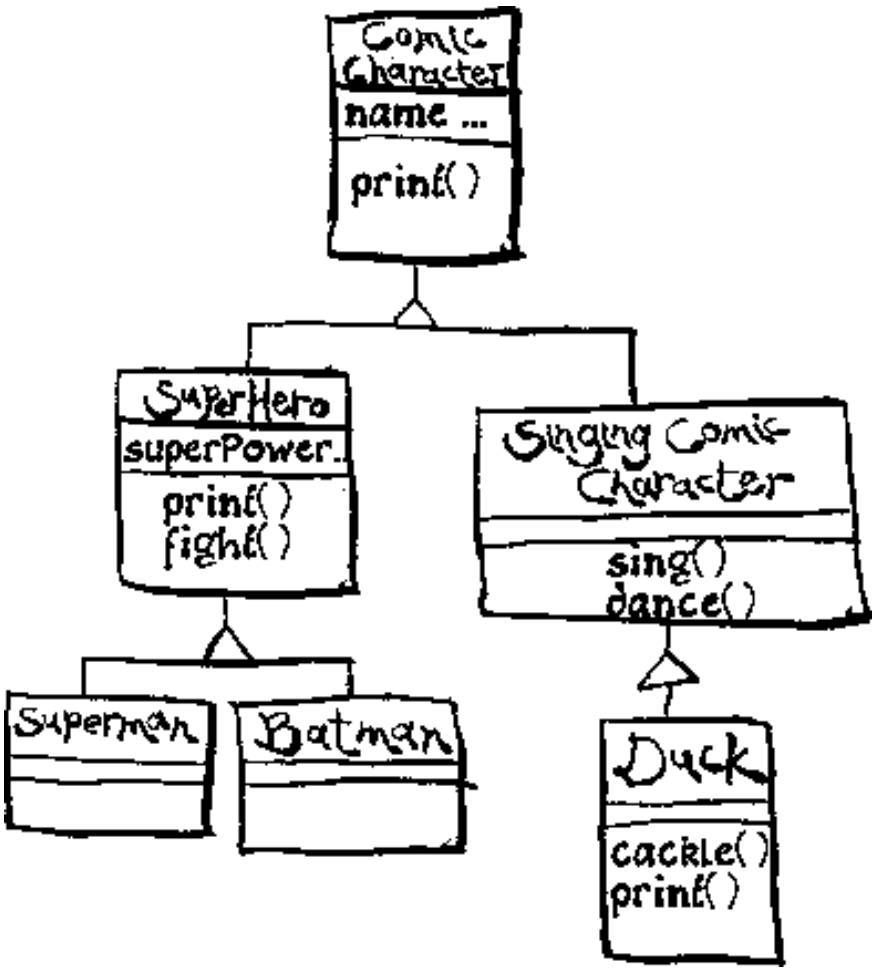


Abbildung 14.2: Klassenhierarchie `ComicCharacter` mit abstrakter Basisklasse

Die neuen Basisklassen `ComicCharacter`, `SingingComicChar` und `SuperHero` sind in Programm 14.2 angeführt. Die anderen Klassen bleiben (abgesehen von der anderen Basisklasse im Fall von `Duck`) unverändert.

Programm 14.2: `ComicCharacter`-Klassen

```
class ComicCharacter
{
 public:
  ComicCharacter() {};
  ComicCharacter(const TString& aStr)
    : name(aStr) {};
  virtual ~ComicCharacter() {};
  virtual void print() = 0;     // rein virtuell
 private:
  TString   name;
};

class SingingComicChar : public ComicCharacter
{
 public:
  SingingComicChar() {};
  SingingComicChar(const TString& aStr)
    : ComicCharacter(aStr) {};
  virtual ~SingingComicChar() {};
  virtual void dance() { cout << getName()
    << " dances... \n";
```

```
  };
  virtual void sing() { cout << getName()
    << " sings... \n";
  };
};

class SuperHero : public ComicCharacter
{
 public:
  SuperHero(const TString& aName,
            const TString& aSuperPower = "")
   : ComicCharacter(aName), superPower(aSuperPower) {};
  virtual ~SuperHero() {};
  virtual void fight() { cout << getName()
    << " fights... \n";
  };
  virtual void print() { cout << "Superhero, name: "
    << getName() << ", special abilities: "
    << superPower << "\n";
  };
 private:
  TString superPower;
};
```

`ComicCharacter` ist jetzt eine abstrakte Basisklasse. Damit können keine `ComicCharacter`-Objekte mehr angelegt werden. Die Anweisung

```
ComicCharacter    cc;
```

führt beim Übersetzen zu einem Fehler. Auch `SingingComicChar` ist eine abstrakte Klasse, da `print` nicht überschrieben wird.

Vorige Seite: 14.2.6 Repräsentation polymorpher Objekte **Eine Ebene höher:** 14 Polymorphismus und spezielle **Nächste Seite:** 14.4 Mehrfachvererbung

Vorige Seite: 14.3 Abstrakte Basisklassen und **Eine Ebene höher:** 14 Polymorphismus und spezielle
Nächste Seite: 14.5 Virtuelle Basisklassen

14.4 Mehrfachvererbung

Bei Mehrfachvererbung wird eine Klasse von mehreren Basisklassen abgeleitet. So kann zum Beispiel eine Klasse `DuckHero` definiert werden, die eine Ente mit Superkräften darstellt:

```
class DuckHero : public Duck, public SuperHero
{
public:
  DuckHero(const TString& aName, const TString& aSuperPower = "")
    : Duck(aName), SuperHero(aName, aSuperPower) {};
   virtual ~DuckHero() {};
  DuckHero(
};
```

Abbildung 14.3 zeigt die neue Vererbungshierarchie.

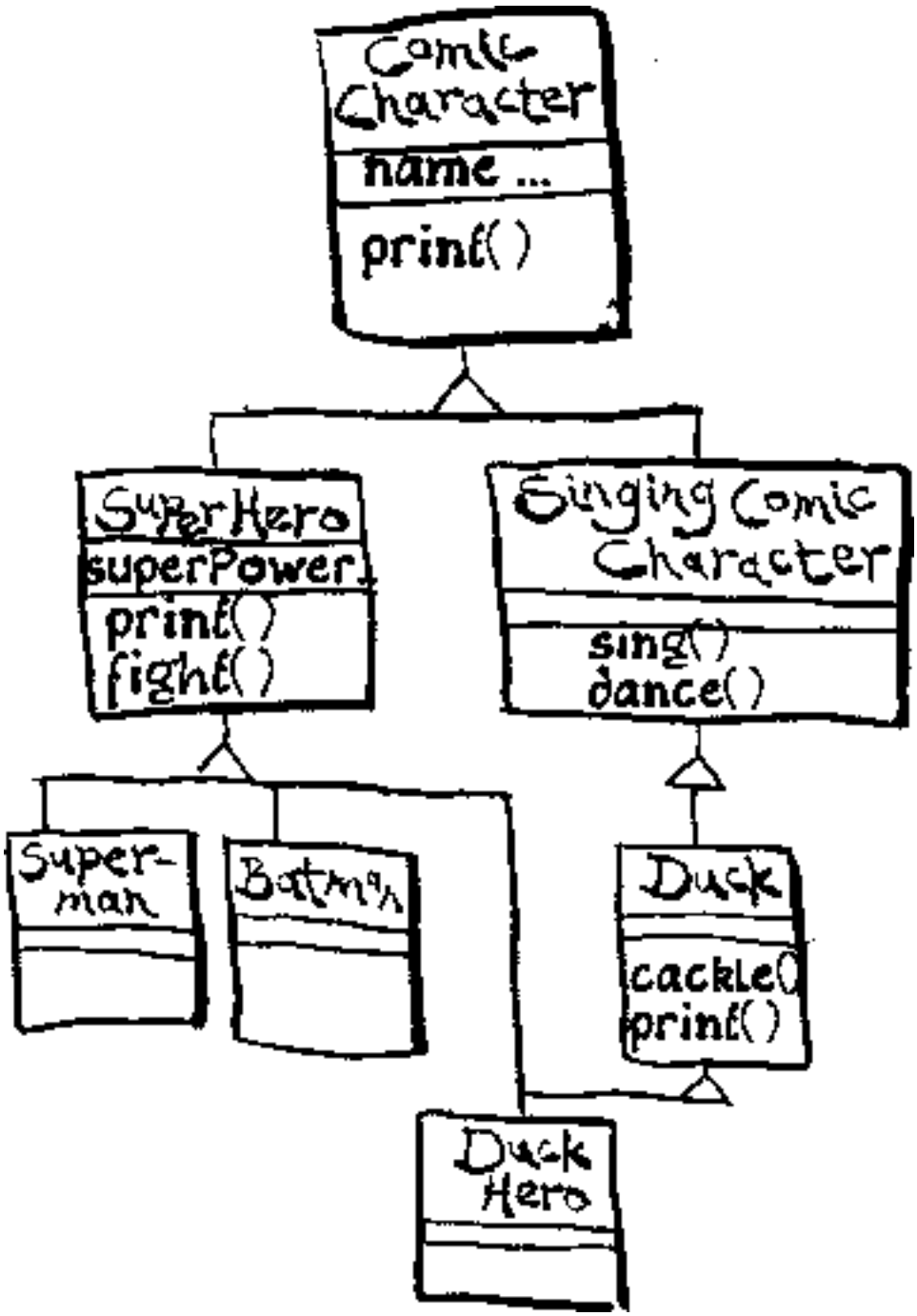


Abbildung 14.3: Vererbungshierarchie `ComicCharacter` mit Mehrfachvererbung

Die Klasse `DuckHero` erbt von `Duck` und von `SuperHero`. Sowohl die Methode `fight` als auch `cackle` werden ererbt. Ein `DuckHero` ist damit sowohl Ente als auch Superheld:

```
DuckHero*   dh = new DuckHero;

dh->fight();
dh->sing();
...
```

Mehrfachvererbung bringt aber gerade in C++ oft Probleme mit sich.

In unserem Fall tritt als erstes das Problem der Mehrdeutigkeit von Methoden auf. `DuckHero` erbt eine

Reihe von Methoden von `Duck` und von `SuperHero`. Im Fall von `print` ergeben sich daraus Mehrdeutigkeiten. Bezieht sich die Methode `DuckHero::print` auf `Duck::print` oder aber auf `SuperHero::print`? Um die Mehrdeutigkeit zu umgehen, muß `print` mit expliziter Angabe des Gültigkeitsbereichs aufgerufen werden:

```
dh->print();             // Fehler! Mehrdeutig!
dh->SuperHero::print();  // OK
```

Eine andere und meist bessere Möglichkeit ist es, in `DuckHero` eine neue Methode `print` einzuführen, die die Zweideutigkeit beseitigt:

```
virtual void print() const { Duck::print(); };
```

Ein ähnliches Problem existiert in bezug auf die Daten: `Duck` wird von `ComicCharacter` abgeleitet, enthält also einen `ComicCharacter`-Teil. Ebenso verhält es sich mit `SuperHero`. `DuckHero` ist von diesen beiden Klassen abgeleitet und enthält damit zwei `ComicCharacter`-Teile, einen im `SuperHero`-Teil und einen im `Duck`-Teil (vergleiche dazu den Code des Konstruktors in der Klasse). Diese Mehrdeutigkeit kann durch virtuelle Basisklassen (siehe Abschnitt 14.5) umgangen werden.

Ein anderes Problem ist, daß Objekt-Identitäten nicht mehr eindeutig durch die Adressen der Objekte bestimmt werden können. Alle im folgenden vereinbarten Zeiger verweisen auf dasselbe Objekt und haben aber doch verschiedene Adressen:

```
DuckHero          dh;
DuckHero*         pDH=&dh;
SuperHero*        pSH=&dh;
Duck*             pDU=&dh;
ComicCharacter*   pCC1= (Duck*)&dh;
ComicCharacter*   pCC2= (SuperHero*)&dh;
```

Mehrfachvererbung ist ein sehr mächtiges Konzept. In vielen Fällen führt Mehrfachvererbung zu wesentlich einfacherem, leichter verständlichen Code. Richtig eingesetzt bringt sie viele Vorteile mit sich. Beispiele für die erfolgreiche Verwendung von Mehrfachvererbung finden sich unter anderem in der Eiffel-Bibliothek oder in den SOM-Frameworks der IBM [IBM94].

Leider wird Mehrfachvererbung sehr oft mißbraucht und dient vorwiegend dazu, Codieraufwand zu ersparen. So ergeben sich komplexe und unübersichtliche Klassenhierarchien, die nur sehr schwer zu verstehen und anzuwenden sind.

Im Umfeld von C++ birgt die Mehrfachvererbung allerdings neben der hohen Komplexität noch andere Gefahren: die Frage der Objekt-Identität, das Problem der mehrfachen Basisklassen etc. Diese Punkte und Einwände aus dem akademischen Bereich haben dazu geführt, daß die Nachteile der Verwendung von Mehrfachvererbung oft als größer erachtet werden als die zu erwartenden Vorteile.

Hier soll dies etwas differenzierter formuliert werden: Mehrfachvererbung ist ein mächtiges und nützliches Konzept, das viele Vorteile mit sich bringt, aber auch mit Bedacht eingesetzt werden sollte. Es sollte nur in jenen Fällen verwendet werden, in denen die Vorteile offensichtlich sind und andere Lösungen umständlich scheinen. Eine derartige Anwendung ist zum Beispiel die Vererbung von zusätzlicher Funktionalität im Sinne von *Interface*-Vererbung und *Programming by Contract*.

Im folgenden Beispiel wird eine Basisklasse `MCollectible` verwendet, die Methoden zum Vergleich und zur Indizierung von Objekten zur Verfügung stellt:

```
class MCollectible
{
public:
  virtual int getHashValue() const = 0;
  virtual bool isEqual(const MCollectible& o) const = 0;
  virtual bool isSame(const MCollectible& o) const {
     return this == &o;
  };
};
```

Die Klasse dient ausschließlich dazu, Funktionalität zu vererben: Eine Klasse `CollectibleStudent` wird von `Student` und von `MCollectible` abgeleitet. Die Methoden `getHashValue` und `isEqual` werden überschrieben. Die neue Klasse ist damit ein `Student`, der zusätzlich über die `Collectible`-Funktionalität verfügt. Klassen wie `MCollectible`, die lediglich der Vererbung von zusätzlicher Funktionalität dienen, werden auch als *Mixin*-Klassen bezeichnet.

[Mey92] faßt die kontroverse Debatte um die Vor- und Nachteile von Mehrfachvererbung so zusammen:

> „Depending on who's doing the talking, multiple inheritance is either the product of divine inspiration or the manifest work of the devil.``

Vorige Seite: 14.3 Abstrakte Basisklassen und **Eine Ebene höher:** 14 Polymorphismus und spezielle
Nächste Seite: 14.5 Virtuelle Basisklassen

Vorige Seite: 14.4 Mehrfachvererbung **Eine Ebene höher:** 14 Polymorphismus und spezielle **Nächste Seite:** 14.6 Laufzeit-Typinformation

14.5 Virtuelle Basisklassen

Im Beispiel in Abschnitt 14.4 wurde `DuckHero` von `Duck` und von `SuperHero` abgeleitet. Die Klasse `DuckHero` enthält damit eine Komponente `Duck` und eine Komponente `SuperHero`. Die Klasse `Duck` wiederum enthält eine Komponente `SingingComicChar`, diese eine Komponente `ComicCharacter`, ebenso `SuperHero`. `DuckHero` enthält damit zwei Komponenten `ComicCharacter`. Beim Referenzieren eines `ComicCharacter`-Elements (beispielsweise `name`) aus der Klasse `DuckHero` muß somit der jeweilige *Scope* mit angegeben werden (`Duck::...` oder `SuperHero::...`), um die Zweideutigkeit aufzulösen.

Um die mehrfachen Basisklassen von `DuckHero` zu vermeiden, müssen die Klassen `SingingComicCharacter` und `SuperHero` virtuell abgeleitet werden:

```
class SingingComicChar : virtual public ComicCharacter
{
  ...
};

class SuperHero : virtual public ComicCharacter
{
  ...
};
```

Damit enthalten `SingingComicChar` und `SuperHero` keine direkte Komponente `ComicCharacter`, sondern nur mehr einen Verweis auf eine ,,externe`` Komponente `ComicCharacter`. `DuckHero` enthält dann nur noch *eine* Komponente `ComicCharacter` und die Zweideutigkeit ist aufgelöst. Abbildung 14.4 zeigt die schematische Darstellung der Klasse `DuckHero` bei Verwendung von virtuellen Basisklassen.

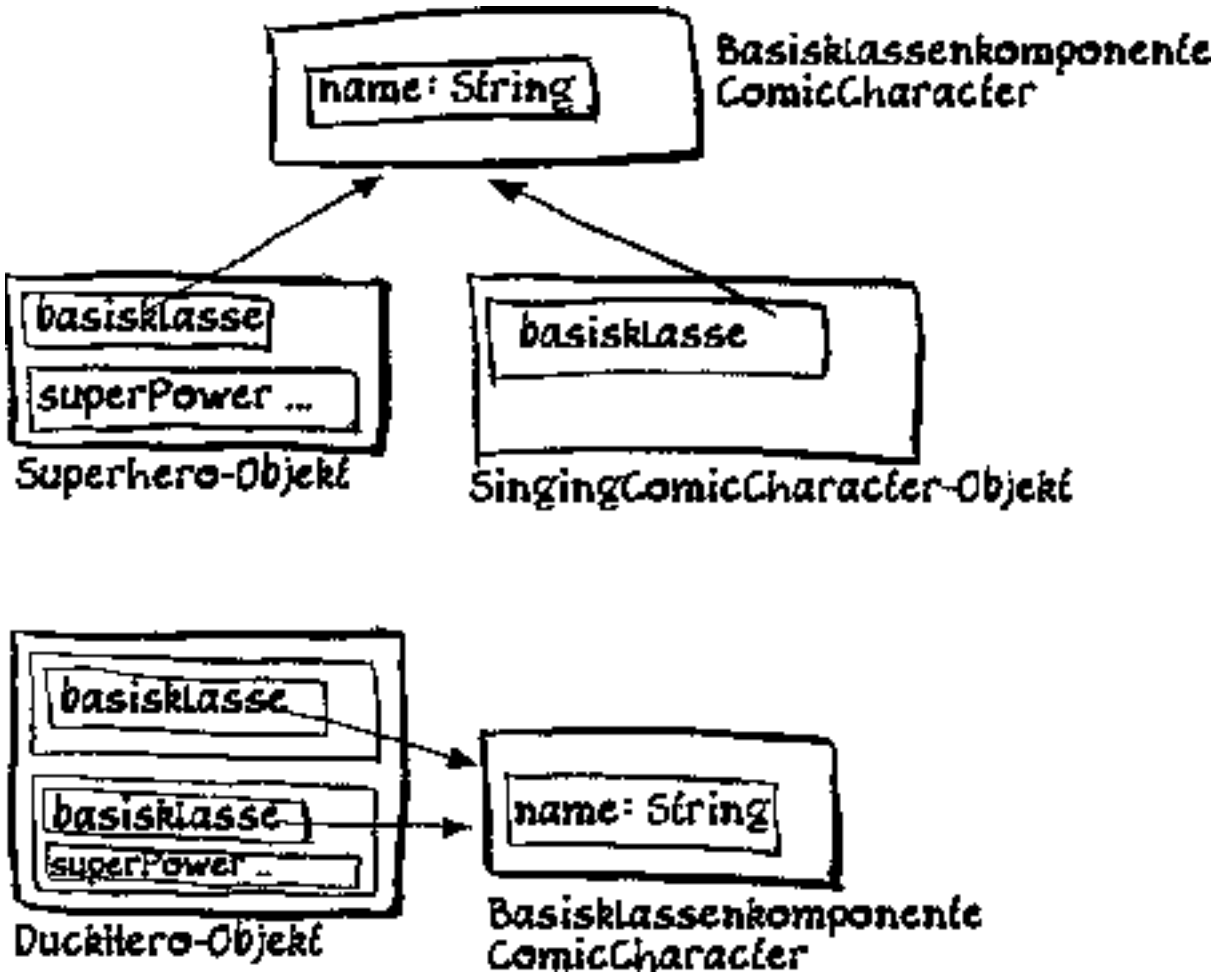


Abbildung 14.4: `DuckHero` mit virtuellen Basisklassen

Die Verwendung virtueller Basisklassen hat mehrere Konsequenzen:

- Virtuelle Basisklassen definieren eine Ordnung unter den abgeleiteten Methoden. Bei Mehrdeutigkeiten von Methoden wird - wenn sich die Methode in einer virtuellen Basisklasse befindet - jene Methode aufgerufen, die der Klasse im Ableitungsbaum am nächsten ist. Zudem werden virtuelle Basisklassen vor allen anderen konstruiert beziehungsweise zerstört und damit auch deren Konstruktoren und Destruktoren zuerst ausgeführt.

  Im Konstruktor der Klasse `SuperDuck` wird daher als erstes der Konstruktor für die Klasse `ComicCharacter` aufgerufen. Das ist nötig, weil virtuelle Klassen immer vor allen anderen initialisiert werden. Wird dieser Konstruktor nicht explizit angeführt, so fügt der Compiler den entsprechenden *Default*-Konstruktor-Aufruf automatisch ein.

- *Casts* von virtuellen Basisklassen aus sind nicht möglich (beziehungsweise nur über `dynamic_cast`, siehe Abschnitt 14.6). Das ergibt sich schon aus dem in Abbildung 14.4 angeführten Objekt-Layout: Von einer über einen Zeiger referenzierten Basisklasse kann vom System aus nicht auf die abgeleiteten Klassen geschlossen werden.

  [Str94, S. 267] führt dazu folgendes Beispiel an:

```
class A : public virtual complex { /* ... */ };
class B : public virtual complex { /* ... */ };
```

```
class C : public A, public B { /* ... */ };

void f(complex* p1, complex* p2, complex* p3)
{
  // eine Reihe von "illegalen" Casts:
  (A*)p1;  // (1)
  (A*)p2;  // (2)
  (A*)p3;  // (3)
}

void g()
{
  f(new A, new B, new C);
}
```

Die Anweisungen (1)-(3) führen beim Aufruf von `g()` zu Fehlern, da *Casts* von virtuellen Basisklassen aus nicht erlaubt sind.

Vorige Seite: 14.4 Mehrfachvererbung **Eine Ebene höher:** 14 Polymorphismus und spezielle **Nächste Seite:** 14.6 Laufzeit-Typinformation

Vorige Seite: 14.5 Virtuelle Basisklassen **Eine Ebene höher:** 14 Polymorphismus und spezielle
Nächste Seite: 14.6.1 Typumwandlung mit `dynamic_cast`

14.6 Laufzeit-Typinformation

C++ stellt über die sogenannte *Run-Time Type Information* (RTTI) Möglichkeiten zur Verfügung, den Typ eines Objekts einer polymorphen Klasse festzustellen.

Der Standard-RTTI-Mechanismus besteht im wesentlichen aus zwei Operatoren und einer Struktur:

- Der Operator `dynamic_cast` erlaubt die sichere Typumwandlung von Zeigern auf polymorphe Objekte.
- Über den Operator `typeid` kann der Typ eines polymorphen Objekts festgestellt werden.
- Eine Struktur `type_info` erlaubt die Erweiterung der bereits vorhandenen Typinformationen.

RTTI steht ausschließlich für polymorphe Klassen zur Verfügung.

Vorige Seite: 14.6 Laufzeit-Typinformation **Eine Ebene höher:** 14.6 Laufzeit-Typinformation **Nächste Seite:** 14.6.2 Der `typeid`-Operator

14.6.1 Typumwandlung mit `dynamic_cast`

Die Anweisung

```
dynamic_cast<SuperHero*>p
```

versucht, den Zeiger `p` in einen Zeiger auf ein `SuperHero`-Objekt zu konvertieren. Im Unterschied zu den bisher besprochenen *Cast*-Operatoren (Abschnitt 9.4.2) wird dabei allerdings der *tatsächliche* Typ des polymorphen Objekts in Betracht gezogen. Die Umwandlung wird nur dann durchgeführt, wenn `p` tatsächlich auf ein Objekt vom Typ `SuperHero` beziehungsweise eine davon abgeleitete Klasse verweist, andernfalls wird 0 zurückgegeben.

Der Operator kann auch für Objekt-Referenzen verwendet werden:

```
dynamic_cast<SuperHero&>(aCCRefObject)
```

Damit sind Konstruktionen der folgenden Art möglich:

```
if (SuperHero* p = dynamic_cast<SuperHero*>(q)) {
  ...  // p nur gueltig, wenn Umwandlung ok
}
       // p hier ungueltig
```

Mit dem `dynamic_cast`-Operator können auch Typumwandlungen von virtuellen Basisklassen durchgeführt werden, die ansonsten nicht erlaubt sind:

```
class A { ... };
class B { ... };
class C : public A, public virtual B { ... };
...
A* pa = new A;
B* pb = new B;

C* pc1 = (C*)pa;              // OK, keine Pruefung
                              // zur Laufzeit
C* pc2 = dynamic_cast<C*>pa;  // OK, Laufzeitpruefung
C* pc1 = (C*)pb;              // Fehler, virtuelle
                              // Basisklasse
C* pc2 = dynamic_cast<C*>pb;  // OK, Laufzeitpruefung
```

Allerdings kann der Operator (wie auch die anderen Laufzeit-Typinformationen) nur für polymorphe Klassen verwendet werden!

Vorige Seite: 14.6.1 Typumwandlung mit `dynamic_cast` **Eine Ebene höher:** 14.6 Laufzeit-Typinformation **Nächste Seite:** 14.6.3 Erweiterung und Einsatz

---

14.6.2 Der `typeid`-Operator

Der `typeid`-Operator kann auf alle polymorphen Objekte und Klassen angewandt werden und gibt eine Referenz auf ein Objekt vom Typ `type_info` zurück. `type_info` ist eine Klasse, die im Standard-Header-File `type_info.h` zur Verfügung gestellt wird. Die Klasse stellt zumindest folgende Funktionalität bereit:

- Die Operatoren `==` und `!=`,
- die Methode `before` und
- die Methode `name`.

`before` vergleicht zwei `type_info`-Objekte aufgrund einer Ordnungsrelation. Diese Relation steht aber in *keinem* Zusammenhang zu Vererbungsbeziehungen. Die Methode dient vielmehr dazu, Typinformations-Objekte zu ordnen und zu vergleichen.

`name` liefert den Namen der jeweiligen Klasse.

```
cout << "p ist ein " << typeid(*p).name() << "-Objekt";
```

Vorige Seite: 14.6.2 Der `typeid`-Operator **Eine Ebene höher:** 14.6 Laufzeit-Typinformation

14.6.3 Erweiterung und Einsatz der Typinformation

Die bisher besprochenen Typinformationen sind minimal:

- `dynamic_cast` erlaubt die Prüfung, ob Zeiger oder Referenzen von einer bestimmten Klasse oder einer davon abgeleiteten Klasse sind.
- `typeid` liefert Typinformationen, die verglichen werden können und einen Typnamen bereitstellen.

Mit diesen beiden Operatoren kann der Typ von Objekte festgestellt werden beziehungsweise können Objekte kontrolliert in Objekte eines anderen Typs (insbesonders abgeleitete Klassen) umgewandelt werden.

Die Laufzeit-Typinformation erlaubt allerdings keine Abfragen nach der Anzahl der Methoden, nach verschiedenen Basisklassen etc.

Mit den zur Verfügung gestellten Informationen können aber auf einfache Art und Weise eigene und umfassendere Typinformationen aufgebaut werden. Dazu muß eine assoziative Datenstruktur nach der Art einer `Map` bereitgestellt werden, in der die erweiterten Typinformationen zusammen mit der zugehörigen `type_info`-Struktur abgelegt werden.

Mögliche Einsatzgebiete von RTTI sind geprüfte Typumwandlungen mittels `dynamic_cast` oder etwa der Einsatz der Typinformation zur Realisierung von persistenten Objekten.

Analog [Str94] soll aber auch hier vor der exzessiven Verwendung der RTTI-Features gewarnt werden. Es ist weder guter Programmierstil noch im Sinne der objektorientierten Programmierung, Programme als eine Ansammlung von `typeid`-Abfragen und `dynamic_cast`-Umwandlungen aufzubauen. Explizite Typabfragen sind vergleichsweise selten notwendig.

Als abschreckendes Beispiel sei eine „explizite`` Version von virtuellen Element-Funktionen angeführt:

```
void foo(const CCharPtrList& theCList)
{
  CCharPtrList::Iterator  it(theCList);
  while (!it.isAtEnd()) {
```

```
    ComicCharacter*   pcc = it.element();
    if (typeid(*pcc) == typeid(Duck)) {
      // Duck, fuehre Duck-Methode aus
    } else if (typeid(*pcc) == typeid(SuperHero)) {
      // SuperHero, fuehre SuperHero-Methode aus
    }
    ...
    ++it;
   }
}
```

Diese `if`-Abfragen sind aufwendig, schwer zu warten und fehleranfällig. Jede Erweiterung der Klassenhierarchie führt zu einer Änderung des Source-Codes dieser Funktion. Hier wurde ein eklatanter Design-Fehler begangen: Derselbe Effekt kann auch durch die Wahl einer geeigneten Basisklasse und entsprechenden virtuellen Methoden erreicht werden.

Vorige Seite: 14.6.2 Der `typeid`-Operator **Eine Ebene höher:** 14.6 Laufzeit-Typinformation

Vorige Seite: 14.6.3 Erweiterung und Einsatz **Eine Ebene höher:** 14 Polymorphismus und spezielle
Nächste Seite: 14.7.1 Klassenhierarchie `Animals`

14.7 Beispiele und Übungen

Dieses Kapitel umfaßt die umfangreichsten und komplexesten Übungsbeispiele:

- `Animal` ist eine relativ kleine Hierarchie von einfachen Tierklassen.
- `CellWar` ist eine objektorientierte Variante des bekannten *Game of Life*, die den Einsatz von abstrakten Basisklassen und die daraus resultierende einfache Erweiterbarkeit von Systemen zeigt.
- `Maze` ist eine stark vereinfachte Umsetzung eines Labyrinths, das nach einem ähnlichen Prinzip wie `CellWar` arbeitet.
- `Kellerbar` ist eine Simulation der bekannten Linzer Kellerbar und realisiert eine einfache Variante einer objektorientierten diskreten Ereignissimulation.

- Aufgabenstellung

Vorige Seite: 14.7 Beispiele und Übungen **Eine Ebene höher:** 14.7 Beispiele und Übungen **Nächste Seite:** 14.7.2 `CellWar`

Teilabschnitte

14.7.1 Klassenhierarchie `Animals`

Themenbereich

Polymorphismus, Typinformation

Komplexität

Mittel

Aufgabenstellung

Zu entwerfen und zu implementieren ist eine Tier-Klassenhierarchie. Alle Klassen der Hierarchie werden (direkt oder indirekt) von der Wurzelklasse `Object` abgeleitet. `Object` weist eine Methode `debug` auf, die das Objekt (= seinen Klassennamen und seine Instanzvariablen) am Bildschirm ausgibt.

Außerdem enthält `Object` eine Methode `isKindOf`. Diese Methode hat einen Parameter `className` vom Typ `TString` und gibt an, ob das aktuelle Objekt (= der Empfänger der Nachricht `isKindOf`) von der Klasse `className` abgeleitet ist beziehungsweise eine Instanz der Klasse `className` ist.

Von `Object` wird `Animal` abgeleitet. `Animal` ist die Basisklasse für alle Tiere und definiert folgende Eigenschaften:

- Tiere haben eine Farbe, eine Geschwindigkeit und einen Namen.
- Tiere gehören einer bestimmten Gattung an, die beim Erzeugen automatisch gesetzt wird und abgefragt werden kann.
- Mit einer Methode `print` können Name und Gattung eines Tiers am Bildschirm ausgegeben werden.

Von `Animal` werden einige Tiere abgeleitet: `Mammal` (Säugetier), `Bird` (Vogel) und `Parrot` (Papagei). `Mammal` stellt die Basisklasse für spätere Erweiterungen dar, hat aber keine speziellen Eigenschaften. Ein `Bird` ist ein Tier, das fliegen kann. Ein `Parrot` ist ein Vogel, der über einen beliebig großen Sprachschatz (= Wörter) verfügt und sprechen kann (Methode `speak`). Beim Sprechen wird aus seinem Sprachschatz zufällig ein Wort ausgewählt und ausgegeben. Abbildung 14.5 zeigt die Klassenhierarchie.

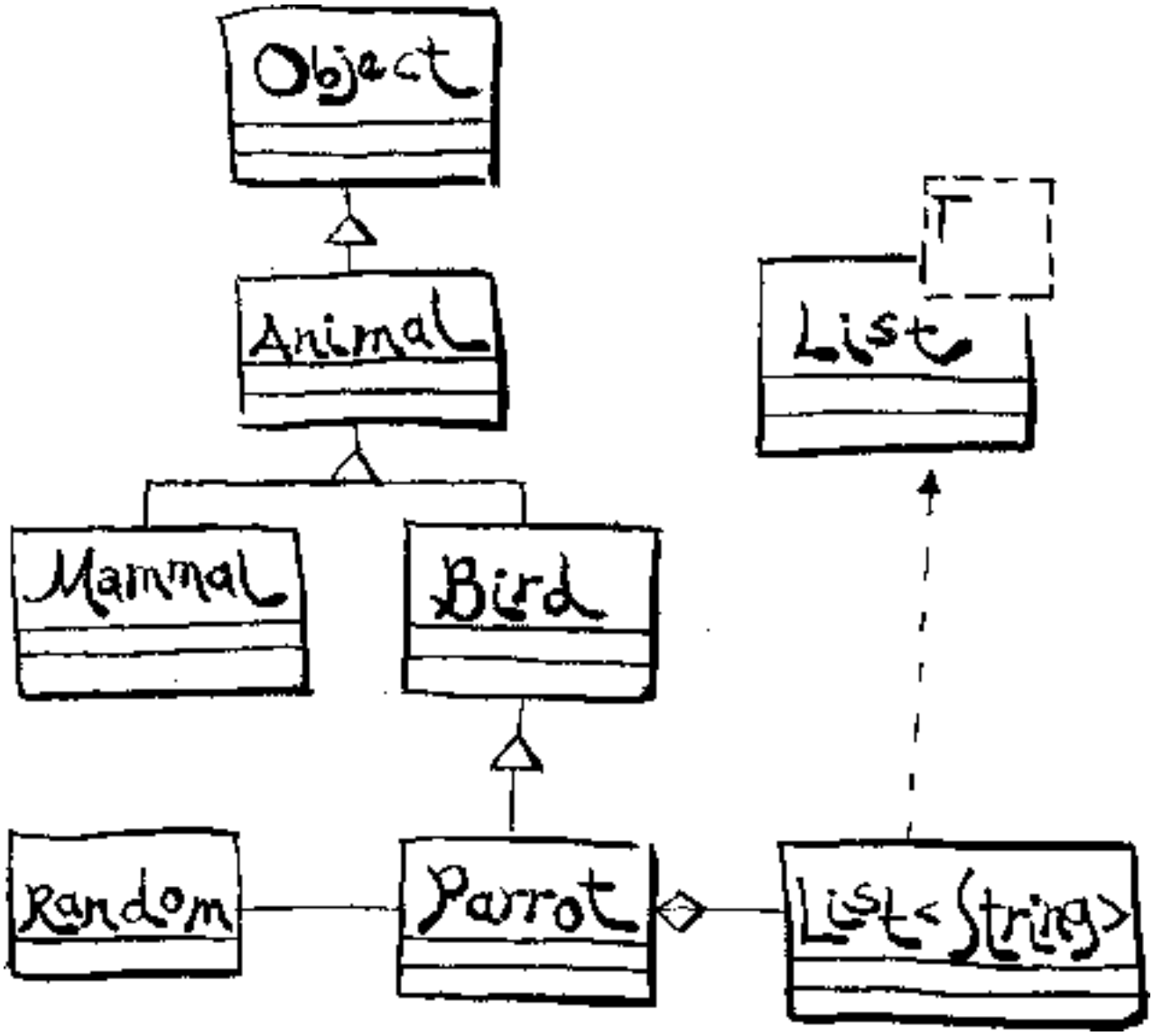


Abbildung 14.5: Klassenhierarchie `Animal`

Alle Klassen der Hierarchie müssen die Methoden `debug` und `isKindOf` „richtig`` implementieren, das heißt, die Methoden müssen die korrekten Resultate liefern.

Beispiel:

Ein `Mammal` ist ein `Mammal`, ein `Animal` und ein `Object`, aber kein `Bird` oder `Parrot`. Ein `Object` wiederum ist „nur`` ein `Object`.

Vorige Seite: 14.7 Beispiele und Übungen **Eine Ebene höher:** 14.7 Beispiele und Übungen **Nächste Seite:** 14.7.2 `CellWar`

Vorige Seite: 14.7.1 Klassenhierarchie `Animals` **Eine Ebene höher:** 14.7 Beispiele und Übungen
Nächste Seite: 14.7.3 Klassenhierarchie `Maze`

Teilabschnitte

14.7.2 `CellWar`

Themenbereich

Polymorphismus

Komplexität

Hoch

Aufgabenstellung

Eine objektorientierte Variante der bekannten Simulation *Game of Life* soll erstellt werden. Jede Zelle (`Cell`) ist dabei ein Objekt und verfügt über ein eigenes Verhalten. Zellen kennen ihre vier unmittelbaren Nachbarn sowie weitere vier, die über die unmittelbaren Nachbarn erreicht werden können (siehe Abbildung 14.6). Sie weisen Methoden zum Setzen und Abfragen der Nachbarzellen auf, können wachsen, haben ein spezielles Verhalten etc.

An Daten weist jede Zelle zumindest ihre Stärke und ihr Aussehen auf. Die Stärke einer Zelle ist maßgeblich bei der Zellteilung, beim Kampf gegen eine andere Zelle und so weiter.

Zellen wachsen mit jedem Simulationszyklus (ihre Stärke steigt) und teilen sich ab einer gewissen Größe (= Stärke). Die neu entstandenen Zellen erhalten einen Teil der Stärke ihrer Mutterzelle (die Stärke dieser wird um den entsprechenden Faktor verringert) und müssen nun eine Nachbarzelle angreifen, um Lebensraum zu erobern. Um eine bestehende Zelle besiegen zu können, muß die angreifende Zelle eine höhere Kampfkraft aufweisen als die angegriffene Zelle. In diesem Fall wird die angegriffene Zelle durch die neue ersetzt und zerstört, andernfalls stirbt der Angreifer.

Verschiedene Zellenarten haben verschiedene Überlebensstrategien und können beispielsweise die jeweils schwächsten Nachbarn angreifen, sich möglichst früh oder spät teilen oder andere Strategien verfolgen.

Der Angriff einer Zelle wird in einer Funktion `attack` implementiert und läuft wie folgt ab: Die Funktion bestimmt die Kampfstärke der angreifenden und der verteidigenden Zelle, die sich aus deren Stärke und einem jeweils zufälligem Wert zwischen 0 und *x* zusammensetzt. Ist die Kampfstärke des Angreifers höher, so ersetzt die Funktion die angegriffene Zelle durch den Angreifer (`replace`) und zerstört die angegriffene Zelle durch Aufruf des `delete`-Operators. Andernfalls ist der Angriff fehlgeschlagen und der Angreifer wird zerstört.

Ihre Aufgabe ist es, eine derartige abstrakte Klasse `Cell` zu spezifizieren und zu implementieren. Leiten Sie von dieser Klasse eine Klasse `NullCell` ab, die über keine speziellen Eigenschaften verfügt, aber eine konstante Stärke von 1 besitzt. Leiten Sie zumindest drei weitere `Cell`-Klassen ab, die über spezielle Eigenschaften und gegebenenfalls ein spezielles Verhalten verfügen. Äußere Einflüsse sollen in Form des Zufallszahlengenerators berücksichtigt werden.

Abbildung 14.6 zeigt, wie einzelne `CellWar`-Objekte (beziehungsweise Objekte von abgeleiteten Klassen) eine zweidimensionale Struktur bilden.

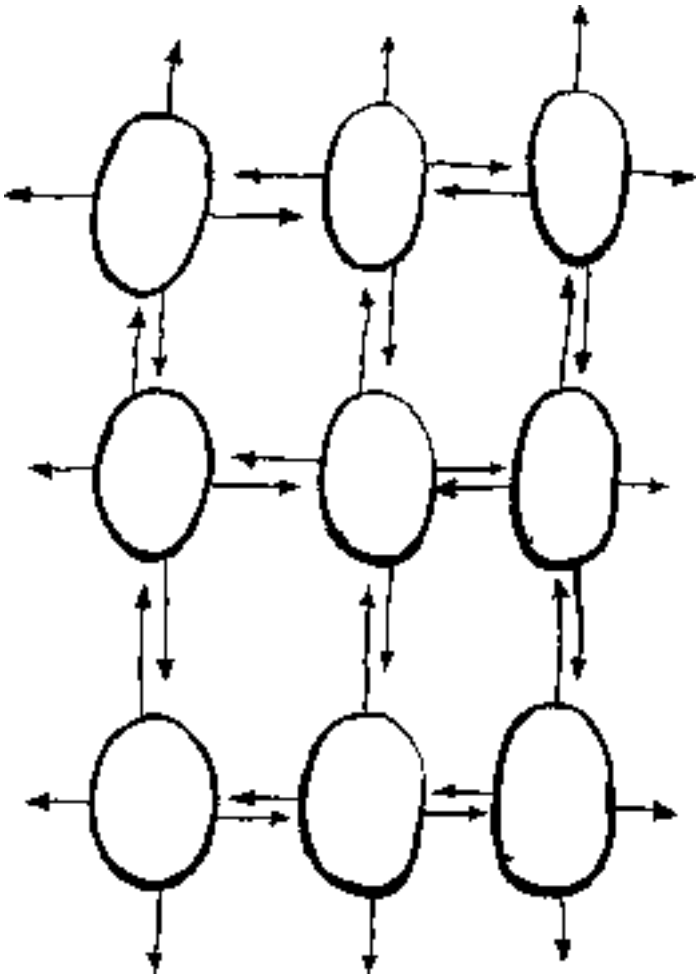


Abbildung 14.6: Simulation mit `CellWar`-Objekten

Die Schnittstelle der Funktion `attack` sieht wie folgt aus:

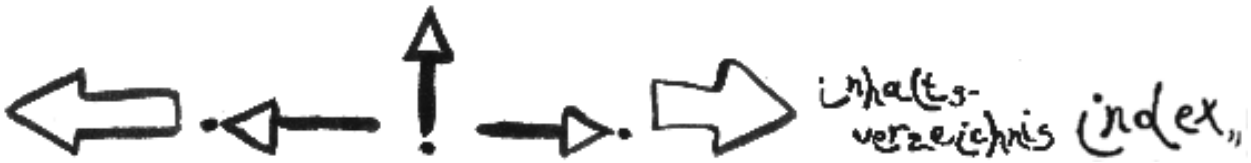
```
int attack(Cell* agressor, Cell* defender);
```

Erstellen Sie zudem eine Klasse `CellSimulation`, die zumindest folgende Funktionalität bietet:

- Anlegen eines Spielfelds
  - legt ein Feld von einer gegebenen Ausdehnung an. Die einzelnen Zellenelemente sind durch Objekte der Klasse `NullCell` gegeben. Der globale Zusammenhang der einzelnen Zellen ergibt sich durch die Verkettung der einzelnen Elemente untereinander.
- Hinzufügen einer Zelle
  - ersetzt die Zelle an einer bestimmten Stelle (gegeben durch x, y) durch eine neue Zelle. Damit können spezielle Zellen in das Spielfeld eingefügt werden.
- Ausgeben des Spielfelds
  - gibt das Spielfeld aus, indem die jeweiligen Zeichen der Zellen ausgegeben werden.
- Durchführen eines Simulationszyklus

Implementieren Sie die Klassen und führen Sie einige Simulationen durch.

Vorige Seite: 14.7.1 Klassenhierarchie `Animals` **Eine Ebene höher:** 14.7 Beispiele und Übungen
Nächste Seite: 14.7.3 Klassenhierarchie `Maze`



Vorige Seite: 14.7.2 `CellWar` **Eine Ebene höher:** 14.7 Beispiele und Übungen **Nächste Seite:** 14.7.4 Klassenhierarchie `Kellerbar`

Teilabschnitte

14.7.3 Klassenhierarchie `Maze`

Themenbereich

Polymorphismus

Komplexität

Mittel

Aufgabenstellung

Zu erstellen ist eine Klassenbibliothek `Maze`. Sie umfaßt eine Reihe von Klassen zur Verwaltung und Erzeugung von zweidimensionalen Labyrinthen. Ein Labyrinth besteht aus einer Ansammlung von Räumen. Jeder Raum hat genau vier Seiten. Jede dieser Seiten ist entweder eine Mauer oder aber eine Tür. Türen verbinden jeweils zwei Räume miteinander.

Die Topologie der Klassenbibliothek ist in Abbildung 14.7 im Überblick angeführt.

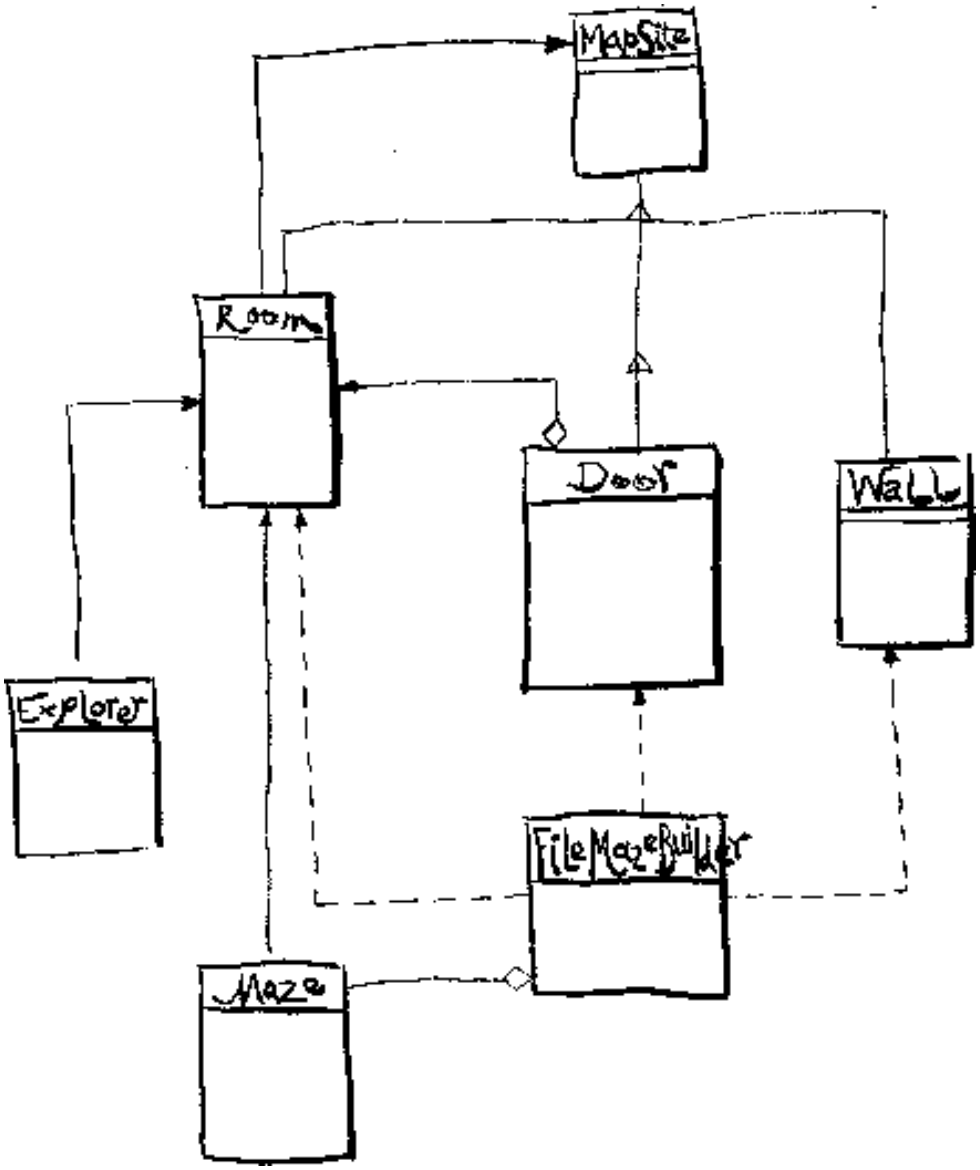


Abbildung 14.7: Klassentopologie `Maze`

Eine kurze Beschreibung der einzelnen Klassen:

- Die Klasse `MapSite` stellt die abstrakte Basisklasse dar. Alle `Maze`-Klassen werden von ihr abgeleitet. Als Methoden werden `enter` und `operate` bereitgestellt.

  `operate` führt eine für das aktuelle Objekt spezielle Aktion aus: Türen beispielweise können geöffnet oder geschlossen werden, Mauern könnten eingeschlagen werden etc.

  `enter` versucht, das aktuelle Objekt zu ,,betreten`` und liefert im Rückgabewert, ob der Versuch erfolgreich war.
- `Room` repräsentiert einen Raum im Labyrinth. Die vier Seiten werden durch vier Referenzen auf `MapSites` dargestellt. Jede dieser Referenzen kann auf ein beliebiges `MapSite`-Objekt verweisen. In unserem Fall ist dies entweder eine Mauer oder eine Tür.
- Eine Mauer (`Wall`) ist ein einfaches Objekt, das nicht betreten werden kann und auf das keine spezielle Operation definiert ist. Denkbar ist es aber, spezielle Mauern zu implementieren, die zum Beispiel mit einem Zauberspruch oder mit roher Gewalt zerstört werden können.
- Eine Tür (`Door`) ist ein Objekt, das zwei Räume miteinander verbindet. Türen können offen oder geschlossen sein. In späteren Versionen können sie auch verschlossen sein und nur mit speziellen Schlüsseln (beziehungsweise Zaubersprüchen) geöffnet werden.

Die Basisklasse `MapSite` ist in Programm 14.3 spezifiziert. Die Schnittstellen der anderen Klassen (`Room`, `Door` und `Wall`)

sind aus Platzgründen hier nicht angeführt, aber auf der CD angegeben.

Programm 14.3: `MapSite`-Schnittstelle

```
class MapSite

{
public:
  enum Direction {North, East, South, West};

  MapSite();
  virtual ~MapSite();

  virtual int enter() = 0;
  virtual int operate() = 0;

  static Direction oppositeDir(const Direction& m);
private:
  MapSite(const MapSite& o);
  MapSite& operator=(const MapSite& o);
};
```

Um ein Labyrinth auf einfache Art erschaffen zu können, ist neben den eigentlichen Labyrinth-Klassen ein `MazeBuilder` zu erstellen, der Labyrinthe aus Dateien konstruiert. Die Dateien sind wie folgt aufgebaut:

MazeFile:

OperationSeq

OperationSeq:

Operation

OperationSeq Operation

Operation:

RoomCreation

DoorCreation

EndStatement

RoomCreation:

R *IntNumber*

DoorCreation:

D *IntNumber IntNumber IntNumber IntNumber*

EndStatement:

X

MazeFile besteht aus einer (nicht leeren) Folge von *Operations*. Jede *Operation* ist entweder die Erschaffung eines neuen Raums (*RoomCreation*), einer Tür zwischen zwei Räumen (*DoorCreation*) oder das abschliessende Ende-Zeichen (*EndStatement*). Jeder Raum hat eine eindeutige Nummer, die durch die Zahl hinter `R` angegeben ist. Bei der Erschaffung einer Tür müssen vier Zahlen angegeben werden: die Nummern der beiden Räume, die durch die Tür verbunden werden, die

Richtung, in der die Tür vom ersten Raum aus gesehen wird sowie den Anfangzustand (0=geschlossen, 1=offen).

Anmerkung: Das Beispiel ist in wesentlichen Zügen [GHJV95] entnommen.

Vorige Seite: 14.7.2 `CellWar` **Eine Ebene höher:** 14.7 Beispiele und Übungen **Nächste Seite:** 14.7.4 Klassenhierarchie `Kellerbar`

Vorige Seite: 14.7.3 Klassenhierarchie `Maze` **Eine Ebene höher:** 14.7 Beispiele und Übungen

Teilabschnitte

14.7.4 Klassenhierarchie `Kellerbar`

Themenbereich

Polymorphismus

Komplexität

Hoch

Aufgabenstellung

Ihre Aufgabe besteht darin, eine Simulation einer Bar (der ,,Kellerbar``) zu erstellen. Ziel dabei ist es, herauszufinden, wieviele Barkeeper nötig sind, um die Besucher bei möglichst niedrigem Personalaufwand in akzeptabler Zeit bedienen zu können.

Folgende Fakten sind bekannt:

- Gäste betreten die Kellerbar und begeben sich (vereinfacht gesagt) sofort an die Bar, um ein Getränk zu bestellen. Sie konsumieren das Getränk und bestellen ein neues. Der Vorgang wiederholt sich, bis eine gewisse Grenze erreicht ist.
- *x* Barkeeper bedienen die Gäste. Ein Barkeeper kann zu einem Zeitpunkt genau einen Gast bedienen, das heißt sein Getränk einschenken. Dazu benötigt er eine gewisse Zeit, dann wendet er sich dem nächsten Gast zu. Die ,,Rüstzeiten``, die zwischen dem Bedienen von zwei Kunden entstehen, können vernachlässigt werden. Ist kein Gast da, der zu bedienen ist, dann widmet sich der Kellner verschiedenen anderen Tätigkeiten (Putzen).
- In der Kellerbar werden *Y* Getränke ausgeschenkt, von denen aber nur die fünf meistverkauften von Relevanz sind. Für jedes Getränk ist ein Faktor für die Simulation maßgeblich: die Zeit, die Barkeeper (im Schnitt) brauchen, um das Getränk einzuschenken.

Die verschiedenen Besucher haben (natürlich) verschiedene Trinkgewohnheiten, verschiedene Lieblingsgetränke und verweilen verschieden lang in der Kellerbar. Für die Simulation sollen in der ersten Phase fünf verschiedene ,,Typen`` von Gästen kategorisiert werden, die unterschiedliche Vorlieben an den Tag legen.

Die erste Aufgabe umfaßt die Feststellung der relevanten Daten. Die zweite Aufgabe besteht in der Erstellung von geeigneten Simulations-Klassen.

Basis für die Simulation ist das Simulation-Framework aus [GR89]. In diesem Framework besteht eine Simulation im wesentlichen aus der eigentlichen Simulation und den sogenannten Simulations-Objekten.

- Die Simulation enthält zwei Warteschlangen, in denen die Simulations-Objekte abgelegt sind, und übernimmt die eigentliche Steuerung. Zudem verwaltet die Simulation die ,,globale`` Zeit, dargestellt durch eine ganze Zahl. Für die Teilnehmer einer Simulation werden unter anderem folgende Methoden zur Verfügung gestellt:

- Abfragen der globalen Zeit
- Aufnahme eines Simulations-Objekts in die Simulation, das zu einem Zeitpunkt *t* aktiviert werden soll.
- Eine Möglichkeit, ein Simulations-Objekt für *t* Zeiteinheiten zu blockieren. Diese Möglichkeit wird zum Beispiel benutzt, um ,,Arbeiten`` von Simulations-Objekten zu realisieren. Braucht ein Kellner etwa sieben Zeiteinheiten, um ein Getränk für einen Gast zu servieren, so teilt er der Simulation mit, daß er entsprechend lange blockiert ist. Die Simulation aktiviert ihn dann nach genau sieben Zeiteinheiten erneut und er kann fortfahren.
- Eine Möglichkeit, blockierte Simulations-Objekte erneut zu ,,starten``.
- Eine Methode, die einen vollständigen Simulationszyklus durchführt. Ein Simulationszyklus ist dann beendet, wenn es keine Simulations-Objekte mehr gibt, die zum aktuellen Zeitpunkt gestartet werden müssen.

Zudem ermöglicht die Simulation eine einfache Ressourcen-Verwaltung. Ressourcen sind Simulations-Objekte, die auf ein bestimmtes Ereignis warten. Die Simulation stellt dazu unter anderem folgende Methoden zur Verfügung:

- Ressourcen können erzeugt und damit in die Warteschlange für Ressourcen eingefügt werden. Simulations-Objekte können so ,,ihre Ressource`` anderen zur Verfügung stellen und warten dann, bis ihre Ressource von einem anderen Simulations-Objekt angefordert wird.
- Resourcen können abgefragt und angefordert werden. Eine Ressource-Anforderung bewirkt, daß die Ressource aus der entsprechenden Warteschlange entfernt wird.

- Ein Simulations-Objekt ist Basisklasse für alle ,,Teilnehmer`` an der Simulation. Simulations-Objekte weisen die Methoden `startUp`, `finishUp` und `do` auf, die beim Betreten der Simulation (`startUp`), beim Verlassen derselben (`finishUp`) beziehungsweise während der Simulation selbst (`do`) aufgerufen werden.

  In der `do`-Methode wird ein einzelner Simulationszyklus abgewickelt. Ein Simulationszyklus ist eine in sich abgeschlossene Aufgabe eines Simulations-Objekts, die nicht unterbrochen werden kann. Ein Kellner zum Beispiel wird Gäste bedienen oder die Bar putzen, ein Gast wird hier sein Getränk bestellen oder trinken
- Eigene Simulations-Objekte werden von dieser Klasse abgeleitet und überschreiben die Methoden entsprechend. Dabei ist die `do`-Methode so einfach zu wählen, daß sie einen in sich abgeschlossenen Teil der Simulation umfaßt.

Kern der eigentlichen Simulation ist die Ereigniswarteschlange der Simulation, in der alle Simulations-Objekte geordnet nach dem Zeitpunkt ihrer nächsten Aktion abgelegt werden. Die Simulation wird durchgeführt, indem das erste ,,Ereignis`` (= ein Simulations-Objekt) aus der Ereigniswarteschlange entfernt wird, die Simulationszeit auf die Zeit dieses Ereignis gesetzt wird und die Methode `do` des Ereignisses (= Simulations-Objekt) aufgerufen wird.

Das Simulations-Objekt wird sich in der `do`-Methode entweder selbst zerstören (wenn seine Aufgabe beendet ist) oder aber für eine bestimmte Zeit blockieren. In diesem Fall ordnet die Simulation das Objekt wieder neu in die Ereigniswarteschlange ein (zum Zeitpunkt *jetzt+blockierZeit*).

Die Basis des Frameworks sowie einige Beispielklassen sind bereits vorhanden und unten angegeben. Ihre Aufgabe ist es, die vorhandene Basis zu adaptieren und das Simulations-Framework zu erweitern. Folgende Vorgaben sind dabei einzuhalten:

- Gäste und Kellner werden durch verschiedene Klassen repräsentiert, die von der gemeinsamen Basisklasse abgeleitet werden. Die Verwendung weiterer ,,Zwischen``-Basisklassen wie etwa ,,Anti-Alkoholiker`` ist erlaubt.
- Die grundsätzlichen Schnittstellen der Klassen sind bereits vorgegeben. Weichen Sie von den Schnittstellen nur in begründeten Fällen ab. Nicht angegeben sind die zu verwendenden Container-Klassen.
- Jeder Besucher bestimmt zufällig sein Getränk. Er kann dabei aus einer bestimmten Liste von Getränken (im Extremfall kann es auch eine Klasse ,,Nur Biertrinker`` geben) wählen. Alle Getränke sind in unbeschränkter Form vorrätig.
- Achten Sie darauf, daß in den wesentlichen Simulationsmethoden (`startUp`, `finishUp`, `do` etc.) Meldungen ausgegeben werden, anhand derer der Simulationsablauf nachvollzogen werden kann. Geben Sie Ihren Kellnern und Gästen Namen (Instanzvariablen & Methoden), die die Simulations-Objekte im Protokoll identifizieren.

Programm 14.4: Klasse `Simulation`, Schnittstelle

```
class Simulation
{
 public:
  Simulation();
```

```
 virtual ~Simulation();

   // sleepFor laesst ein SimObjekt forTheTime Zeit-
   // einheiten "schlafen"
 virtual void sleepFor(SimObj* aSimObj, int forTheTime);

   // enterNewObj bringt ein neues SimObjekt in die
   // Simulation ein
 virtual void enterNewObj(SimObj* aSimObj);

   // startUp startet die Simulation
 virtual void startUp();

   // finishUp stoppt die Simulation
 virtual void finishUp();

   // performNextCycle fuehrt einen Simulations-
   // zyklus durch:
   // * Naechstes SimObjekt aus der simQueue holen
   //     (wenn keines mehr da ist: Simulation
   //      beenden, false zurueckgeben)
   // * Solange "neue" SimZeit gleich der alten:
   //    - "do" fuer neues Objekt ausfuehren
   //    - naechstes Objekt holen
   // * Neue SimZeit setzen
   // * true zurueckgeben
 virtual bool performNextCycle();

   // getNextWaitingSimObj liefert naechstes wartendes
   // SimObjekt (0 falls keines mehr vorhanden)
 virtual SimObj* getNextWaitingSimObj();

   // getNextSimObj liefert das naechste
   // SimObjekt (0 falls keines mehr vorhanden)
 virtual SimObj* getNextSimObj();

   // addWaitingSimObj fuegt ein simObj in die
   // WarteQueue ein
 virtual void addWaitingSimObj(SimObj* s);

   // Simulationszeit:
 const int& getSimTime() const;
 void setSimTime(const int &value);

private:
   // CConstr. & = sind private:
 Simulation(const Simulation &right);
 const Simulation & operator=(const Simulation &right);
   // SimZeit:
 int simTime;
   // Collection-Klassen fuer Queues (aus STL)
 deque<SimObject*> simQueue;
 deque<SimObject*> waitingSimObjs;
};
```

Programm 14.5: Klasse `SimObject`, Schnittstelle

```
class SimObject
{
 public:
  SimObject();
  SimObject(Simulation*mySimulation);
  virtual ~SimObject();

    // enter wird aufgerufen, wenn ein SimObjekt die
    // Simulation theSimulation betritt.
    // theSimulation wird gespeichert, Init-Aktionen
    // werden ausgefuehrt. enter terminiert ueber ein
    // "sleepFor", ansonsten ist das SimObjekt nicht
    // weiter aktiv (nicht in SimQueue)
  virtual void enter(Simulation*theSimulation);

    // leave wird aufgerufen, wenn ein SimObjekt die
    // Simulation verlaesst
  virtual void leave();

    // do ist die "Hauptmethode", sie wird in jedem
    // SimZyklus des SimObjekts aufgerufen. Das
    // Objekt kann (nach eigenen Aktionen)
    // * sich ueber sleepFor zu einem spaeteren Zeit-
    //   punkt wieder aktivieren lassen (Rueckgabe
    //   von true),
    // * oder die Simulation verlassen (Rueckgabe
    //   von false)
  virtual int do();

    // Simulation:
  Simulation * & getMySimulation();

 private:
    // CConstr. & = sind private:
  SimObject(const SimObject &right);
  const SimObject& operator=(const SimObject& right);
    // Simulation:
  Simulation *mySimulation;
};
```

Vorige Seite: 14.7.3 Klassenhierarchie `Maze` **Eine Ebene höher:** 14.7 Beispiele und Übungen

Vorige Seite: Aufgabenstellung **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** 15.1 Motivation

15 Ausnahmebehandlung

Ausnahmebehandlung erlaubt es, ,,unerwartete`` Situationen in eigenen Codeabschnitten, also losgelöst vom ,,normalen`` Code, kontrolliert zu behandeln. Dieses Kapitel erklärt den zugrundeliegenden Mechanismus und gibt eine kurze Einführung in die Konzepte der Ausnahmebehandlung.

Vorige Seite: 15 Ausnahmebehandlung **Eine Ebene höher:** 15 Ausnahmebehandlung **Nächste Seite:** 15.2 Ausnahmebehandlung in C++

15.1 Motivation

Eine Ausnahme ist, vereinfacht gesagt, das Auftreten einer abnormalen Bedingung bei der Programmausführung. Im Gegensatz dazu wird mit dem Begriff ,,Fehler`` (*Error*) die Situation beschrieben, die eintritt, wenn Software ihre Spezifikation nicht erfüllt.

Tritt bei der Programmausführung eine Ausnahme auf (zum Beispiel Division durch 0, Speicher erschöpft etc.), so gibt es im allgemeinen drei Möglichkeiten, darauf zu reagieren:

- Ignorieren
  Wenn eine Ausnahme auftritt, so wird sie ignoriert. Das ist der einfachste und zum Teil auch übliche Weg (frei nach dem Motto alle nicht lösbaren Probleme zu ignorieren).
- Abbruch
  Beim Auftreten einer Ausnahme terminiert das Programm mit einer entsprechenden Meldung. Diese Möglichkeit ,,behandelt`` die Ausnahmesituation immerhin, allerdings nicht auf eine zufriedenstellende Art und Weise.
- Verwendung von Fehlercodes
  Funktionen geben durch einen Rückgabewert (oder einen Parameter) an, ob ihre Ausführung erfolgreich war. Tritt während der Ausführung eine Ausnahme auf, so wird ein entsprechender ,,Fehlercode`` gesetzt. Dies führt zu Konstrukten der folgenden Art:

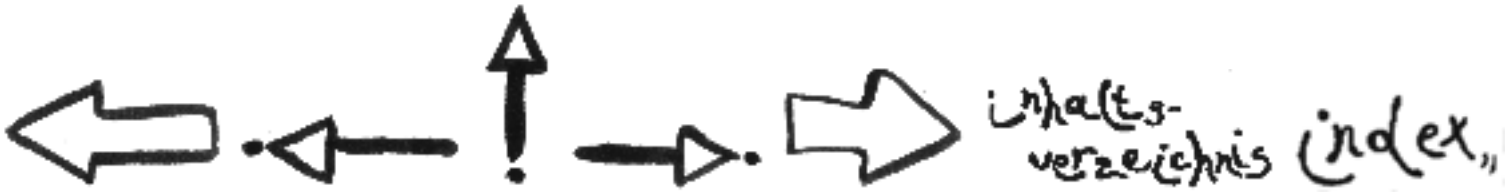
  ```
  if (!s->push(ch)) { // push nicht erfolgreich
    ...               // -> Fehlerbehandlung
  } else {
    if (!h->do(p)) {  // do nicht erfolgreich
  ...
  ```

  Eine Alternative zur Rückgabe von Statusvariablen besteht in der Implementierung eines globalen Fehlercodes. Jede Funktion, deren Ausführung fehlerhaft war, setzt einen entsprechenden Fehlercode und zeigt dem Programm damit an, daß eine Ausnahme aufgetreten ist. Aufgrund der schwereren Lesbarkeit und Wartbarkeit führt diese Möglichkeit zu kaum noch verständlichen Programmen.

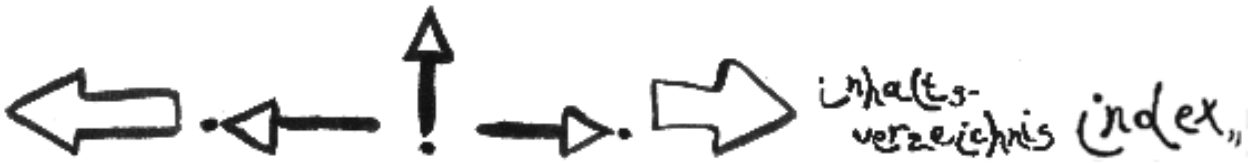
Alle drei angeführten Varianten können nicht als sinnvoll erachtet werden. Die ersten beiden Varianten behandeln die eigentliche Ausnahme überhaupt nicht, die dritte Variante tut dies nur höchst unzureichend. Problematisch an dieser Variante ist vor allem der Umstand, daß die Natur von Ausnahmen ignoriert wird: Ausnahmen sind (wie der Name sagt) nicht der Regelfall. Es scheint daher

nur wenig sinnvoll, den Code zur Behandlung von Ausnahmesituationen mit dem ,,normalen`` Programmcode zu mischen, da das Ergebnis aus einer Reihe von Selektionen besteht, in denen in jedem Fall alle möglichen Alternativen geprüft werden.

Gesucht ist daher ein Mechanismus, der der Natur von Ausnahmen näher kommt und es erlaubt, spezielle Teile des Codes für die Behandlung von Ausnahmen vorzusehen. Der ,,normale`` Code hingegen sollte von derartigem Code frei bleiben.



Vorige Seite: 15 Ausnahmebehandlung **Eine Ebene höher:** 15 Ausnahmebehandlung **Nächste Seite:** 15.2 Ausnahmebehandlung in C++



Vorige Seite: 15.1 Motivation **Eine Ebene höher:** 15 Ausnahmebehandlung **Nächste Seite:** 15.3 Auslösen von Ausnahmen

15.2 Ausnahmebehandlung in C++

Das Prinzip der Ausnahmebehandlung in C++ ist relativ einfach:

- Spezielle Konstrukte (sogenannte *Handler*) übernehmen die Behandlung der Ausnahmen. Sie sind nicht Teil des ,,normalen`` Programmablaufs sondern dienen ausschließlich dazu, Ausnahmesituationen zu kontrollieren (vergleiche dazu etwa Ausnahmebehandlung in Ada [LRM83, Coh86]).
- Tritt eine Ausnahme auf, so zeigt dies der Programmierer dem System an, indem er eine `Exception` (Ausnahme) erzeugt. Das führt dazu, daß die Ausführung des aktuellen Blocks abgebrochen wird und die Kontrolle an den ersten umgebenden *Handler* weitergereicht wird. Dieser kann die Ausnahme behandeln, die Ausführung wird dann nach dem Ausnahme-*Handler* fortgesetzt.
- Ausnahmen werden durch C++-Objekte repräsentiert. Dem Programmierer wird damit die Möglichkeit gegeben, beliebige Objekte als ,,eigene`` Ausnahmen zu vereinbaren und so Informationen (wie etwa die Art der Ausnahme oder den Ort des Auftretens) über eine Ausnahmesituation zu verwalten.

Drei Schlüsselwörter wurden eingeführt, um Ausnahmebehandlung in C++ zu ermöglichen: `try` und `catch`. `try` dient dazu, Anweisungen ,,probehalber`` auszuführen und dabei auf eventuelle Ausnahmen zu achten. Mit `catch` werden die *Handler* definiert und `throw` zeigt das Auftreten einer Ausnahme an.

Ein Beispiel nach [Mur93, S. 250] illustriert dies:

Programm 15.1: `Xcpt`-Programm

```
class Xcpt {
  // ohne Details ...
 public:
  Xcpt(const char* text);
  ~Xcpt();
  const char* getDiagStr();
 private:
  const char* diagStr;
};

  // Beispiel fuer die Ausloesung einer Ausnahme
void allocateFoo()
{
  b1;
  if (allocation() == 0) {
    throw Xcpt("Allocation failed");
  }
  b2;
}

  // Testprogramm
void testFoo()
{
  a1;
```

```
  try {
    ..
    a2;
    allocateFoo();
    a3;
    ..
  } catch (Xcpt& excCaught) {
    cout << "Caught Xcpt-exception. Diagnosis: "
         << excCaught.getDiagStr() << endl;
  }
  a4;
}
```

Die Klasse `Xcpt` stellt die „`Exception`-Klasse`` dar und enthält lediglich eine Zeichenkette für Diagnosezwecke. In der Funktion `allocateFoo` wird - falls `allocation` 0 liefert - eine Ausnahme erzeugt.

Im „Normalfall`` (keine Ausnahme) ist der Ablauf wie folgt:

1. Die Anweisungsfolge `a1` wird ausgeführt.
2. Der `try`-Block wird betreten, `a2` wird ausgeführt.
3. `allocateFoo` wird ausgeführt:
   - `b1` wird ausgeführt.
   - `allocation` liefert 1, die `if`-Anweisung wird nicht ausgeführt.
   - `b2` wird ausgeführt.
4. `a3` wird ausgeführt.
5. Der *Handler* wird nicht betreten, die Ausführung wird nach dem Handler fortgesetzt (`a4`).

Im Falle einer Ausnahme innerhalb von `testFoo` ist der Ablauf wie folgt:

1. Die Anweisungsfolge `a1` wird ausgeführt.
2. Der `try`-Block wird betreten, `a2` wird ausgeführt.
3. `allocateFoo` wird ausgeführt:
   - `b1` wird ausgeführt.
   - Eine Ausnahme tritt auf. Die Ausführung wird abgebrochen, die Kontrolle an den nächsten umgebenden *Handler* weitergereicht. `b2` wird nicht ausgeführt.
4.

Der nächste umgebende *Handler* ist in `testFoo`. Der entsprechende `catch`-*Handler* wird aufgerufen.

5. Die Ausführung wird nach dem *Handler* fortgesetzt (Anweisungsfolge `a4`), `a3` wird nicht ausgeführt.

Die einzelnen Ausnahme-Konstruktionen werden im folgenden im Detail besprochen.

Vorige Seite: 15.1 Motivation **Eine Ebene höher:** 15 Ausnahmebehandlung **Nächste Seite:** 15.3 Auslösen von Ausnahmen

Vorige Seite: 15.2 Ausnahmebehandlung in C++ **Eine Ebene höher:** 15 Ausnahmebehandlung **Nächste Seite:** 15.4 Ausnahme-*Handler*

15.3 Auslösen von Ausnahmen und Ausnahme-Objekte

Wird eine Ausnahme mit der `throw`-Anweisung ausgelöst, so wird die Programmausführung abgebrochen und nach dem unten beschriebenen Schema beim ersten ,,passenden`` umgebenden Ausnahme-*Handler* fortgesetzt (siehe Abschnitt 15.4).

Wesentlich ist dabei, daß die `throw`-Anweisung nicht einem einfachen Sprung wie etwa `goto` entspricht. Im Gegensatz zu einer Sprunganweisung werden nämlich beim Auslösen einer Ausnahme alle lokalen Objekte wieder zerstört (*Stack-Unwinding*). Nur so ist garantiert, daß das Auslösen einer Ausnahme nicht zu ,,Speicherleichen`` führt.

Die `throw`-Anweisung kann in zwei Formen auftreten:

- Wird hinter dem Schlüsselwort `throw` ein sogenanntes ,,Ausnahme-Objekt`` angegeben, so wird eine *Kopie* dieses Objekts an den ersten passenden Ausnahme-*Handler* weitergegeben.
- Die Anweisung `throw;` hingegen ist eine spezielle Variante von `throw` und nur innerhalb eines Exception-*Handlers* erlaubt. Sie bewirkt nicht das Auslösen einer ,,neuen`` Ausnahme, sondern reicht eine bereits bestehende Ausnahme weiter (mehr dazu in Abschnitt 15.4).

```
throw "Fehler!";   // Form 1: Argument beliebigen Typs
throw myExcObject;

throw;             // Form 2: kein Ausnahme-Objekt angegeben
```

Ausnahme-Objekte können beliebigen Typs sein. Einfache Datentypen wie `int` sind ebenso zulässig wie beliebige Klassen-Objekte. Im Falle der Verwendung von Klassen zur Beschreibung von Ausnahmen können eigene Hierarchien vereinbart werden, um Ausnahmen zu spezifizieren. In solchen Hierarchien lassen sich auch sehr gut Gruppen von Ausnahmen definieren.

Ein Beispiel dafür sind die im ANSI/ISO-Standard von C++ bereits vereinbarten Ausnahmen, die in eigenen Programmen zur Ausnahmebehandlung verwendet werden können [ISO95, lib.diagnostics]. Im Detail ist die Beschreibung den entsprechenden Bibliotheksdokumentationen zu entnehmen, hier erfolgt lediglich ein kurzer Überblick über die bereitgestellten Klassen.

Grundsätzlich wird zwischen Laufzeitfehlern und logischen Fehlern unterschieden. Logische Fehler sind

das Resultat von Fehlern im Programmablauf. Theoretisch kann das Auftreten von Fehlern dieser Art verhindert werden. Im Gegensatz dazu umschreibt der Begriff Laufzeitfehler Fehler, die nicht vorhersehbar sind und erst zur Laufzeit auftreten. Ein typischer Fehler dieser Art ist ein Überlauf bei arithmetischen Berechnungen.

`exception` ist die Basisklasse für alle Ausnahmen. Sie stellt einfache Methoden zum Vergleich und zum Kopieren von Ausnahmen zur Verfügung und nimmt zumindest eine Zeichenkette zur Fehlerbeschreibung auf. Zur Unterscheidung zwischen den beiden grundsätzlichen Arten von Ausnahmen werden die Klassen `logic_error` und `runtime_error` abgeleitet.

Konkrete `logic_error`-Klassen sind die Klassen `domain_error`, `invalid_argument`, `length_error` und `out_of_range`. `length_error` beschreibt den Fehler, der auftritt, wenn ein Objekt mit zu großer Länge oder Größe angelegt wird. Eine `out_of_range`-Ausnahme tritt dann auf, wenn Werte von Objekten außerhalb ihrer gültigen Bereiche liegen.

`runtime_error`-Unterklassen sind die Klassen `range_error` und `overflow_error`. Ein `range_error` beschreibt Bereichsfehler, `overflow_error` ist ein Ausnahme-Objekt zur Beschreibung von (numerischen) Überläufen.

Die Klasse `bad_exception` ist eine spezielle Klasse, die verwendet wird, um Verletzungen der Ausnahme-Spezifikationen zu markieren (mehr dazu in den folgenden Abschnitten).

Vorige Seite: 15.2 Ausnahmebehandlung in C++ **Eine Ebene höher:** 15 Ausnahmebehandlung **Nächste Seite:** 15.4 Ausnahme-*Handler*

Vorige Seite: 15.3 Auslösen von Ausnahmen **Eine Ebene höher:** 15 Ausnahmebehandlung **Nächste Seite:** 15.5 Spezifikation von Ausnahmen

---

15.4 Ausnahme-*Handler*

Ein Ausnahme-*Handler* besteht aus einem oder mehreren `catch`-*Handler*. Die einzelnen `catch`-*Handler* müssen sich in ihren Parametern unterscheiden und können (in etwa) mit einer Reihe von überladenen Funktionen verglichen werden. Dabei vereinbart jeder *Handler* einen eigenen Gültigkeitsbereich (Block).

Das Auflösen einer Ausnahme (= die Suche nach einem passenden `catch`-*Handler*) erfolgt nach folgendem Schema:

- Der nächste umgebende Ausnahme-*Handler* wird bestimmt. Das ist derjenige, der als erster gefunden wird, wenn man den *Call-Stack* entlang ,,rückwärts`` geht. Dieser *Call-Stack* speichert (ausgehend von `main`) alle Funktionsaufrufe.
- Von oben nach unten werden die einzelnen `catch`-*Handler* des Ausnahme-*Handler* daraufhin durchsucht, ob ein *Handler* ,,paßt``. Ein *Handler* ,,paßt``, wenn
  - die aktuelle `Exception` und der im *Handler* angegebene Parameter vom gleichen Typ sind,
  - der angegebene Parameter eine direkte oder indirekte Basisklasse der aktuellen `Exception` ist, oder
  - der angegebene Parameter ein Pointertyp ist und die aktuelle `Exception` ebenfalls von einem Pointertyp ist, der durch eine Standardkonvertierung umgewandelt werden kann.
- Paßt kein `catch`-*Handler*, so wird der nächste umgebende Ausnahme-*Handler* bestimmt. Existiert kein umgebender Ausnahme-*Handler* mehr, so wird die Funktion `terminate` aufgerufen. Diese kann vom Benutzer festgelegt werden.

Eine besondere Rolle spielt ein Ausnahme-*Handler* mit dem Parameter `...`. Die Ellipse `...` steht für alle möglichen Ausnahmen und paßt auf jeden Typ.

Eine weitere Besonderheit stellt die bereits erwähnte Anweisung `throw;` dar, die nur in einem Ausnahme-*Handler* auftreten darf (und nur dort sinnvoll ist). Sie gibt die aktuelle `Exception` weiter und erlaubt es damit, eine Ausnahme lokal zu behandeln und anschließend dieselbe Ausnahme an das umgebende Programm weiterzureichen. Man spricht in diesem Zusammenhang von *Exception Propagation*.

Vorige Seite: 15.3 Auslösen von Ausnahmen **Eine Ebene höher:** 15 Ausnahmebehandlung **Nächste Seite:** 15.5 Spezifikation von Ausnahmen

Vorige Seite: 15.4 Ausnahme-*Handler* **Eine Ebene höher:** 15 Ausnahmebehandlung **Nächste Seite:** 15.6 Unerwartete und nicht

15.5 Spezifikation von Ausnahmen

Für jede Funktion kann festgelegt werden, welche Ausnahmen in der Funktion (und in Funktionen, die direkt oder indirekt von der Funktion aufgerufen werden) ausgelöst werden. Nur so ist es möglich, eine Funktion auch unter Berücksichtigung von Ausnahmen weiter als *Black Box* zu betrachten, von der Klienten nur die Schnittstelle kennen müssen.

In den Schnittstellen von Funktionen und Element-Funktionen kann angegeben werden, welche Ausnahmen bei ihrer Ausführung auftreten können. Damit kann ohne Untersuchung des Source-Codes entschieden werden, welche Ausnahmen beim Funktionsaufruf auftreten *können*. Nur so kann ein Programmierer gezielt entsprechende Ausnahme-*Handler* vereinbaren.

Die Ausnahme-Spezifikation erfolgt durch das Anfügen von `throw()` an die Funktionsdeklaration. In den runden Klammern werden alle Ausnahmen angeführt, die innerhalb der Funktion auftreten können:

```
class Stack
{
public:
  Stack(int maxSize=10) throw(length_error);
  void push() throw(overflow_error);
  ...
};

void foo1() throw(specificError1, specificError2,
                  specificError3);
void foo2() throw();
void foo3();
```

Der Konstruktor von `Stack` und die Methode `push` dürfen beide jeweils eine Ausnahme auslösen. `foo1` kann drei Ausnahmen auslösen, die Funktion `foo2` darf *keine* Ausnahme auslösen (leere Ausnahmeliste).

Wird keine Ausnahmeliste spezifiziert, so kann eine Funktion *jede* Ausnahme auslösen (`foo3`).

Problematisch an Ausnahme-Spezifikationen ist, daß sie nicht spezifizieren, was sein soll, sondern das, was *nicht* passieren soll. Eine Ausnahme-Spezifikation gibt an, welche Fehler in einer Funktion auftreten können, wenn die Ausführung nicht klappt. Zudem sind Ausnahme-Spezifikationen vollständig in dem

Sinne, daß außer den angegebenen Ausnahmen *keine* anderen auftreten dürfen. Eine Ausnahme-Spezifikation für eine Funktion festzulegen ist daher keinesfalls als triviale Aufgabe zu betrachten.

Vorige Seite: 15.4 Ausnahme-*Handler* **Eine Ebene höher:** 15 Ausnahmebehandlung **Nächste Seite:** 15.6 Unerwartete und nicht

Vorige Seite: 15.5 Spezifikation von Ausnahmen **Eine Ebene höher:** 15 Ausnahmebehandlung **Nächste Seite:** 15.7 Ausnahmebehandlung in der

15.6 Unerwartete und nicht behandelte Ausnahmen

Löst eine Funktion oder Element-Funktion eine ,,nicht erlaubte`` Ausnahme aus, so wird die Funktion `unexpected` aufgerufen. ,,Nicht erlaubte`` Ausnahmen sind Ausnahmen, die *nicht* spezifiziert sind, wenn eine Ausnahmeliste angegeben ist.

Aufgabe von `unexpected` ist es, die unerwartete Ausnahme zu behandeln. In der Regel wird die Funktion das Programm beenden oder eine andere Ausnahme auslösen. Aktiviert `unexpected` eine ,,nicht erlaubte`` Ausnahme (= eine andere als die spezifizierten Ausnahmen), so wird die neue Ausnahme automatisch in die spezielle Ausnahme `bad_exception` umgewandelt. Ist `bad_exception` ebenfalls nicht in der Ausnahmeliste angegeben, so wird die Funktion `terminate` aufgerufen.

Das Verhalten im Fall von nicht erwarteten Ausnahmen kann beeinflußt werden, indem eine eigene `unexpected`-Funktion vereinbart wird. Eigene `unexpected`-Funktionen müssen folgende Schnittstelle aufweisen:

```
  // Typ einer unexpected-Funktion
typedef void (*unexpected_handler)();

  // eigene unexpected-Funktion
void myUnexpected() { ... };
```

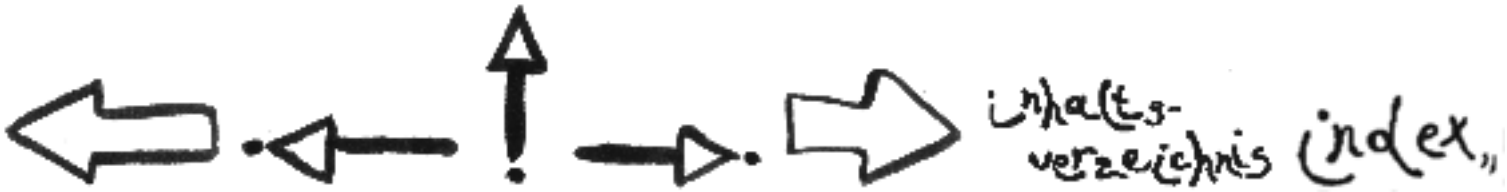
Dem C++-System wird die Adresse der eigenen Funktion durch einen Aufruf der Funktion `set_unexpected` mitgeteilt:

```
set_unexpected(myUnexpected);
```

Wird ein eigener `unexpected`-Handler vereinbart, so sollte `bad_exception` in die Ausnahmeliste aufgenommen werden, um den oben angegeben Programmabbruch mittels `terminate` zu vermeiden.

Neben der beschriebenen `unexpected`-Funktion, existiert eine weitere Funktion, die aufgerufen wird, wenn eine Ausnahme nicht behandelt wird: die Funktion `terminate`. Im ,,Normalfall`` ruft `terminate` die Funktion `abort` auf, die den Abbruch eines Programms im Fehlerfall bewirkt. Ebenso wie im Fall von `unexpected` existiert auch hier eine Funktion zum Setzen eigener `terminate`-Funktionen. Die in diesem Zusammenhang wesentlichen Vereinbarungen sind wie folgt:

```
  // Typ einer terminate-Funktion
typedef void (*terminate_handler)();

  // Funktion zum Setzen von eigenen terminate-Handlern,
  // gibt Adresse des "alten" Handlers zurueck
terminate_handler set_terminate(terminate_handler f);
```



Vorige Seite: 15.5 Spezifikation von Ausnahmen **Eine Ebene höher:** 15 Ausnahmebehandlung **Nächste Seite:** 15.7 Ausnahmebehandlung in der

Vorige Seite: 15.6 Unerwartete und nicht **Eine Ebene höher:** 15 Ausnahmebehandlung **Nächste Seite:** 15.8 Beispiele und Übungen

15.7 Ausnahmebehandlung in der Praxis

Ausnahmebehandlung ist ein sehr mächtiges Konzept zur Behandlung von *Ausnahmen*. Wie der Name bereits signalisiert, sollte Ausnahmebehandlung aber ausschließlich zur Behandlung von Ausnahmesituationen eingesetzt werden, die Behandlung von ,,normalen`` Fehlern in eigenen *Handler* erscheint wenig zweckvoll.

Eine Ausnahme ist ein kontrollierter Sprung von der aktuellen Anweisung zum ersten passenden *Handler*. Sprunganweisungen widersprechen den Regeln der strukturierten Programmierung. Alle Gründe, die gegen die Verwendung von anderen Sprunganweisungen sprechen, können auch gegen die Verwendung von Ausnahmen vorgebracht werden. So ist der Programmablauf beim Auftreten von Ausnahmen zum Teil nur schwer zu verfolgen.

Ausnahme-*Handler* sind Codestücke, die ausschließlich zur Behandlung von Ausnahmen dienen sollten. Wird ,,normaler`` Programmcode in Ausnahme-*Handler*n notiert, so wird das gesamte Programm schwerer verständlich.

Um hier keine Mißverständnisse zu produzieren: Ausnahmebehandlung ist ein Konzept, das hervorragend zur Behandlung von Ausnahmen geeignet ist. Genau dazu sollten Ausnahmen auch verwendet werden. Problematisch sind der ,,Mißbrauch`` des Konzepts sowie ein allzu umfassender Einsatz.

Als ein Beispiel für den mißbräuchlichen Einsatz von Ausnahmen soll die Verwendung von Ausnahmen als eine Art ,,Kontrollstruktur`` angegeben werden:

```
try {
  myStack.push(i);
} catch (overflow_error) {
  cout << "Ueberlauf, Stack bereits voll!";
}
...
```

Im diesem Beispiel wird eine Ausnahme verwendet, um ,,lästige`` Abfragen der folgenden Art zu vermeiden:

```
if (! myStack.isFull()) {
  myStack.push(i);
}
```

```
...
```

Anstelle von defensivem Programmieren mit einer vorbeugenden Abfrage wurde das Auftreten einer Ausnahme bewußt in Kauf genommen. Der Ausnahme-*Handler* wird so zu einem Teil des ,,normalen`` Programmablaufs und zur Ablaufsteuerung eingesetzt. Wird Ausnahmebehandlung konsequent auf diese Weise mißbraucht, so ist das Ergebnis ein ineffizientes und kaum mehr wartbares Programm.

Auch vor einem allzu umfassenden Einsatz von Ausnahmen kann nur gewarnt werden. Es ist zum einen nur schwer möglich, alle Funktionen gegen alle möglichen Ausnahmen abzusichern, zum anderen werden dadurch Programme weder einfacher noch leichter verständlich oder sicherer. Wesentlich ist es vielmehr, Programme gegen die entscheidenden Ausnahmen abzusichern.

Ausnahmebehandlung sollte ausschließlich zur kontrollierten Erzeugung und Behandlung der wesentlichen Ausnahmen eingesetzt werden.

Als grobe Richtlinien für den Einsatz von Exceptions sollen die zehn Regeln von [Ree96] angegeben werden. Die einzelnen Regeln sind nur kurz kommentiert, für eine umfassendere Betrachtung emfiehlt sich das Studium entsprechender Literatur wie etwa [Car94, Mey96, Ree96].

1. *Ausnahme-Spezifikationen sind so festzulegen, daß sie beschreiben, was eine Funktion tut und nicht was eine Funktion tun sollte.*

   Ausnahme-Spezifikationen beschreiben zwar die *möglichen* Ausnahmen, sie bestimmen die Ausführung aber nicht. Eine Spezifikation `throw()` bewirkt nicht, daß keine Ausnahme auftreten kann, sondern nur, daß das Programm beim Auftreten einer Ausnahme abbricht!

   Falsche oder unzureichende Ausnahme-Spezifikationen führen dazu, daß Klienten der Funktion mögliche Fehlerfälle von vornherein nicht berücksichtigen.

2. *Ausnahme-Spezifikationen sollten nur für ,,wesentliche`` Funktionen angegeben werden.*

   Es ist weder möglich noch sinnvoll, für *alle* Funktionen Ausnahme-Spezifikationen anzugeben. Wesentlich sind die Funktionen, die das Verhalten von Klassen implementieren. Hilfsfunktionen, *Access*-Funktionen etc. sind keine Kandidaten für Ausnahme-Spezifikationen.

3. `throw()` *sollte nur in den trivialsten Fällen verwendet werden.*

   Nur sehr wenige Funktionen *können* keine Ausnahme auslösen. Eine Ausnahme-Spezifikation `throw()` fordert Klienten dazu auf, keine Vorsorge für eventuelle Ausnahmesituationen zu treffen. Dies ist nur dann sinnvoll, wenn die Funktion auch tatsächlich weder jetzt noch in einer zukünftigen Version eine Ausnahme verursachen kann.

4. `bad_exception` *sollte in jede Ausnahme-Spezifikation aufgenommen werden.*

Das Fehlen von `bad_exception` führt im Fall von unerwarteten Ausnahmen zum Programmabbruch (beziehungsweise zum Aufruf von `unexpected`). Eine kontrollierte Behandlung ist nur möglich, wenn `bad_exception` angegeben ist.

5.

Für Funktions-Templates dürfen keine Ausnahme-Spezifikationen angegeben werden.

Templates sind generische Einheiten, die mit (fast) beliebigen Typen ausgeprägt werden. Eine Spezifikation von Ausnahmen ist nur dann korrekt, wenn sie alle möglichen Ausnahmen umfaßt. Im Fall von Templates ist das kaum möglich, da die Ausnahmen, die im Template erzeugt werden unter anderem auch von den Template-Parametern abhängen. Eine fehlende Ausnahme-Spezifikation ist immer noch besser als eine fehlerhafte.

6.

Virtuelle Funktionen sollten ausschließlich allgemeine Ausnahme-Spezifikationen aufweisen.

Im Fall von virtuellen Funktionen gilt, daß abgeleitete Klassen für überschriebene Funktionen restriktivere Ausnahme-Spezifikationen angeben können, nicht aber weniger restriktive (vergleiche *Programming by Contract*, Abschnitt 14.2.5).

Folgendes Beispiel zeigt gültige und ungültige ,,Verfeinerungen`` von Ausnahme-Spezifikationen:

```
class A {
  virtual void foo() throw(exception);
};

class B : public A {
  virtual void foo() throw(logic_error);    // (1) ok
};

class C : public B {
  virtual void foo() throw(domain_error);   // (2) ok
};

class D : public B {
  virtual void foo() throw(runtime_error);  // (3) Fehler!
};

class E : public C {
  virtual void foo() throw(length_error);   // (4) Fehler!
};
```

(1) und (2) sind gültig, da die Ausnahmen `logic_error` beziehungsweise `domain_error` ,,Verfeinerungen`` von `exception` beziehungsweise `logic_error` sind. Im Fall von (3) und (4) sind die angegebenen Ausnahmen keine ,,Verfeinerungen`` und die Funktionen daher nicht gültig.

7.

`typedef`*s sollten keine Ausnahme-Spezifikationen enthalten.*

Werden Funktionstypen vereinbart, so kann dabei angegeben werden, welche Ausnahme-Spezifikation eine entsprechende Funktion *mindestens* aufweisen muß. Eine entsprechende `typedef`-Anweisung ist aber ungültig.

```
  // ok, anyFunct mit Parameter foo(int, int)
  // und ohne Ausnahmen
void anyFunct(void(*foo)(int, int) throw());

  // Fehler, Ausnahmen in typedefs sind nicht erlaubt!
typedef void(*FooType)(int, int) throw();
void anyFunct(FooType foo);
```

8.

`terminate_handler` *sollte nicht in Bibliotheken verändert werden.*

`terminate` wird aufgerufen, um ein Programm zu verlassen. Es macht daher nur wenig Sinn, eigene `terminate_handler` innerhalb einer Bibliothek, die ja keinerlei Kenntnisse über ihre späteren Klienten hat, neu zu vereinbaren.

9.

Werden `unexpected` *oder* `terminate` *in Bibliotheken verändert, so sollten sie auch wiederhergestellt werden.*

Ist es (aus welchen Gründen auch immer) doch nötig, innerhalb einer Bibliothek eine der beiden *Handler*-Funktionen neu zu setzen, so sollten beim Verlassen des entsprechenden Codestücks die alten Werte wieder hergestellt werden. Der Grund dafür ist, daß eine Bibliothek keinerlei Annahmen darüber treffen kann (darf), ob eine der beiden Funktionen nicht vom umgebenden Programm neu gesetzt wird.

10.

`unexpected_handler` *sollte entweder* `bad_exception` *oder eine andere gültige Ausnahme auslösen.*

`unexpected_handler` behandeln nicht erwartete Ausnahmen und rufen entweder `terminate` auf oder lösen eine andere Ausnahme aus. Dieses grundsätzliche Verhalten sollte auch in einem eigenen *Handler* beachtet werden.

Vorige Seite: 15.6 Unerwartete und nicht **Eine Ebene höher:** 15 Ausnahmebehandlung **Nächste Seite:** 15.8 Beispiele und Übungen

Vorige Seite: 15.7 Ausnahmebehandlung in der **Eine Ebene höher:** 15 Ausnahmebehandlung **Nächste Seite:** 15.8.1 Klasse `TString` II

15.8 Beispiele und Übungen

In diesem Kapitel sind

- die Klasse `TString` und
- die Klasse `Rational`

aus dem Kapitel 11 mit entsprechenden Ausnahmen zu versehen.

Vorige Seite: 15.8 Beispiele und Übungen **Eine Ebene höher:** 15.8 Beispiele und Übungen **Nächste Seite:** 15.8.2 Klasse `Rational` II

Teilabschnitte

15.8.1 Klasse `TString` II

Themenbereich

Ausnahmebehandlung

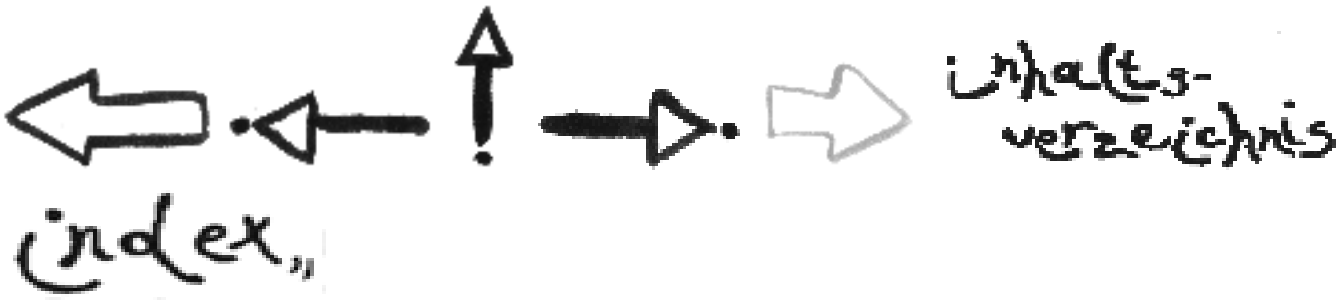
Komplexität

Mittel

Aufgabenstellung

Die in Kapitel 11 besprochene Klasse `TString` ist mit Ausnahmen zu realisieren. Als Ausnahmen stehen ausschließlich die Standard-Ausnahmen der ISO/ANSI-Klassenbibliothek zur Verfügung.

Spezifizieren Sie die jeweiligen Ausnahmen und geben Sie eine Begründung für Ihre Wahl an.



Vorige Seite: 15.8.1 Klasse `TString` II **Eine Ebene höher:** 15.8 Beispiele und Übungen

Teilabschnitte

15.8.2 Klasse `Rational` II

Themenbereich

Ausnahmebehandlung

Komplexität

Mittel

Aufgabenstellung

Erweitern Sie die Klasse `Rational` aus Abschnitt 11.9.6 um Ausnahmebehandlung. Verwenden Sie dabei (soweit dies möglich und sinnvoll ist) die Standard-Ausnahme-Klassen.

Vorige Seite: Aufgabenstellung **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** A.1 *Streams* als Abstraktion der

A Ein-/Ausgabe in C++: *Streams*

Dieses Kapitel stellt eine kurze Einführung in die Ein-/Ausgabebibliothek von C++ dar. Für eine genauere Abhandlung sei auf die entsprechende Dokumentation des Entwicklungssystems beziehungsweise auf andere Literatur (zum Beispiel [Str92]) verwiesen.

Vorige Seite: A Ein-/Ausgabe in C++: *Streams* **Eine Ebene höher:** A Ein-/Ausgabe in C++: *Streams*
Nächste Seite: A.2 Ausgabe

A.1 *Streams* als Abstraktion der Ein-/Ausgabe

Alle Objekte werden durch binäre Zeichenfolgen repräsentiert, die durch die jeweiligen Typen eine spezielle Bedeutung haben. Die Ausgabe eines Objekts kann daher auch als das Problem betrachtet werden, diese binären Zeichenfolgen in eine Folge von für Menschen ,,verständliche`` Textzeichen umzuwandeln. Diese Textzeichen können dann vom System auf dem Bildschirm angezeigt oder in einer Datei gespeichert werden. Die Eingabe von Objekten wiederum ist dann als Umwandlung von Textzeichen in die binäre Repräsentation der Objekte zu sehen.

Ein *Stream* repräsentiert einen ,,Strom`` (= sequentielle Abfolge) von Zeichen. Auf diese Ströme können Zeichen sequentiell geschrieben und von ihnen gelesen werden.

Für vordefinierte Datentypen stehen für die Ausgabe bereits entsprechende Operator-Funktionen (<<) zur Verfügung. Die Ausgabe von ,,eigenen`` Typen wird realisiert, indem ,,neue`` Operator-Funktionen << implementiert werden, die die nötige Umwandlung vornehmen.

Ähnliches trifft auf die Eingabe zu. Die Operator-Funktion >> wandelt die lesbare Repräsentation von Objekt-Werten in die entsprechende binäre Form um. Die Eingabe kann ebenso für eigene Datentypen erweitert werden, indem entsprechende Operator-Funktionen >> implementiert werden.

Standardmäßig stehen in C++-Programmen vier Ströme für die Ein-/Ausgabe zur Verfügung:

- `cin` stellt den Standard-Eingabestrom dar. Dieser Strom ist (normalerweise) mit der Tastatur ,,verbunden``, so daß jede Leseoperation gleichbedeutend mit dem Lesen einer Zeicheneingabe von der Tastatur ist.
- `cout` ist der Standard-Ausgabestrom, der üblicherweise direkt mit dem Bildschirm ,,verbunden`` ist. Jede Schreiboperation auf `cout` entspricht damit der Darstellung eines Objekts beziehungsweise Zeichens auf dem Bildschirm.
- `cerr` ist ein (ungepufferter, siehe später) Ausgabestrom wie `cout` und ist mit dem Standard-Fehler-Ausgabestrom gekoppelt. Er wird (üblicherweise) ausschließlich zur Ausgabe von Fehlermeldungen verwendet.
- `clog` ist ebenso wie `cerr` ein Ausgabestrom, der mit dem Standard-Fehler-Ausgabestrom gekoppelt ist.

Anmerkung: Die angeführten Ströme basieren auf dem Datentyp `char`. Für eine sogenannte *Wide*-Ein-/Ausgabe (basierend auf `wchar_t`) stehen die Ströme `win`, `wout`, `werr` und `wlog` zur Verfügung.

Mit Strömen können auch beliebige Dateien assoziiert und verarbeitet werden. Dabei wird jede Datei (wie im Fall der Ein-/Ausgabe) als eine Folge von Zeichen gesehen, die gelesen und beschrieben werden kann.

Vorige Seite: A Ein-/Ausgabe in C++: *Streams* **Eine Ebene höher:** A Ein-/Ausgabe in C++: *Streams*
Nächste Seite: A.2 Ausgabe

Vorige Seite: A.1 *Streams* als Abstraktion der **Eine Ebene höher:** A Ein-/Ausgabe in C++: *Streams*
Nächste Seite: A.3 Eingabe

A.2 Ausgabe

Die Klasse `ostream` stellt Methoden zur Ausgabe aller vordefinierten Datentypen (`char`, `bool`, `int` etc) zur Verfügung. Alle ,,Ausgabemethoden`` sind überladene Versionen des Operators `<<`. Die verschiedenen Versionen unterscheiden sich dabei in ihren Parametern und haben etwa folgende Schnittstelle:

```
ostream& operator<<(bool n);
ostream& operator<<(int n);
ostream& operator<<(short n);
ostream& operator<<(double n);
```

Anmerkung: Tatsächlich ist die Klasse anders realisiert [ISO95, lib.ostream]. Die tatsächliche Schnittstelle ist hier allerdings nicht von Relevanz.

Jede der Funktionen wandelt das jeweilige Objekt (angegeben durch den Parameter) in die lesbare Form um, indem die entsprechenden Werte auf den aktuellen Strom geschrieben werden. Der Rückgabewert ist der (veränderte) Strom selbst (siehe dazu auch Abschnitt 11.3).

Eine Anweisung

```
cout << "Wert von i: " << i << endl;
```

ist damit nichts anderes als die Sequenz der einzelnen Operator-Funktionen:

```
((cout.operator<<("Wert von i: ")).operator<<(i)).operator<<(endl);
```

Die Vereinbarung der Ausgabefunktion für die Klasse `Stack` aus Abschnitt 11.7.3 wird damit klarer:

```
ostream& operator<<(ostream& onTheOS,
                    const Stack& theStack)
{
  ...
}
```

Die Operator-Funktion `<<` wandelt ein `Stack`-Objekt in die ,,lesbare`` Form um, indem das Objekt Element für Element auf den Standard-Ausgabestrom geschrieben wird. Da die Schnittstelle von `Stack` ,,ungeschickt`` gewählt ist, kann von außen nicht auf alle Elemente der Datenstruktur zugegriffen werden (zum Beispiel über einen Iterator). Daher muß `<<` als `friend` von `Stack` vereinbart sein.

Dieser Umstand ist nur in der aktuellen Implementierung von `Stack` von Bedeutung. In der Regel weisen Klassen entsprechende Iteratoren beziehungsweise Zugriffsoperationen für alle Daten auf. Der jeweilige Ausgabeoperator kann dann direkt und ohne eine `friend`-Vereinbarung auf alle Elemente zugreifen.

Zeichen und Zeichenketten können auch über die Element-Funktionen `put(char c)` beziehungsweise `write(char* s, int n)` ausgegeben werden. `put` gibt ein Zeichen aus, während `write` eine Folge von `n` Zeichen ausgibt. Diese beiden Funktionen führen eine ,,unformatierte`` Ausgabe durch. Bei den `<<`-Ausgabefunktionen hingegen kann das Format festgelegt werden und man spricht von ,,formatierter`` Ausgabe (siehe Abschnitt A.4).

Wichtig ist auch zu wissen, daß die Ausgabe in der Regel ,,gepuffert`` erfolgt: Das System speichert aus Effizienzgründen *x* Zeichen zwischen und gibt sie dann ,,gemeinsam`` aus. Durch den Aufruf der Methode `flush` kann die Ausgabe aber auch erzwungen werden. Die Anweisungen

```
cout << i << flush;
cout << j; cout.flush();
```

geben die Werte von `i` und `j` unmittelbar aus. Die beiden Möglichkeiten, den Ausgabepuffer zu entleeren, sind gleichwertig.

Vorige Seite: A.1 *Streams* als Abstraktion der **Eine Ebene höher:** A Ein-/Ausgabe in C++: *Streams*
Nächste Seite: A.3 Eingabe

Vorige Seite: A.2 Ausgabe **Eine Ebene höher:** A Ein-/Ausgabe in C++: *Streams* **Nächste Seite:** A.4 Formatierte Ein-/Ausgabe

---

A.3 Eingabe

Die Eingabe ist ähnlich organisiert wie die Ausgabe. Die Klasse `istream` ist die Abstraktion eines ,,Eingabestroms`` und stellt unter anderem folgende Möglichkeiten zur Verfügung:

```
istream& operator>>(bool& n);
istream& operator>>(int& n);
istream& operator>>(short& n);
istream& operator>>(double& n);
```

Wie bereits in Abschnitt 11.7.3 im Zusammenhang mit der Ein-/Ausgabe von `Stack` dargestellt, ist die Operator-Funktion `>>` nichts anderes als die Umwandlung der Textzeichen des Standard-Eingabestroms in die binäre Repräsentation des jeweiligen `Stack`-Objekts.

Neben den Möglichkeiten der ,,formatierten`` Eingabe mit `>>` stellt `istream` unter anderem folgende Methoden zur ,,unformatierten`` Eingabe zur Verfügung [ISO95, lib.istream.unformatted]:

```
istream& get(char& c);
istream& getline(char* s,
                 streamsize n,
                 char delim=traits::newline());
char& ignore(stream_size n=1, int delim=traits::eof());
int peek();
istream& read(char* s, streamsize n);
stream& putback(char c);
stream& unget();
```

- `get` liest ein einzelnes Zeichen ein.
- `getline` liest eine Folge von maximal `n` Zeichen ein. `getline` bricht den Einlesevorgang ab, wenn
  1. das Ende des Eingabestroms erreicht wird,
  2. das Begrenzerzeichen `delim` auftritt oder aber
  3. bereits *n-1* Zeichen gespeichert sind.

- `ignore` „ignoriert`` *n* Zeichen beziehungsweise so viele Zeichen, bis das Ende des Eingabestroms oder ein `delim`-Zeichen gelesen wird. Die Zeichen werden gelesen und „weggeworfen``.
- `peek` liefert ein Zeichen, ohne es aus dem Eingabestrom zu entfernen.
- `read` liest maximal *n* Zeichen des Eingabestroms (beziehungsweise weniger, falls das Eingabeende vorher auftrat).
- `putback` gibt ein beliebiges Zeichen „zurück`` in den Eingabestrom. Die nächste Leseoperation liefert dann dieses Zeichen. `unget` führt dieselbe Aktion mit dem zuletzt gelesenen Zeichen durch.

Vorige Seite: A.2 Ausgabe **Eine Ebene höher:** A Ein-/Ausgabe in C++: *Streams* **Nächste Seite:** A.4 Formatierte Ein-/Ausgabe

Vorige Seite: A.3 Eingabe **Eine Ebene höher:** A Ein-/Ausgabe in C++: *Streams* **Nächste Seite:** A.5 *Streams* und Dateien

A.4 Formatierte Ein-/Ausgabe

Das Format der Aus- und Eingabe über die Operatoren << und >> kann, wie bereits erwähnt, festgelegt werden. Dazu stellt `ios`, eine Basisklasse von `iostream`, verschiedenste Möglichkeiten (linksbündig, rechtsbündig, Hexadezimal, Dezimal, Oktal) zur Verfügung.

Bei der Manipulation gibt es zwei grundsätzliche Möglichkeiten:

- Über das Setzen von sogenannten Formatfelder kann das Format bei der Ein- und Ausgabe festgelegt werden. Dazu wird das entsprechende *Flag* mit der Methode `setf` gesetzt. `setf` weist zwei Parameter auf. Der erste bestimmt den konkreten Wert eines *Flags*, der zweite, welches *Flag* verändert wird.
- Daneben existieren spezielle Methoden (Manipulatoren), die die entsprechenden *Flags* direkt verändern.

Im folgenden werden die ,,wichtigsten`` Manipulationsmöglichkeiten anhand von Beispielen gezeigt. Für genauere Abhandlungen sei auf das Hilfesystem beziehungsweise die Dokumentation des jeweiligen Entwicklungssystems verwiesen.

- `int`-Zahlen können in verschiedenen Formaten ausgegeben werden:

```
int    i = 512;
cout << "\nNormal: " << i;
cout << "\nOktal:  "   // oct stellt Ausgabe
     << oct << i;      // auf oktal um
cout << "\nOktal: "    // kein "oct" noetig, da
     << i;             // Einstellung erhalten bleibt
cin >> i;              // Oktale Eingabe
cout << "\nHex:    "   // Hexadezimale Ausgabe
     << hex << i;

  // Alternative Moeglichkeit: Feld direkt
  // ueber setf setzen
cout.setf(ios::dec, ios::basefield);
cout << "\nDezimal:" << i;
```

  Die Ausgabe bei der Eingabe von `15`:

```
Normal: 512
Oktal:  1000
Oktal:  1000
Hex:    f
Dezimal:15
```

- Auch das Format von Fließkommazahlen kann festgelegt werden:

```
  // Fliesskomma
cout << "Scientific: " << scientific << 123.45
     << endl;
cout << "Fixed: " << fixed << 123.45 << endl;
cout << "Precision: " << precision(2)
     << 123.45678;
```

  Die Ausgabe:

```
Scientific: 1.2345E2
Fixed: 123.45
Precision: 123.45
```

- Zudem können für alle Ausgaben Feldlängen festgelegt werden. Die Ausgabe von Objekten erfolgt dann (falls möglich) innerhalb der festgelegten Feldlänge. Mit Methoden wie `left`, `right` und `internal` wird die Ausrichtung der Ausgabe innerhalb des angegebenen Felds bestimmt, mit `fill` das „Füllzeichen``:

```
  // Feldlaenge festsetzen
cout.width(5);
cout << '(' << "ab" << ')' << endl;

  // Feldlaenge & Fuellzeichen festsetzen
cout.width(5);
cout.fill('*');
cout << '(' << "ab" << ')' << endl;

  // Ausrichtung festsetzen
cout.width(20);
cout.right();
cout << "Rechts" << endl;
```

  Die Ausgabe:

```
   ab
******ab
**************Rechts
```

Tabelle A.1 zeigt eine kurze Übersicht über die wichtigsten Formatmöglichkeiten.

Element	Wirkung wenn gesetzt
`boolalpha`	`bool`-Werte werden textuell ausgegeben.
`dec`	Ausgabe erfolgt dezimal.

`fixed`	Ausgabe der Gleitkommazahlen im Fixpunktformat.
`hex`	Ausgabe erfolgt hexadezimal.
`internal`	Ausgabe erfolgt innerhalb eines angegebenen Felds.
`left`	Ausgabe wird mit Füllzeichen (rechts) aufgefüllt.
`oct`	Ausgabe erfolgt oktal.
`right`	Ausgabe wird mit Füllzeichen (links) aufgefüllt.
`scientific`	Gleitkommaausgabe im wissenschaftlichen Format (Mantisse genormt, mit Exponent).
`showbase`	Zahlenbasis wird bei der Ausgabe angegeben.
`showpoint`	Der Dezimalpunkt wird immer ausgegeben.
`showpos`	Bei der Ausgabe von positiven Zahlen wird Vorzeichen angezeigt.
`skipws`	Führende *Whitespaces* werden übersprungen.
`unitbuf`	Leert Puffer des Ausgabestroms nach jeder Ausgabeoperation.
`uppercase`	Alle Kleinbuchstaben werden automatisch durch Großbuchstaben ersetzt.

Tabelle A.1: Format-*Flags* im Überblick [ISO95, lib.ios::fmtflags]

Vorige Seite: A.3 Eingabe **Eine Ebene höher:** A Ein-/Ausgabe in C++: *Streams* **Nächste Seite:** A.5 *Streams* und Dateien

Vorige Seite: A.4 Formatierte Ein-/Ausgabe **Eine Ebene höher:** A Ein-/Ausgabe in C++: *Streams*
Nächste Seite: A.5.1 Öffnen von Dateien

A.5 *Streams* und Dateien

Wie bereits beschrieben, sind Ströme Abstraktionen für die zeichenweise Ein-/Ausgabe. Dabei ist es unerheblich, ob die Ausgabe auf den Bildschirm erfolgt oder auf eine Datei, da jede Datei auch als eine sequentielle Folge von Zeichen aufgefaßt werden kann. Dateien können also über die ,,normalen`` Möglichkeiten der Ein-/Ausgabe beschrieben werden (`<<`, `>>`, `get`, `write` etc.). Allerdings sind dabei einige ,,Besonderheiten`` zu beachten:

- Dateien müssen geöffnet werden. Das Öffnen einer Datei assoziiert die physikalische Datei mit dem entsprechenden Strom. Existiert die Datei noch nicht, so kann sie beim Öffnen angelegt werden.

  Dateien können zum Lesen (*Read Only*), Schreiben (*Write Only*), oder zum Lesen und Schreiben (*Read-Write*) geöffnet werden.
- Dateien müssen nach ihrer Verarbeitung geschlossen werden. Jede Datei wird automatisch geschlossen, wenn der assoziierte `Stream` zerstört wird.
- Die Verarbeitung von Dateien kann im Textmodus oder im Binär-Modus erfolgen. Bei der binären Verarbeitung werden Ströme Zeichen für Zeichen ohne jegliche Transformation gelesen beziehungsweise geschrieben. Im Textmodus wird zum Beispiel das Steuerzeichen `endl` in die ,,Plattform-übliche`` Zeilen-Ende-Sequenz umgewandelt (`0x0a` `0x0d` auf DOS, OS/2 etc.).

Der Zugriff auf eine geöffnete Datei schließlich erfolgt über einen fiktiven ,,Schreib-/Lesekopf``, der die aktuelle Position in der jeweiligen Datei angibt. Diese Position kann über spezielle Methoden verändert werden.

Anmerkung: Neben den ,,normalen`` `char`-basierten Klassen (`ifstream` für Eingabeströme, `ofstream` für Ausgabeströme und `fstream` für Ein-/Ausgabeströme) existieren auch entsprechende Versionen für `wchar_t`-basierte Ströme: `wifstream`, `wofstream` und `wfstream`.

Vorige Seite: A.4 Formatierte Ein-/Ausgabe **Eine Ebene höher:** A Ein-/Ausgabe in C++: *Streams*
Nächste Seite: A.5.1 Öffnen von Dateien

Vorige Seite: A.5 *Streams* und Dateien **Eine Ebene höher:** A.5 *Streams* und Dateien **Nächste Seite:** A.5.2 Lesen und Setzen

A.5.1 Öffnen von Dateien

Das Öffnen einer Datei erfolgt beim Anlegen eines `Stream`-Objekts beziehungsweise über die Methode `open`. Dabei werden der Name der Datei und der Öffnungsmodus als Parameter übergeben. Daneben können weitere, auch implementierungsspezifische Parameter angegeben werden.

Tabelle A.2 zeigt die verschiedenen Möglichkeiten, eine Datei zu öffnen. Die Möglichkeiten können (sofern sie sich nicht widersprechen) auch kombiniert werden.

Einige Beispiele:

```
  // Default-Modi:
ifstream   readFrom("Eingabe.txt");
ofstream   writeTo("Ausgabe.txt");

  // Fuer Eingabe oeffnen
ifstream   readFrom("Eingabe.txt", ios_base::in);

  // Binaere Eingabe
ifstream   readFrom("Eingabe.txt", ios_base::in  |
                                   ios_base::bin);
  // Binaere Ausgabe und Datei leeren,
  // falls sie existiert
ifstream   readFrom("Eingabe.txt", ios_base::out  |
                                   ios_base::bin |
                                   ios_base::trunc);
```

Modus	Kommentar
`in`	Datei für Eingabe öffnen.
`out`	Datei für Ausgabe öffnen.
`app`	Schreiboperationen am Dateiende ausführen.
`ate`	Nach dem Öffnen der Datei sofort an das Dateiende verzweigen.
`trunc`	Zu öffnende Datei zerstören, wenn sie bereits existiert.

`bin[ary]`	Ein-/Ausgabe wird binär durchgeführt und nicht im Textmodus.

Tabelle A.2: Modi beim Öffnen einer Datei [ISO95, lib.ios::openmode]

Vorige Seite: A.5 *Streams* und Dateien **Eine Ebene höher:** A.5 *Streams* und Dateien **Nächste Seite:** A.5.2 Lesen und Setzen

Vorige Seite: A.5.1 Öffnen von Dateien **Eine Ebene höher:** A.5 *Streams* und Dateien

A.5.2 Lesen und Setzen von Positionen

Dateien müssen nicht streng sequentiell verarbeitet werden. Vielmehr kann die aktuelle Position innerhalb einer Datei über verschiedene Methoden abgefragt und verändert werden. Dazu stehen unter anderem folgende Typen und Methoden zur Verfügung:

- `streampos` ist der Typ einer Dateiposition.
- `seekg(offset, direction)` zum Beispiel setzt die aktuelle Dateiposition des fiktiven Lesekopfs einer Datei. `offset` gibt die Position vom Dateianfang beziehungsweise -ende aus an, `direction` legt fest, von wo aus die Position bestimmt wird: `ios::beg` (Dateianfang), `ios::cur` (aktuelle Position) oder `ios::end` (Dateiende).

  `aFile.seekg(-10, ios::end)` setzt die aktuelle Position zehn Bytes *vor* dem Dateiende, `aFile.seekg(10, ios::beg)` zehn Bytes *nach* dem Dateianfang.

  Der Suffix `g` in `seekg` steht für *Get*.
- `tellg` liefert die aktuelle Position des fiktiven Lesekopfs in der Datei.
- `seekp` und `tellp` sind die entsprechenden Versionen für die *Put*-Varianten (in bezug auf Ausgabeströme).

Das folgende Beispiel ist [Lip91, S. 358] entnommen. Es liest eine Datei und schreibt die Positionen aller Zeilenumbrüche (*Newline*) sowie die Größe der gesamten Datei an das Dateiende.

Die Datei

```
Hallo
das
ist die Test-Eingabedatei.
Zeile vier
und fuenf.
```

wird vom Programm A.1 wie folgt verändert:

```
Hallo
das
ist die Test-Eingabedatei.
Zeile vier
und fuenf.
6 10 37 48 59 [ 73 ]
```

Die ersten fünf Zahlen (6..59) geben die Positionen der *Newlines* in der Datei an, die Zahl in eckigen Klammern gibt die Gesamtgröße in Bytes an.

Programm 16.1: Dateiposition zählen [Lip91, S. 358]

```
#include <iostream>
#include <fstream.h>

void main()
{
  // Datei f. Eingabe & zum "Anhaengen" oeffnen
  fstream inout("test.txt", ios::in|ios::app);
```

```
  int cnt=0;
  char ch;

  // aktuelle Position = File-Anfang
  inout.seekg(0, ios::beg);

  while (inout.get(ch)) {
    cout.put(ch);
    cnt++;
    // Wenn Zeilenende gelesen: aktuelle Position
    // am Ende der Datei anhaengen
    if (ch=='\n') {
      streampos   mark = inout.tellg();
      inout << cnt << ' ';
      // Wieder zur "alten" Position
      inout.seekg(mark);
    }
  }
  // eof trat auf, daher erfolgen keine Ein-/Ausgaben
  // mehr Status muss "geloescht" werden -> clear
  inout.clear();
  inout << cnt << endl;

  cout << "[ " << cnt << " ]" << endl;
}
```

Einige Kommentare zum Programm:

- `inout` wird als Ein-/Ausgabedatei geöffnet. Alle Schreiboperationen erfolgen automatisch am Dateiende (Modus: `app`).
- Um die Datei von Anfang an zu verarbeiten, wird die Position des fiktiven Lesekopfs auf den Dateianfang gesetzt (`seekg(0, ios::beg`).
- Wenn ein Zeilenumbruch gelesen wurde, wird die aktuelle Position sowie ein Leerzeichen geschrieben. Die aktuelle Position in der Datei wird über die Methode `tellg` ermittelt. Um die Datei nach den Schreiboperationen (die ja am Dateiende erfolgen) weiter zu verarbeiten, wird die Position anschließend wieder hergestellt.
- Die `while`-Schleife wird verlassen, wenn kein Zeichen mehr gelesen werden kann. Da auftretende Fehler und das Überlesen des letzten Zeichens einer Datei dazu führen, daß keine weiteren Lese- und Schreiboperationen mehr stattfinden, werden die Status-*Flags* durch `clear` wieder gelöscht.

Vorige Seite: A.5.1 Öffnen von Dateien **Eine Ebene höher:** A.5 *Streams* und Dateien

Vorige Seite: A.5.2 Lesen und Setzen **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** B.1 Direktiven

---

B Präprozessor-Direktiven

Vor der eigentlichen Übersetzung eines C++-Programms wird der sogenannte Präprozessor aktiviert. Der Präprozessor führt textuelle Manipulationen von Source-Dateien durch. Dazu gehören Makro-Substitution, bedingte Übersetzung und Einfügen von Dateien. Wenn der Präprozessor fertig ist, übergibt er die *geänderte* Version der Datei an den eigentlichen Compiler.

Zeilen, die mit # beginnen, sind Direktiven für den Präprozessor. Die Syntax dieser Zeilen ist unabhängig von der Syntax der eigentlichen Programmiersprache C++. Sie dürfen überall innerhalb einer Datei stehen, ihre Auswirkungen bleiben unabhängig von Gültigkeitsbereichen bis zum Ende der Quelldatei erhalten.

Die wichtigsten Präprozessor-Anweisungen umfassen Makros, Einfügen von Dateien, bedingte Compilation von Source-Teilen und einige spezielle Direktiven [ISO95, cpp]:

```
preprocessing-file:
                group opt
group:
                group-part
                group group-part
group-part:
                pp-tokens opt new-line
                if-section
                control-line
lparen:
                left parenthesis without preceding white-space
replacement-list:
```

pp-tokens$_{opt}$

pp-tokens:

preprocessing-token

pp-tokens preprocessing-token

new-line:

the new-line character

Zudem wird die Ersetzung der sogenannten Trigraph-Sequenzen und der alternativen Operator-Symbole vom Präprozessor durchgeführt. Diese wurden aber bereits behandelt (siehe Abschnitt 3.5.3) und werden nicht mehr gesondert angeführt.

Anmerkung: Der Präprozessor wird in C++ anders als in C lediglich für einige genau definierte Bereiche eingesetzt: bedingte Übersetzung, Pragmas etc.

- B.1 Direktiven
- B.2 Bedingte Übersetzung

Vorige Seite: A.5.2 Lesen und Setzen **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** B.1 Direktiven

Vorige Seite: B Präprozessor-Direktiven **Eine Ebene höher:** B Präprozessor-Direktiven **Nächste Seite:** B.2 Bedingte Übersetzung

---

B.1 Direktiven

Die sogenannten Präprozessor-Direktiven umfassen die Definition von Makros, das Einfügen von Dateien sowie verschiedene Anweisungen zum Festlegen von Übersetzungsinformationen.

control-line:

```
# define identifier replacement-list new-line
# define idenifier  lparen identifier-listopt replacement-list newline
# undef  identifier newline
# include pp-tokens
# line pp-tokens
# error pp-tokensopt
# pragma pp-tokensopt
```

Eine Präprozessor-Direktive der Form

```
#define identifier replacement-list new-line
```

stellt ein einfaches Makro dar. Diese Direktive veranlaßt den Präprozessor, den *identifier* in der Folge überall dort, wo er auftritt, durch die angegebene Zeichenkette *replacement-list* textuell zu ersetzen. Semikolons in oder am Ende dieser Zeichenkette gehören zu dieser und werden bei der Ersetzung mit eingefügt.

Beispiele:

```
#define MAXINT 32767
#define MY_NAME "THOMAS"
...
if (i>MAX_INT) {
  cout << ... << MY_NAME << ...
...
```

Makronamen werden üblicherweise in Großbuchstaben notiert.

Eine Direktive der Form

```
#define identifier  lparen identifier-listopt replacement-list newline
```

ist eine Makrodefinition mit Argumenten. In der Folge wird der erste Bezeichner mit den folgenden formalen Parametern überall, wo er auftritt, durch *replacement-list* in der Definition ersetzt. Jeder Bezeichner, der in der Definition in der formalen Parameterliste (*identifier-list*) steht, wird durch das entsprechende aktuelle Argument ersetzt.

Die aktuellen Argumente im Aufruf sind ebenfalls Baustein-Zeichenketten, die durch Kommas getrennt werden. Kommas, die in Hochkommas eingeschlossen oder durch Klammern geschützt sind, trennen jedoch keine Argumente. Die Anzahl der formalen Parameter und aktuellen Argumente muß unbedingt übereinstimmen.

Folgendes Codestück zeigt die Definition eines Makros und seine Expansion:

```
#define MIN(a,b) (((a)<(b))?(a):(b))   // Definition
min = MIN(x,y);                        // Verwendung

 // obige Zeile wird wie folgt expandiert:
 // min = (((x)<(y))?(x):(y));
```

Makros sind Zeichenkettenoperationen und ,,wissen`` daher nichts von C++-Syntax, Typen oder Gültigkeitsbereichen. Erst nach erfolgter Expansion durch den Präprozessor ist ein Makro für den Compiler sichtbar. Ein Fehler in einem Makro zeigt sich daher erst nach seiner Expansion, nicht bei seiner Definition. Diese Tatsache kann zu versteckten Fehlern führen, die sehr schwierig zu finden sind.

Beispiel:

```
#define SQUARE(a) a*a
x = SQUARE(x+2);  // expandiert zu: x = x+2*x+2
```

Für beide Formen von Makros gilt, daß eine Definition in der nächsten Zeile fortgesetzt werden darf, wenn die fortzusetzende Zeile durch ein *Backslash* abgeschlossen wird.

Beispiel:

```
#define SWAP(a,b)                \
{                                \
  int h = a;                     \
  a = b;                         \
  b = h;                         \
}
```

Eine Präprozessor-Direktive der Form

`#undef` *`identifier newline`*

hebt die vorangegangene Präprozessor-Definition wieder auf.

Beispiel:

```
#define MIN(a,b) (((a)<(b))?(a):(b))
               // hier ist das Makro MIN definiert
...
#undef MIN
               // ab hier ist MIN undefiniert
```

Eine Präprozessor-Direktive der Form

`#include` *`pp-tokens`*

bewirkt, daß diese Zeile durch den gesamten Inhalt der in *pp-tokens* angegebenen Datei ersetzt wird. Die benannte Datei wird zunächst im aktuellen Verzeichnis gesucht. Endet diese Suche nicht erfolgreich, so wird anschließend in einer Sequenz von spezifizierten Pfaden oder Standardpfaden (*Include-Pfade*) gesucht.

Hat die `#include`-Direktive die Form

```
#include <  <Dateiname>  >
```

so wird nur in den spezifizierten Pfaden oder in der `include`-Standardbibliothek gesucht. Systemdateien werden normalerweise mit `#include <...>` eingebunden (Suche nur in den definierten `include`-Directories beziehungsweise im Compiler-`include`-Verzeichnis), eigene `include`-Dateien mit `#include"..."` (Suche erfolgt zuerst im aktuellen Verzeichnis):

```
#include "main.h"
#include <iostream>
```

Achtung: In einer `#include`-Direktive haben auch Leerzeichen ihre Bedeutung:

```
#include < stdio >       // findet stdio nicht!
```

Eine `#error`-Direktive veranlaßt den Compiler, die Übersetzung abzubrechen und eine Fehlermeldung auszugeben. Der als Meldung spezifizierte Text wird ausgegeben.

Beispiel:

```
#ifndef(MODEL)
#error MODEL nicht definiert
#endif
```

Eine `#line`-Direktive setzt die Numerierung von Quelltextzeilen direkt. Die Nummer der nächsten Zeile wird durch den Wert der Konstante festgelegt. Durch die optionale Angabe eines Dateinamens läßt sich der Compiler einen neuen Dateinamen unterschieben. Wenn kein neuer Dateiname angegeben ist, bleibt der aktuelle Dateiname unverändert.

Beispiel:

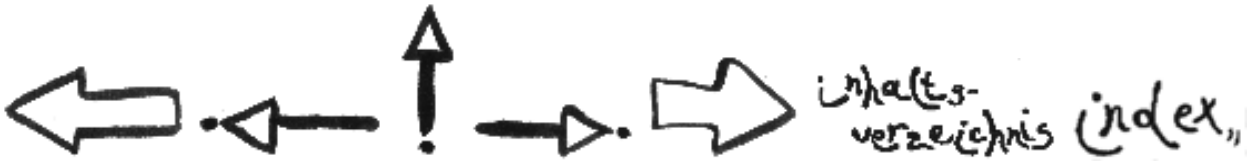
```
#line 67 "main.c"
  // die naechste Zeile im Quelltext erhaelt die Nummer
  // 67, die Quelldatei erhaelt den Namen "main.c"
```

Die Direktive `#pragma` erlaubt die Verwendung implementierungs-spezifischer Direktiven. Mit `#pragma` können beliebige eigene Direktiven definiert werden, ohne dabei mit verschiedenen Compilern in Konflikt zu geraten: Kennt ein Compiler den Direktiven-Namen nicht, so ignoriert er diesen, ohne Warnungen und Fehlermeldungen auszugeben. Weitere Informationen über diese `#pragma`-Direktiven können dem jeweiligen C++-Manual entnommen werden.

Im C++-Standard ist ein Makro `assert` (in der Datei `assert.h`) definiert, mit dem zur Laufzeit Prüfungen vorgenommen werden können. `assert` prüft den als Parameter enthaltenen Ausdruck. Ist der Ausdruck wahr (ungleich 0), so erfolgt keine Aktion, andernfalls wird das Programm mit einer entsprechenden Fehlermeldung abgebrochen.

Beispiel:

```
#include <assert.h>
...
void main()
{
   cin >> i;
   ...
   assert (i!=0);
   cout << "1000 / " << i << " = " << 1000/i << "\n";
   ...
}
```



Vorige Seite: B Präprozessor-Direktiven **Eine Ebene höher:** B Präprozessor-Direktiven **Nächste Seite:** B.2 Bedingte Übersetzung

Vorige Seite: B.1 Direktiven **Eine Ebene höher:** B Präprozessor-Direktiven **Nächste Seite:** C Die Funktion `main`

B.2 Bedingte Übersetzung

```
if-section:
                if-group elif-groupsopt else-groupopt endif-line
if-group:
                # if constant-expression new-line groupopt
                # ifdef identifier new-line groupopt
                # ifndef identifier new-line groupopt
elif-groups:
                elif-group
                elif-groups elif-group
elif-group:
                # elif constant-expression new-line groupopt
else-group:
                # else newline groupopt
endif-line:
                # endif newline
```

Es existieren drei Formen von Präprozessor-Bedingungsanweisungen:

- Eine Anweisung der Form

  `# if` *constant-expression*

  prüft, ob der Ausdruck *constant-expression* einen Wert ungleich 0 ergibt. Der Ausdruck muß ein

Konstantenausdruck sein.

- Ob ein Präprozessor-Bezeichner definiert ist oder nicht, wird mit einer Anweisung der folgenden Form getestet:

  `# ifdef` *`constant-expression`*

- Bei einer Direktive der Form

  `# ifndef` *`constant-expression`*

  wird geprüft, ob der Bezeichner zum aktuellen Zeitpunkt dem Präprozessor nicht bekannt ist.

Nach allen drei Formen können eine beliebige Anzahl von Zeilen stehen, denen

- beliebig viele *elif-group*s,
- optional eine *else-group* sowie
- eine *endif-line*

folgen. Die Konstruktion ähnelt einer ,,normalen`` `if`-Abfrage, allerdings auf Präprozessor-Niveau:

- Die *if-group* und die verschiedenen *elif-group*s (falls diese vorhanden sind) werden geprüft.
- Die erste ,,wahre`` Gruppe wird ausgewählt. Ausgewählt bedeutet, daß alle folgenden Zeilen bis zum Beginn der nächsten *elif-group*, der folgenden *else-group* oder der *endif-line* erhalten bleiben. Alle anderen Zeilen der gesamten *if-section* werden *ignoriert* (= aus dem Source-Text entfernt).
- Sind die *if-group* und keine der angegebenen *elif-group*s wahr, so werden die Zeilen der *else-group* ausgewählt und alle anderen Zeilen der *if-section* ignoriert.
- Sind die *if-group* und keine der angegebenen *elif-group*s wahr, und ist keine *else-group* vorhanden, so wird die gesamte *if-section* ignoriert.

Für alle diese Konstrukte gilt, daß sie auch verschachtelt auftreten dürfen, jedoch dürfen sie sich nicht über ihre Dateigrenzen hinweg erstrecken. Das heißt, daß *endif-line* in derselben Datei wie die *if-group* enthalten sein muß.

Beispiele:

```
#ifdef BORLC
#  define GETVECTOR(ino) getvect(ino)
#  define SETVECTOR(ino, ifun) setvector(ino, ifun)
#else
#  define GETVECTOR(ino) dos_getvect(ino)
#  define SETVECTOR(ino, ifun) dos_setvector(ino, ifun)
#endif
...
#if VERSION == 1
#  include "myfile1.h"
#elif VERSION == 2
#  include "myfile2.h"
```

```
...
#else
#  error Fehler! Version muss im Bereich v. 1 bis X liegen
#endif
```

Vorige Seite: B.1 Direktiven **Eine Ebene höher:** B Präprozessor-Direktiven **Nächste Seite:** C Die Funktion `main`

Vorige Seite: B.2 Bedingte Übersetzung **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** D Lösungen

---

C Die Funktion `main`

`main` ist eine spezielle Funktion, die in jedem Programmsystem nur einmal vorkommen darf und deren Schnittstelle schon vordefiniert ist. Die Funktion wird beim Programmstart automatisch aktiviert.

An `main` können von der Kommandozeile Parameter übergeben werden. Standardmäßig hat `main` zwei Parameter: `argc` und `argv`. Optional kann zusätzlich der Parameter `envp` verwendet werden. Der Prototyp von `main` ist wie folgt definiert:

```
int main(int argc, char* argv[]);
```

beziehungsweise

```
int main(int argc, char* argv[], char* envp[]);
```

`argc` gibt die Anzahl der Argumente an das Programm an und besitzt mindestens den Wert eins, da der Name des Programms automatisch als Argument übergeben wird. `argv` ist ein Feld, das die Argumente enthält. Die Argumente sind Zeichenketten, also Elemente vom Typ `char*`. Das letzte Element in `argv` ist ein Zeiger mit dem Wert `0`.

`envp` ist ebenso aufgebaut wie `argv`, enthält aber Zeiger auf die `environment`-Variablen. `envp` muß nicht verwendet werden. Wird es verwendet, so müssen auch die beiden anderen Parameter deklariert werden.

Der Argumentvektor `argv` enthält im Detail folgende Einträge:

- `argv[0]` enthält den Programmnamen,
- `argv[1]` enthält das erste Argument,
-
- `argv[argc]` enthält den Wert 0.

Das Programmbeispiel:

```
void main(int argc, char* argv[])
{
  for (int i=0; argv[i]!=0; ++i) {
    cout << "argv[" << i << "] = " << argv[i] << "\n";
  }
}
```

liefert für die Befehlszeile

```
setday 1991 3 29
```

folgende Ausgabe:

```
argv[0] = setday
argv[1] = 1991
argv[2] = 3
argv[3] = 29
```

Vorige Seite: B.2 Bedingte Übersetzung **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** D Lösungen

Vorige Seite: C Die Funktion `main` **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** D.1 Mehrfach verwendete Dateien

D Lösungen

Dieses Kapitel zeigt Lösungen zu den Übungen im Buch. Dabei ist folgendes zu beachten:

- Die Lösungen sind ,,mögliche`` Lösungen und nur als Anregungen oder Anleitungen zu verstehen. Sie erheben weder den Anspruch auf Fehlerfreiheit, noch sind sie auf *allen* Entwicklungssystemen lauffähig.
- Die Diskussion der Beispiele ist sehr kurz gehalten und beschränkt sich im wesentlichen auf die Aufzählung der benötigten Dateien sowie einige allgemeine Kommentare.
- Alle Beispiele wurden unter *IBM C Set/2* beziehungsweise *IBM VisualAge/C++* und *GNU C++* entwickelt und später auf die Entwicklungssysteme *Microsoft Visual C++* (Version 4.2) und *Borland C++* (Version 5.0) portiert.

  Im Prinzip sollten alle Beispiele unter *allen* Compilern, die den C++*Draft*-ANSI/ISO-Standard unterstützen, zu übersetzen sein.
- Die Lösung der Aufgabe `Kellerbar` stammt von einem Studenten (Armin Elbs) und ist nur unter *Borland C++* ohne Änderungen übersetzbar. Der Grund dafür liegt in der Verwendung der *Standard Template Library*.
- Auf der *Homepage* des Autors werden Informationen zu den Beispielen abgelegt. Eventuelle Fehlerbeseitigungen oder neuere Versionen werden dort ebenfalls dort zu finden sein.

Bezüglich des ANSI/ISO-Standards gelten aufgrund des aktuellen Stands der Entwicklungssysteme bei den Beispielen folgende Einschränkungen:

- Die Programme verwenden die alte Ein-/Ausgabebibliothek (`iostream.h`), da die neue Bibliothek nur sehr unzureichend unterstützt wird.
- Nicht unterstützte Sprachkonstrukte wie `explicit`, `static_cast`, `dynamic_cast` oder `const_cast` werden ebenfalls noch nicht eingesetzt. In zukünftigen Versionen wird (sofern der Stand der Entwicklungssysteme dies erlaubt) eine entsprechende Nachbesserung der Beispiele erfolgen.
- Die *Standard Template Library* wird nur in einem Beispiel (`Kellerbar`) verwendet. Alle anderen Beispiele verwenden die im Buch realisierten *Collection Classes*.

Vorige Seite: C Die Funktion `main` **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** D.1 Mehrfach verwendete Dateien

Vorige Seite: D Lösungen **Eine Ebene höher:** D Lösungen **Nächste Seite:** D.1.1 Datei `fakeiso.h`

D.1 Mehrfach verwendete Dateien

Einige Dateien werden in mehreren Beispielen verwendet (zum Beispiel die Dateien `tstring.h` und `tstring.cpp` der Klasse `TString`).

Anmerkung: Auf diese Dateien wird in den eigentlichen Lösungen nicht weiter eingegangen.

Vorige Seite: D.1 Mehrfach verwendete Dateien **Eine Ebene höher:** D.1 Mehrfach verwendete Dateien
Nächste Seite: D.1.2 Datei `globs.h`

D.1.1 Datei `fakeiso.h`

Die Datei `fakeiso.h` wird verwendet, um einige ANSI/ISO-Sprachelemente auf ,,älteren`` Entwicklungssystemen zu simulieren.

Dabei wird zum Beispiel der Datentyp `bool` für den *Microsoft C++ 4.2* Compiler als `int` vereinbart. Damit können Programme, die `bool` verwenden, auch auf diesem Entwicklungssystem übersetzt werden.

Vorige Seite: D.1.1 Datei `fakeiso.h` **Eine Ebene höher:** D.1 Mehrfach verwendete Dateien **Nächste Seite:** D.1.3 Klasse `TString`

D.1.2 Datei `globs.h`

`globs.h` stellt einige nützliche Funktions-Templates zur Verfügung. So werden zum Beispiel die verschiedenen Vergleichsoperatoren auf die Operatoren < und == zurückgeführt.

Vorige Seite: D.1.2 Datei `globs.h` **Eine Ebene höher:** D.1 Mehrfach verwendete Dateien **Nächste Seite:** D.1.4 Klasse `Random`

D.1.3 Klasse `TString`

Die Klasse `TString` wurde im Buch bereits ausführlich diskutiert. Sie besteht aus den beiden Dateien `tstring.h` und `tstring.cpp` und stellt die Zusammenfassung der im Klassen-Kapitel angeführten Teile dar.

Vorige Seite: D.1.3 Klasse `TString` **Eine Ebene höher:** D.1 Mehrfach verwendete Dateien

D.1.4 Klasse `Random`

Die Klasse `Random` (Dateien `random.h` und `random.cpp`) stellt einen sehr einfachen Zufallszahlengenerator zur Verfügung. Die Klasse muß initialisiert (`setRand`) und kann verwendet werden, um Zufallszahlen im Bereich von 0 bis einschließlich *x-1* zu erzeugen (`Random::randomize(x)`).

Vorige Seite: D.1.4 Klasse `Random` **Eine Ebene höher:** D Lösungen **Nächste Seite:** D.2.1 Klasse `Date`

D.2 Das Klassenkonzept

Vorige Seite: D.2 Das Klassenkonzept **Eine Ebene höher:** D.2 Das Klassenkonzept **Nächste Seite:** D.2.2 Klasse `Counter`

D.2.1 Klasse `Date`

Die Lösung der Aufgabe umfaßt drei Dateien: `date.h`, `date.cpp` und `datetst.cpp`.

Einige Kommentare zum Source-Code:

- `Date` verwendet die Namen `istream` und `ostream` in der Schnittstelle. Allerdings werden keine weiteren Angaben aus der Datei `iostream` verwendet. Die *Forward*-Deklarationen `class istream;` und `class ostream;` geben die Namen bekannt, ohne daß die entsprechende Datei inkludiert werden muß. Damit werden Übersetzungszeit und Abhängigkeiten minimiert.
- Der *Default*-Konstruktor ist als Konstruktor mit drei vorbelegten Parametern realisiert.
- Alle Daten sind als `private` deklariert.
- Alle *Access*-Methoden sind `inline` codiert.
- Alle lesenden Methoden sind als `const` vereinbart.
- Die Operatoren `>`, `==`, `!=`, `<=` und `>=` werden auf die beiden Operatoren `<` und `==` zurückgeführt und daher als `inline`-Methoden bereits in der Schnittstelle angeführt.
- Die Operatoren `<<` und `>>` greifen auf die Daten eines Datums über *Access*-Methoden zu und müssen daher nicht als `friend` vereinbart werden.

Vorige Seite: D.2.1 Klasse `Date` **Eine Ebene höher:** D.2 Das Klassenkonzept **Nächste Seite:** D.2.3 Gültigkeitsbereich, Sichtbarkeit und

---

D.2.2 Klasse `Counter`

Die Lösung der Aufgabe umfaßt die Dateien `counter.h`, `counter.cpp` sowie `cnttst.cpp` und gestaltet sich einfach:

- `Counter` weist eine statische Instanzvariable `nrOfObjects` auf, die die Anzahl der angelegten Objekte darstellt. Eine weitere statische Instanzvariable (`nrOfObjsAlive`) liefert die Anzahl der „lebenden`` Objekte der Klasse `Counter`. Die Instanzvariable (`id`) gibt die `id` eines `Counter`-Objekts an.
- Im Konstruktor wird die Variable `nrOfObjects` an `id` zugewiesen und anschließend inkrementiert. Außerdem werden entsprechende Meldungen im Konstruktor und Destruktor ausgegeben.
- Alle Instanzvariablen sind als `private` vereinbart. Der Zugriff auf `nrOfObjects` erfolgt über statische `protected`-Methoden. Damit wird erreicht, daß die Zahl der Objekte nur von Objekten der Klasse `Counter` beziehungsweise von Objekten abgeleiteter Klassen verändert werden kann.

Vorige Seite: D.2.2 Klasse `Counter` **Eine Ebene höher:** D.2 Das Klassenkonzept **Nächste Seite:** D.2.4 `Calculator`-Klassen

D.2.3 Gültigkeitsbereich, Sichtbarkeit und Lebensdauer

Zunächst einige Kommentare zu den Variablen des Programms:

- Der Gültigkeitsbereich der globalen Variablen `a` und `b` umfassen das gesamte Programmsystem, also auch eventuell andere vorhandene Dateien. Würde in einer anderen Datei ebenfalls eine globale Variable mit dem Namen `a` oder `b` deklariert, so würde der Linker einen Fehler (Objekt doppelt vorhanden) melden.

  Die globalen Variablen im obigen Beispiel leben zwar während der gesamten Programmausführung, werden aber zum Teil von anderen Variablen überdeckt:

  - Die Klasse `A` verwendet eine Instanzvariable `a`, die innerhalb der Klasse die globale Variable `a` verdeckt.
  - Die drei Funktionen `exchange1`, `exchange2` und `exchange3` weisen Parameter mit den Namen `a` und `b` auf. Diese Parameter können als lokale Variablen der Funktionen betrachtet werden und überdecken in ihrem Gültigkeitsbereich ebenfalls die globalen Objekte `a` und `b`.
  - Die Funktion `exchange2` weist einen inneren Block auf, in dem eine weitere Variable mit dem Namen `b` deklariert wird. Diese Variable überdeckt sowohl das globale `b` als auch den Parameter `b`.
- Die Instanzvariable `c` der Klasse `A` weist den Gültigkeitsbereich der Klasse auf und lebt, solange das Objekt `o` lebt.
- Die statische Instanzvariable `a` der Klasse `A` weist den Gültigkeitsbereich der Klasse auf. Die Variable ist allerdings als `static` deklariert und lebt daher von ihrer Initialisierung an (`int A::a = 12`) bis zum Programmende.

  Auf dieses Objekt kann durch Verwendung des *Scope*-Operators zugegriffen werden (`A::a`).

Bei der Feststellung der Ausgabe müssen ferner noch die verschiedenen Arten der Parameterübergabe in den Funktionen `foo1`, `foo2` und `foo3` beachtet werden. Die Ausgabe des Programms ist wie folgt:

```
M1: a = 2 b = 3
M2: a = 0 b = 3
M3: a = 0 b = 3
```

```
A1: a = 19 b = 4
A2: a = 19 b = 4
A3: a = 8 b = 4
M4: a = 5 b = 3
A1: a = 41 b = 6
A2: a = 41 b = 6
A3: a = 12 b = 6
M5: a = 3 b = 3
```

Vorige Seite: D.2.2 Klasse `Counter` **Eine Ebene höher:** D.2 Das Klassenkonzept **Nächste Seite:** D.2.4 `Calculator`-Klassen

Vorige Seite: D.2.3 Gültigkeitsbereich, Sichtbarkeit und **Eine Ebene höher:** D.2 Das Klassenkonzept
Nächste Seite: D.2.5 Klasse `List`

D.2.4 `Calculator`-Klassen

Die Lösung des `Calulator`-Beispiels besteht aus folgenden Komponenten:

- „Hilfsdateien`` (`fakeiso.h`, `globs.h`)
- Klasse `TString` (`tstring.h`, `tstring.cpp`)
- Klasse `Stack` (`stack.h`, `stack.cpp`)
- Eine „Hilfsfunktion`` für die Konvertierung `char*` - `int` (`strhlp.h`, `strhlp.cpp`)
- Klasse `Calculator` (`calc.h`, `calc.cpp`)
- Das Testprogramm (`main.cpp`)

Die Hilfsfunktion `strhlp.h` wird verwendet, um die Konvertierung von Zeichenketten in entsprechende Zahlen zu zeigen. In der Regel wird dafür natürlich keine eigene Funktion erstellt, sondern es wird die bereits vorhandene Funktion `atoi` der Standardbibliothek verwendet.

Die anderen Programmteile enthalten keinerlei Besonderheiten. Die Funktionalität von `Stack` wurde bereits besprochen und die Klasse `Calculator` weist lediglich einfache Methoden auf.

Vorige Seite: D.2.4 `Calculator`-Klassen **Eine Ebene höher:** D.2 Das Klassenkonzept **Nächste Seite:** D.2.6 Klasse `Rational`

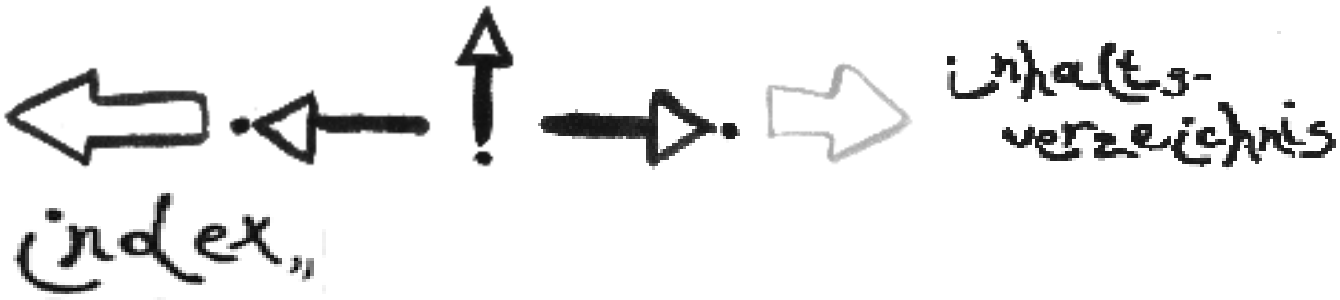
D.2.5 Klasse `List`

Die Lösung der Aufgabe `List` umfaßt folgende Dateien:

- „Hilfsdateien`` (`fakeiso.h`, `globs.h`)
- Die Klasse `TString` (`tstring.h`, `tstring.cpp`)
- Die Klasse `List` (`strlist.h`, `strlist.cpp`)
- Das Testprogramm (`listst.cpp`)

Einige Bemerkungen:

- Die interne Klasse `Node` wird in der Schnittstelle von `List` deklariert, aber erst in der Implementierungsdatei definiert.
- Das geforderte „Gedächtnis`` der Klasse besteht aus einem einzelnen Zeiger, der das zuletzt referenzierte Listenelement angibt.



Vorige Seite: D.2.5 Klasse `List` **Eine Ebene höher:** D.2 Das Klassenkonzept

D.2.6 Klasse `Rational`

Die Lösung der Aufgabe umfaßt folgende Dateien:

- „Hilfsdateien`` (`fakeiso.h`, `globs.h`)
- Die Klasse `TString` (`tstring.h`, `tstring.cpp`)
- Die Klasse `Rational` (`rational.h`, `rational.cpp`)
- Das Klassen-Template `List` (`list.h`)
- Das Testprogramm (`rattst.cpp`)

Einige Bemerkungen zur Klasse `Rational`:

- Einige Operatoren sind (wie in der Angabe gefordert) außerhalb der Klasse als `friend`-Funktionen realisiert.
- Die Klasse prüft die Operationen bezüglich Bereichsüberschreitungen. Im Fall einer Bereichsüberschreitung wird ein entsprechendes Fehler-*Flag* gesetzt.

  Um zu prüfen, ob eine Operation eine Bereichsüberschreitung auslöst, muß festgestellt werden, ob das Ergebnis im zulässigen Bereich liegt (kleinergleich der größten zulässigen Zahl und größergleich der kleinsten zulässigen Zahl).

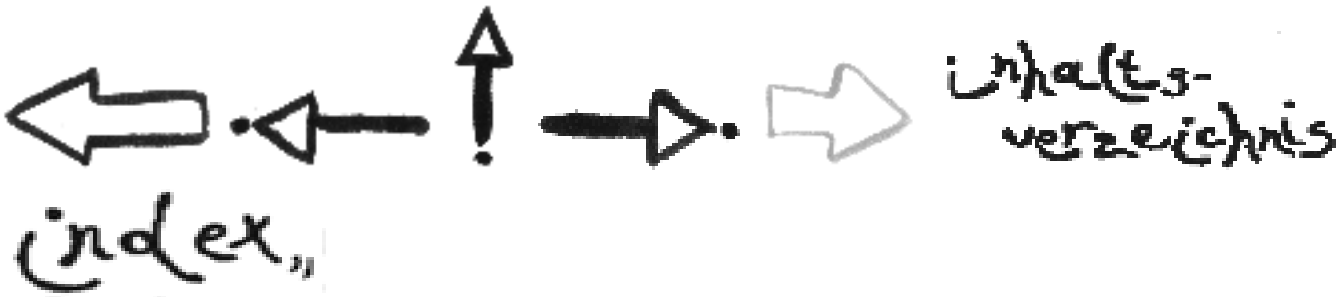
  Dazu genügt es nicht, eine Abfrage der Art `if a + b < MAX_ZAHL` zu verwenden. In diesem Fall kann die Abfrage an sich bereits die Bereichsüberschreitung auslösen!

  Die Abfrage muß umgeformt werden: `if a < MAX_ZAHL - b` bewegt sich bezüglich einer Bereichsüberschreitung „nach oben hin`` im zulässigen Bereich!
- Die größten beziehungsweise kleinsten Zahlenwerte eines Typs sind in der Datei `limits` (ANSI/ISO-Standard) beziehungsweise `limits.h` festgelegt. Je nach Compiler wird eine der beiden Varianten verwendet. Zur Unterscheidung sind entsprechende Präprozessoranweisungen codiert.
- Zur Berechnung werden einige interne Hilfsfunktionen verwendet. Diese sind in einem anonymen Namensraum in der Datei `rational.cpp` abgelegt und daher nur intern sichtbar.

Auch die internen Funktionen prüfen bei allen Operationen auf eventuelle Bereichsüberschreitungen.

- Jede Zahl wird nach jeder Operation vereinfacht (`simplify`).

Der Testtreiber ist sehr einfach aufgebaut und erlaubt es, die verschiedenen Operationen der Klasse mit einzugebenden Daten durchzuführen.



Vorige Seite: D.2.5 Klasse `List` **Eine Ebene höher:** D.2 Das Klassenkonzept

Vorige Seite: D.2.6 Klasse `Rational` **Eine Ebene höher:** D Lösungen **Nächste Seite:** D.3.1 Binär Suchen

D.3 Templates

Vorige Seite: D.3 Templates **Eine Ebene höher:** D.3 Templates **Nächste Seite:** D.3.2 `List`-Template

D.3.1 Binär Suchen

Die Lösung der Aufgabe umfaßt folgende Elemente:

- „Hilfsdateien`` (`fakeiso.h`, `globs.h`)
- Das Funktionstemplate `binSearch`
- Das Testprogramm `main`

Die Implementierung der Funktion ist einfach. Sie entspricht dem in Sedgewick92 (siehe Buch) angeführten Algorithmus.

Die beiden Fragen können wie folgt beantwortet werden:

Die Funktion kann grundsätzlich mit allen angegebenen Typen ausgeprägt werden. Zu beachten ist, daß in der Funktion Elemente *verglichen* werden. Die Typen, mit denen `binSearch` also ausgeprägt wird, müssen die verwendeten Vergleichsoperatoren zur Verfügung stellen *und* sie entsprechend der Vergleichssemantik implementieren.

Im Fall von `char*` zum Beispiel vergleicht die Anweisung `pChar1 < pChar2` *nicht* die beiden Zeichenketten, sondern die beiden Zeigerwerte. `binSearch` arbeitet also in diesen Fällen nicht korrekt!

Vorige Seite: D.3.1 Binär Suchen **Eine Ebene höher:** D.3 Templates **Nächste Seite:** D.3.3 Klasse `Tree`

D.3.2 `List`-Template

Die Lösung der Aufgabe umfaßt folgende Dateien:

- „Hilfsdateien`` (`fakeiso.h`, `globs.h`)
- Die Klasse `TString` (`tstring.h`, `tstring.cpp`)
- Das Klassen-Template `List` (`list.h`)
- Das Testprogramm (`listst.cpp`)

Folgende Punkte sind bei der angeführten Implementierung zu beachten:

- Das Klassen-Template ist vollständig `inline` gelöst. Grund dafür sind ausschließlich Compiler-Beschränkungen!
- `Node` ist eine `struct` und in `List` vereinbart. Nur dort macht die Klasse Sinn. `Node` enthält (wie gefordert) den Datenteil und *zwei* Verweisteile (`prev`, `next`). Der Grund liegt in den zu implementierenden Iterator-Methoden: Die Operation `--` kann nur dann effizient erfolgen, wenn der Vorgänger eines beliebigen Listenelements schnell und einfach gefunden werden kann.
- `List` weist dynamische Datenstrukturen auf und ist nicht automatisch in kanonischer Form. Daher werden Zuweisungsoperator und *Copy*-Konstruktor implementiert. Beide Methoden sind sehr einfach und werden mit dem entsprechenden Iterator realisiert.
- `Iterator` ist intern zu `List` vereinbart. `Iterator` weist einige Besonderheiten auf:
  - Die Klasse ist auf den ersten Blick nicht in kanonischer Form, da *Copy*-Konstruktor und Zuweisungsoperator die Elemente, auf die die Zeiger `posPtr` und `pList` verweisen, nicht kopieren. Allerdings macht in diesem Fall das Kopieren von Elementen keinen Sinn, da ein `Iterator`-Objekt nur auf die entsprechenden Daten *verweist*, sie aber nicht allokiert.

    Die Klasse kann daher so wie angegeben verwendet werden, oder es können *Copy*-Konstruktor und Zuweisungsoperator als `private` deklariert werden. Hier wurde der erste Weg gewählt, da das Kopieren von Iteratoren eine durchaus sinnvolle Operation darstellt.
  - Die einzelnen Zugriffsfunktionen prüfen jeweils, ob der Iterator-Schlüssel mit dem Schlüssel des assoziierten `List`-Objekts übereinstimmt. Stimmen die beiden nicht überein, so wird das Programm abgebrochen. Das ist eine nicht sehr zufriedenstellende Lösung, besser wäre es, entsprechende Ausnahmen zu vereinbaren (siehe Kapitel 15).

Bei der Implementierung fällt vor allem die etwas umständliche und teilweise befremdend wirkende Schreibweise bei Templates auf. Der Vorteil der allgemeinen Verwendbarkeit wird in C++ mit zusätzlichem Codieraufwand und (zur Zeit noch) mit etlichen Compiler-Problemen erkauft.

Kommentar zur Implementierung:

1. Im Destruktor für die Klasse `List` müssen alle Listen-Elemente zerstört werden. Dies wird einfach dadurch erreicht, daß solange Elemente entfernt werden, bis die Liste leer ist.
2. Bei der Schnittstelle ist der Konstruktor für die Klasse Iterator zu beachten. Der Konstruktor verbindet den neu angelegten Iterator mit der in `overTheList` angegebenen Liste, indem er deren Adresse in einer Instanzvariable ablegt. Dieser Parameter muß unbedingt eine Referenz sein, da ansonsten beim Aufruf des Konstruktors eine temporäre Kopie der Liste angelegt wird, auf die dann verwiesen wird. Diese Kopie wird beim Verlassen der Element-Funktion zerstört und der Iterator verweist auf einen nicht allokierten Speicherplatz!
3. Die Implementierung des Iterators ist sehr vereinfacht und nur rudimentär. Wird eine bestehende Liste verändert, so erfolgt keinerlei ,,Meldung`` an eventuell vorhandene Iteratoren, das heißt es ist möglich einen Iterator `i` mit einer Liste `l` zu initialisieren, so daß `i` auf das erste Element in `l` zeigt. Nun kann mit der Methode `remove` das erste Listenelement entfernt werden. Der Iterator referenziert damit einen ungültigen Speicherplatz (siehe dazu [Mur93]).

Vorige Seite: D.3.1 Binär Suchen **Eine Ebene höher:** D.3 Templates **Nächste Seite:** D.3.3 Klasse `Tree`

Vorige Seite: D.3.2 `List`-Template **Eine Ebene höher:** D.3 Templates

---

D.3.3 Klasse `Tree`

Die Lösung der Aufgabe umfaßt folgende Dateien:

- „Hilfsdateien`` (`fakeiso.h`, `globs.h`)
- Die Klasse `TString` (`tstring.h`, `tstring.cpp`)
- Das Klassen-Template `Tree` (`tree.h`)
- Das Testprogramm (`test.cpp`)

Einige Bemerkungen zum Klassen-Template `Tree`:

- Das Template ist (wie bei den anderen Templates) aufgrund von Compiler-Restriktionen vollständig `inline` codiert.
- Die Realisierung der „normalen`` Baum-Funktionalität sollte keine größeren Probleme verursachen und entspricht weitgehend Standard-Algorithmen.
- Die Realisierung des Baum-Iterators entspricht ebenfalls weitgehend den bereits bekannten Realisierungen (zum Beispiel in der Klasse `List`).

  Aufwendiger und schwerer verständlich sind die Element-Funktionen `nextElem` und `prevElem`, die die „Iterator um ein Element vorwärts``- beziehungsweise „Iterator um ein Element rückwärts``-Funktionalität realisieren. Die Suche nach dem logisch nächsten Element erfolgt (vereinfacht ausgedrückt) ähnlich wie die Suche nach dem Element selbst. Der Leser sei dazu angehalten, den Ablauf auf einem Zettel oder mittels *Debugger* nachzuvollziehen.

Vorige Seite: D.3.3 Klasse `Tree` **Eine Ebene höher:** D Lösungen **Nächste Seite:** D.4.1 Klasse `Word`

D.4 Vererbung

Vorige Seite: D.4 Vererbung **Eine Ebene höher:** D.4 Vererbung

---

D.4.1 Klasse `Word`

Die Lösung der Aufgabe umfaßt folgende Dateien:

- „Hilfsdateien`` (`fakeiso.h`, `globs.h`)
- Die Klasse `TString` (`tstring.h`, `tstring.cpp`)
- Die Klasse `Word` (`word.h`, `word.cpp`)
- Die Klasse `List` (`list.h`)
- Die Klasse `SpellChecker` (`spellchk.h`, `spellchk.cpp`)
- Das Testprogramm (`splchk.cpp`)

Einige Bemerkungen zur Klasse `Word`:

- Die Klasse ist `public` von `TString` abgeleitet und stellt damit alle ererbten Element-Funktionen zur Verfügung.
- Die zusätzlichen Wortformen sind in einer Ausprägung des Klassen-Templates `List` abgelegt.
- Der Eingabeoperator liest die Stammform eines Worts und (optional) verschiedene andere Formen ein. Die einzelnen Wörter sind durch eine Tilde und die verschiedenen Wortformen durch ein Leerzeichen getrennt.
- Die Instanzvariable `subWords` der Klasse `Word` ist eine Liste, die mittels dynamischer Datenstrukturen realisiert ist. Allerdings befindet sich die Klasse `List` bereits in kanonischer Form. Damit ist auch `Word` in kanonischer Form.

Zwei Anmerkungen zur Klasse `SpellChecker`:

- Die Klasse ist bewußt einfach gehalten und verfügt lediglich über die unbedingt notwendige Funktionalität.
- Auch `SpellChecker` ist bereits in kanonischer Form, da alle verwendeten Instanzvariablen ebenfalls in kanonischer Form sind und die Klasse keine dynamischen Datenstrukturen enthält.

Vorige Seite: D.4 Vererbung **Eine Ebene höher:** D.4 Vererbung

Vorige Seite: D.4.1 Klasse `Word` **Eine Ebene höher:** D Lösungen **Nächste Seite:** D.5.1 Animals

D.5 Polymorphismus

Vorige Seite: D.5 Polymorphismus **Eine Ebene höher:** D.5 Polymorphismus **Nächste Seite:** D.5.2 `CellWar`

D.5.1 Animals

Die Lösung der Aufgabe umfaßt folgende Dateien:

- „Hilfsdateien`` (`fakeiso.h`, `globs.h`)
- Die Klasse `TString` (`tstring.h`, `tstring.cpp`)
- Die Klasse `Random` (`random.h`, `random.cpp`)
- Die Klasse `Object` (`object.h`, `object.cpp`)
- Die Klasse `Animal` (`animal.h`, `animal.cpp`)
- Die Klasse `Mammal` (`mammal.h`, `mammal.cpp`)
- Die Klasse `Bird` (`bird.h`, `bird.cpp`)
- Die Klasse `Parrot` (`parrot.h`, `parrot.cpp`)
- Das Klassen-Template `List` (`list.h`)
- Das Testprogramm (`test.cpp`)

Neben den eigentlichen Tierklassen werden drei zusätzliche Klassen verwendet:

- Gattungsnamen und andere Informationen werden mit der Klasse `TString` abgelegt.
- Ein `Parrot` verfügt über einen beliebig großen Wortschatz, aus dem zufällig Wörter ausgewählt werden. Um den Wortschatz ablegen zu können, wird das `List`-Template verwendet.
- Um die Anforderung der zufälligen Auswahl von Wörtern erfüllen zu können, wird die Klasse `Random` benötigt.

Die führt zu der in Abbildung 14.5 angegebenen Hierarchie.

In dieser Abbildung ist die Beziehung zwischen `List`, `TString`, `List<TString>` und `Parrot` zu beachten: `List` ist ein Klassen-Template, die mit `TString` ausgeprägt wird. Die „Zwischen-Klasse``, die sich daraus ergibt (= `List<TString>` in der Abbildung), ist im Programm durch die Deklaration `List<TString>` gegeben.

Die Methode `isKindOf` benutzt den Namen der eigenen Klasse (RTTI!), um festzustellen ob die *isKindOf*-Beziehung zutrifft: Ein Objekt *ist ein* (*isKindOf*) `className`, wenn

- `className` der Name der eigenen Klasse ist oder

- `isKindOf(className)` bei einer der Basisklassen gilt.

Mit Hilfe dieser Beziehung kann `isKindOf` sehr einfach implementiert werden. Wichtig ist, `isKindOf` als `virtual` zu spezifizieren.

`debug` gibt Name, Klasse und Instanzvariablen eines Objekts aus. Die Methode muß also in jeder Klasse, die eine neue Instanzvariable definiert, überschrieben werden und virtuell sein. Auch `print` muß in jenen `Animal`-Klassen neu definiert werden, die ,,neue`` Eigenschaften vereinbaren und ist daher ebenfalls `virtual`.

Das angegebene Beispielprogramm legt einige Tiere an und führt verschiedene Operationen mit ihnen durch.

Vorige Seite: D.5 Polymorphismus **Eine Ebene höher:** D.5 Polymorphismus **Nächste Seite:** D.5.2 `CellWar`

Vorige Seite: D.5.1 Animals **Eine Ebene höher:** D.5 Polymorphismus **Nächste Seite:** D.5.3 `Maze`

D.5.2 `CellWar`

Die Lösung Aufgabe umfaßt folgende Dateien:

- „Hilfsdateien`` (`fakeiso.h`, `globs.h`)
- Die Klasse `TString` (`tstring.h`, `tstring.cpp`)
- Die Klasse `Random` (`random.h`, `random.cpp`)
- Die Klassen `Cell`, `NullCell`, `AggressiveCell`, `KillerCell` und `FloodCell` (`cells.h`, `cells.cpp`)
- Die Klasse `CellSimulation` (`cellsimu.h`, `cellsimu.cpp`)
- Das Testprogramm (`main.cpp`)

Die `Cell`-Klassen stellen die verschiedenen Zellen in der Simulation dar. Sie haben alle ein unterschiedliches Verhalten (realisiert in der Element-Funktion `act`) und Wachstum (realisiert in der Element-Funktion `grow`).

Die verschiedenen Implementierungen sind hier nicht diskutiert, dazu sei auf die Datei `cells.cpp` verwiesen.

Die `Cell`-Simulation selbst stellt Methoden zum Setzen und Löschen von Zellen zur Verfügung. Die Realisierung ist problemlos, lediglich beim Destruktor ist zu beachten, daß alle Verweise auf eine Zelle `c` in den Nachbarzellen gelöscht werden *bevor* `c` selbst gelöscht wird.

Vorige Seite: D.5.2 `CellWar` **Eine Ebene höher:** D.5 Polymorphismus **Nächste Seite:** D.5.4 Klassenhierarchie `Kellerbar`

D.5.3 `Maze`

Die Lösung der Aufgabe umfaßt folgende Dateien:

- ,,Hilfsdateien`` (`fakeiso.h`, `globs.h`)
- Die Klasse `TString` (`tstring.h`, `tstring.cpp`)
- Die Klasse `MAZMaze` (`mazmaze.h`, `mazmaze.cpp`)
- Die Klasse `MAZMapSite`, `MAZRoom`, `MAZDoor` und `MAZWall` (`mazelems.h`, `mazelems.cpp`)
- Die Klasse `MAZFileMazeBuilder` (`mazflmzb.h`, `mazflmzb.cpp`)
- Die Klasse `MAZMazeView` (`mazv.h`, `mazv.cpp`)
- Das Testprogramm (`mazetest.cpp`)

`MAZDoor` ist eine Klasse, die eine Referenz auf zwei Räume enthält. Die Tür verbindet die beiden Räume miteinander und kann geöffnet oder geschlossen werden. Ist eine Tür offen, so kann durch sie der andere Raum betreten werden.

`MAZMaze` ist das eigentliche Labyrinth und enthält eine Ansammlung von Räumen, die durch Türen verbunden sind. Ein Labyrinth kann mit einem Objekt der Klasse `MAZFileMazeBuilder` erzeugt werden.

Für einfache Tests ist zudem die Klasse `MAZMazeView` angeführt, die bestehende Labyrinthe am Bildschirm ausgibt. Dabei gilt die Einschränkung, daß nur ,,einfache`` Labyrinthe gedruckt werden können und der oberste linke Raum übergeben werden muß.

Die Lösung orientiert sich an [GHJV95] und geht aus den oben angeführten Anforderungen hervor. Zu bemerken sind folgende Punkte:

- Labyrinthe bestehen aus einer Ansammlung von Räumen, die in einem Vektor abgelegt werden.
- Es werden keine *Copy*-Konstruktoren oder Zuweisungs-Operatoren implementiert, obwohl diese aufgrund der dynamischen Datenstrukturen) nötig wären. Da aber keine Labyrinth-Objekte kopiert werden, werden *Copy*-Konstruktor und Zuweisungsoperator als `private` vereinbart.
- Im Destruktor der Klasse `Door` ist zu beachten, daß das Deallokieren einer Tür zu ungültigen Verweisen führt, da jede Tür von zwei Räumen referenziert wird. Eine Tür kann daher erst deallokiert werden, wenn kein Raum mehr einen Verweis auf sie enthält.

Vorige Seite: D.5.2 `CellWar` **Eine Ebene höher:** D.5 Polymorphismus **Nächste Seite:** D.5.4 Klassenhierarchie `Kellerbar`

Vorige Seite: D.5.3 `Maze` **Eine Ebene höher:** D.5 Polymorphismus

---

Teilabschnitte

D.5.4 Klassenhierarchie `Kellerbar`

Die angeführte Lösung der Kellerbar-Aufgabe stammt von Armin Elbs (Danke!) und nicht von mir. Auch die Diskussion der Lösung beruht zum größten Teil auf seiner Übung.

Die Aufgabe ist umfangreicher und zum Teil auch komplexer als die anderen angeführten Beispiele und wird daher auch wesentlich ausführlicher diskutiert.

Die Lösung umfaßt folgende Dateien:

- „Hilfsdateien`` (`fakeiso.h`, `globs.h`)
- Die Klasse `TString` (`tstring.h`, `tstring.cpp`)
- Die Klasse`Random` (`random.h`, `random.cpp`)
- Die Klassen `SimObject`, `Event`, `EventComp` und `Simulation` (`simulati.h`, `simulati.cpp`)
- Die Klassen `Drink`, `Guest`, `Waiter` und `Kellerbar` (`kellerba.h`, `kellertba.cpp`)
- Die Klasse `Bier`, `Coke`, `Mineral`, `Schnaps`, `Wein`, `Alkoholiker`, `LimoTrinker`, `MineralTrinker`, `Mechatroniker` und `Informatiker` (`5und5.h`, `5und5.cpp`)
- Ein „Schwindel`` (*Fake*) zur Ausprägung von Templates (`fake.cpp`)
- Zwei Testprogramme (`keller1.cpp` und (`keller2.cpp`)

Die Kellerbar basiert auf den Klassen des Simulations-*Frameworks*(`Simulation`, `SimObject`, und einem Zufallszahlengenerator) konstruiert. Tatsächlich kommen dann spezielle, von diesen Basen

abgeleiteten Klassen, zum Einsatz. So wird die `Kellerbar` selbst beispielsweise von `Simulation` abgeleitet. In der `Kellerbar` kommen dann keine Instanzen von `Simulation`, sondern die von spezielleren, abgeleiteten Klassen wie zum Beispiel `Waiter`, `Guest` und andere zum Einsatz.

Die Klasse `SimObject`

`SimObject` ist die Basisklasse für alle Simulationsobjekte.

Einige Kommentare zu den wesentlichen Element-Funktionen der Klasse:

- `startUp` führt alle nötigen Basis-Initialisierungen durch. Die abgeleiteten Klassen müssen sofort dafür sorgen, daß das Objekt zur richtigen Zeit in die Simulationsschlange eingereiht wird.
- `finishUp` wird vor dem Verlassen der Simulation aufgerufen.
- `enter` speichert eine Referenz zur Simulation, in der dieses Objekt abläuft, und ruft die Methode `startUp` auf.
- `act` ist die Hauptmethode für alle Simulationsobjekte. Sie führt alle wesentlichen Aktionen des Objekts durch. Falls erforderlich fügt sich das Objekt am Ende von `act` wieder mittels `sleepFor` (Methode der Klasse `Simulation`) in die Simulationsschlange ein und gibt `true` zurück. Ansonsten wird nach dem Aufruf von `finishUp` durch den Rückgabewert `false` die Simulation verlassen. Die Simulation selbst sorgt dann für die Freigabe des Speicherplatzes. Per Voreinstellung wird hier in der Basisklasse `false` zurückgegeben.

Zwei weitere Klassen im Zusammenhang mit `SimObject` sind `Event` und `EventComp`:

- Die Klasse `Event` „verpackt`` die in der Simulations-Warteschlange zu speichernden Simulationsobjekte und versieht sie mit einem Zeitstempel.
- `EventComp` stellt eine Vergleichsmethode für zwei `Event`-Objekte zur Verfügung.

Die Klasse `Simulation`

Die Klasse `Simulation` ist die Basisklasse für Simulationen.

Einige Kommentare zu den wichtigsten Element-Funktionen beziehungsweise Datenelemente:

- `startUp` führt die nötigen Initialisierungen durch (zum Beispiel das Setzen der Simulationszeit auf 0).
- `finishUp` übernimmt alle notwendigen De-Initialisierungen: die Simulations- und die Warteschlange werden geleert und dabei die enthaltenen Simulationsobjekte freigegeben.
- `enterNewObj` fügt der Simulation ein neues Simulationsobjekt hinzu und ruft dessen Methode `enter` auf.
- `removeFirstSimObj` entfernt das erste Simulationsobjekt aus der Simulationsschlange und liefert dessen Adresse. Der Rückgabewert ist 0, wenn kein Simulationsobjekt mehr vorhanden ist.
- `getFirstSimObj` liefert Adresse und Anreihungszeit des ersten Simulationsobjekts in der Simulationsschlange. `getFirstSimObj` kehrt mit 0 zurück, falls kein Objekt mehr enthalten ist. Die Zeit `timeScheduledAt` wird in diesem Fall auf `simTime` gesetzt.

- `sleepFor` reiht das am Zeiger `aSimObj` hängende Objekt in der Simulationsschlange an der Stelle `simTime` + `forTheTime` ein. Damit wird für die Zeit `forTheTime` keine Operation an diesem Objekt ausgeführt.
- `addWaitingSimObj` reiht das am Übergabezeiger hängende Objekt an die Warteschlange.
- `removeNextWaitingSimObj` entfernt das erste Objekt aus der Warteschlange und liefert dessen Adresse.
- `performNextCycle` ist der „Taktgeber`` der Simulation. Bei jedem Aufruf der Methode wird ein kompletter Zeitschritt abgearbeitet. Die Methode ermittelt das nächste Simulationsobjekt der Simulationsschlange mit der Methode `getFirstObj` und verwendet das Feld `simTime` als Rückgabeparameter für die Zeit.

  Falls kein Objekt mehr vorhanden ist (0 wurde von `getFirstSimObj` geliefert), wird 0 zurückgegeben. Ansonsten wird die globale Zeit der Simulation auf die neue Zeit vorgerückt.

  Anschließend wird wiederholt `removeFirstObj` gefolgt von einem Aufruf von `act`, gefolgt von `getFirstObj` ausgeführt, solange `getFirstObj` eine der aktuellen `simTime` entsprechende Zeit liefert. Falls `act` den Wert `false` liefert, wird das entsprechende Objekt aus der Simulation entfernt.
- `getSimTime` liefert die aktuelle Globalzeit der Simulation.
- `simTime` ist die globale Simulationszeit. Die Zeit wird ausschließlich von der Simulation verwaltet.
- `waitingSimObjs` ist eine Instanz einer *STL*-Klasse. Alle Simulationsobjekte, die auf eine Operation (einer Ressource) warten, sind in dieser Warteschlange angereiht und somit vorübergehend deaktiviert.
- `simQueue` ist eine Prioritätswarteschlange und ebenfalls als *STL*-Klasse realisiert. Diese Prioritätswarteschlange enthält alle an der Simulation beteiligten Objekte aufsteigend nach deren Anreihungszeit sortiert.

Die Simulation verwaltet alle am Geschehen beteiligten Objekte in einer *Queue*, in der eine zeitliche Rangordnung herrscht. Die aktuell zu bedienenden Objekte stehen ganz vorne und haben eine zeitlich höhere Priorität (niederer Zeitwert) als die weiter hinten (hoher Zeitwert).

Das Abarbeiten erfolgt durch die Funktion `performNextCycle`:

Kellner, Getränke und Gäste

Die Klassen `Waiter`, `Drink` und `Guest` sowie die von ihnen abgeleiteten Klassen (siehe `5und5.h`) werden nicht mehr im Detail besprochen. Ihre Realisierung ist relativ einfach und kann anhand des Sourcecodes nachvollzogen werden.

Hier sei lediglich der grundsätzliche Ablauf der `Kellerbar`-Simulation angegeben. Er kann wie folgt beschrieben werden:

Das Geschehen wird durch `Waiter` und `Guest` bestimmt.

Ein Kellner holt sich in seiner `act`-Methode von der Simulation einen wartenden Gast (mittels `getWaitingGuest`). Falls keiner wartet, wird für fünf Zeiteinheiten die Bar aufgeräumt (`sleepFor`). Ansonsten wird der Gast bedient. Dazu wird dieser befragt, welchen *Drink* er möchte (`whichDrink`). Will der Gast nichts mehr trinken, so wird er aus der Bar gewiesen. Andernfalls wartet er für die Zeit, die das Einschenken in Anspruch nimmt (`sleepFor`).

Danach wird der Gast mittels `consumeDrink` zum Trinken aufgefordert. Er reiht sich selbständig in die Simulations-*Queue* ein, nachdem der Kellner ihm mitgeteilt hat, wie lange er für den Genuß des Getränks zu brauchen habe. Zuletzt reiht sich dann der Kellner selbst wieder in die Simulationsschlange ein.

Aus der Sicht eines Gasts stellt sich das Geschehen so dar, daß er den Kellner über `callWaiter` ruft und sich in die Warteschlange einreiht. Alles weitere wird vom Kellner übernommen und läuft wie oben beschrieben ab.

Generierung von Ereignissen (Simulationsobjekten)

Ausgehend von der Gleichverteilung des Zufallszahlengenerators `Random` wird die Getränkeauswahl eines Gasts durch Gewichtung ermittelt.

Das Trinkverhalten eines Durchschnittsgasts:

Bier	Wein	Coke	Mineral	Schnaps
55%	5%	20%	15%	5%

Der Generator wird angewiesen, eine Zahl im Bereich von 1 bis 100 zu generieren. Anschließend wird geprüft, welches Getränk ausgewählt wird. Dies erfolgt durch ein schrittweises Aufsummieren der Prozentsätze in der Liste.

Die Anzahl der Getränke eines Gasts wird ebenfalls über den Zufallszahlengenerator festgelegt. Es kann angenommen werden, daß jeder Gast, der die Bar betritt mindestens ein Getränk konsumiert. Mit steigender Verweilzeit und steigender Anzahl konsumierter Getränke wird es jedoch immer unwahrscheinlicher, daß eine Nachbestellung erfolgt.

Die Anzahl der Gäste im Lokal orientiert sich an Beobachtungen: Erfahrungsgemäß gibt es eine Stoßzeit im Lokal, in der es zu einem übermäßigen Andrang kommt. Die Zufallszahl der pro Zeitabschnitt (zum Beispiel 10 Minuten) eintreffenden Gäste könnte gegebenenfalls durch eine (logarithmische-)Normalverteilung angenommen werden. Im Beispielprogramm ist ,,lediglich`` eine Normalverteilung realisiert.

Vorige Seite: D.5.3 `Maze` **Eine Ebene höher:** D.5 Polymorphismus

Vorige Seite: Generierung von Ereignissen (Simulationsobjekten) **Eine Ebene höher:** D Lösungen
Nächste Seite: D.6.1 Klasse `TString` II

D.6 Ausnahmen

Vorige Seite: D.6 Ausnahmen **Eine Ebene höher:** D.6 Ausnahmen **Nächste Seite:** D.6.2 Die Klasse `Rational` II

D.6.1 Klasse `TString` II

Die Lösung der Aufgabe umfaßt folgende Dateien:

- Allgemeine ,,Hilfsdateien`` (`fakeiso.h`, `globs.h`)
- Die ,,Hilfsdatei`` für die Standard-Ausnahmen (`fakeexc.h`)
- Die Klasse `TString` (`tstring.h`, `tstring.cpp`)
- Das Testprogramm (`tststrtst.cpp`)

Die Standard-Bibliothek mit den im Standard definierten Ausnahmen ist nur auf wenigen Entwicklungssystemen verfügbar. Um die Standard-Bibliothek dennoch nutzen zu können, werden die Standard-Klassen ,,nachimplementiert`` (Datei `fakeexc.h`). Diese Klassen entsprechen weitgehend den Standard-Klassen, sind aber in einem eigenen Namensraum gekapselt, um Namenskonflikte zu vermeiden.

Ferner ist anzumerken, daß zur Zeit Ausnahmespezifikationen nicht unterstützt werden. Die Spezifikationen sind daher nur als Kommentare angegeben.

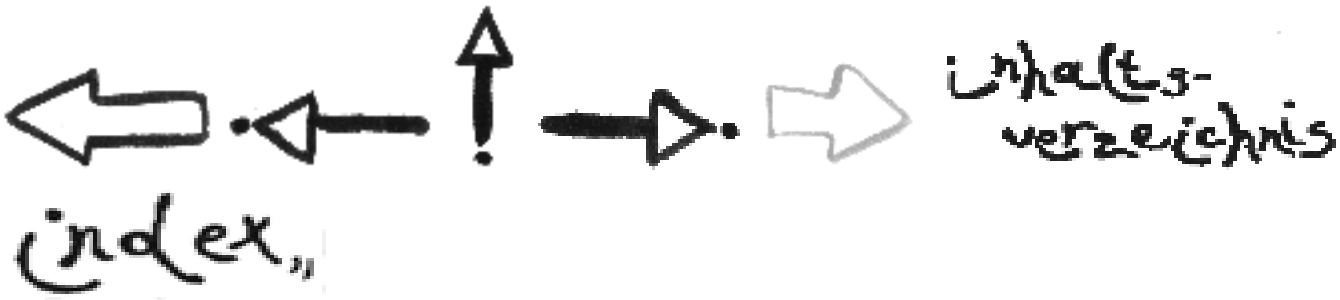
In bezug auf die Ausnahmen der Klasse `TString` ist folgendes anzumerken:

- Zwei verschiedene Arten von Ausnahmen können auftreten: `bad_alloc` und `range_error`.
- `bad_alloc` tritt immer dann auf, wenn eine Speicheranforderung fehlschlägt. Bisher wurde diese Situation in der Klasse `TString` vollständig ignoriert.

  `bad_alloc` wird von `new` ausgelöst. Zu beachten ist aber, daß damit alle Element-Funktionen, die Speicher anfordern (zum Beispiel die Konstruktoren von `TString`) diese Ausnahme indirekt auslösen (Ausnahmespezifikation!).
- Die Ausnahme `range_error` wird immer dann ausgelöst, wenn eine Bereichsverletzung festgestellt wird. Dies kann in den zwei Indexoperatoren und im ,,*Substring*``-Operator geschehen.

Um einen aussagekräftigen Fehlertext in den Ausnahmeobjekten ablegen zu können, werden relativ lange Fehlertexte generiert. Dazu wird eine eigene interne Funktion (`sampleRangeStr`) verwendet. Zu beachten ist in diesem Zusammenhang, daß die Fehlertexte in den Ausnahme-Klassen abgelegt werden, die entsprechenden Destruktoren diese aber nicht freigeben!

Vorige Seite: D.6 Ausnahmen **Eine Ebene höher:** D.6 Ausnahmen **Nächste Seite:** D.6.2 Die Klasse `Rational` II



Vorige Seite: D.6.1 Klasse `TString` II **Eine Ebene höher:** D.6 Ausnahmen

D.6.2 Die Klasse `Rational` II

Die Lösung der Aufgabe umfaßt folgende Dateien:

- Allgemeine ,,Hilfsdateien`` (`fakeiso.h`, `globs.h`)
- Die ,,Hilfsdatei`` für die Standard-Ausnahmen (`fakeexc.h`)
- Die Klasse `TString` (`tstring.h`, `tstring.cpp`)
- Die Klasse `Rational` (`rational.h`, `rational.cpp`)
- Das Testprogramm (`rattst.cpp`)

In bezug auf die Klasse `Rational` gelten die selben Einschränkungen wie auf die Klasse `TString` aus Abschnitt D.6.1. Auch hier gilt, daß ,,eigene`` Ausnahme-Klassen verwendet werden und Ausnahmespezifikationen durch Kommentare ,,simuliert`` sind.

Zur Implementierung der Klasse `Rational` ist folgendes anzumerken:

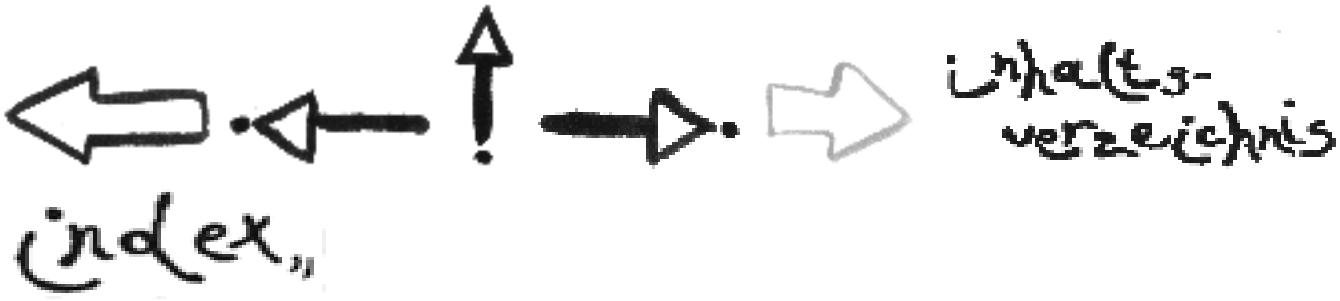
- Die Generierung der Fehlerinformationstexte erfolgt über eine eigene Hilfsfunktion (`sampleExcTxt`).
- Die internen Hilfsfunktionen (wie zum Beispiel `multiply` oder `negate`) können Bereichsverletzungen verursachen. In diesem Fall werden einfache `range_error`-Ausnahmen ohne spezielle Fehlertexte generiert.

  Die aufrufenden Funktionen behandeln die von den internen Funktionen generierten Außnahmen und reichen ,,neue`` Ausnahmeobjekte, die mit entsprechenden Ausnahmetexten versehen sind, an den Aufrufer weiter.
- Die Methoden der Klasse `Rational` können in erster Linie Bereichsverletzungen verursachen. Diese werden durch Objekte der Klasse `range_error` dargestellt.

  Indirekt können beim Aufruf von `Rational`-Methoden auch Ausnahmen der Klasse `TString` auftreten, da Fehlerinformationen teilweise über Methoden der Klasse `TString` generiert werden.

Um einen Eindruck von der Vereinfachung des Codes bei der Verwendung von Ausnahmen zu erhalten, kann die Klasse `Rational` mit jener aus Abschnitt D.2.6 verglichen werden. Im Vergleich zur ,,herkömmlichen`` Lösung ist der normale Code weitgehend frei von Fehlerabfragen und -behandlungen

und damit besser lesbar.



Vorige Seite: D.6.1 Klasse `TString` II **Eine Ebene höher:** D.6 Ausnahmen

Vorige Seite: D.6.2 Die Klasse `Rational` II **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** F Programmverzeichnis

E C++-Literatur

Im folgenden wird ein kurzer und subjektiver Überblick über einige wesentliche C++-Bücher gegeben. Tabelle E.1 führt einige Referenzen zur Programmiersprache C++ an sich an, Tabelle E.2 ergänzt die Liste um weiterführende Themenbereiche.

Referenz	Kommentar
[ES90]	*Das* C++-Buch, auch als *ARM* bezeichnet, teilweise überholt und für Anfänger etwas schwer verständlich, aber wohl das umfassendste C++-Buch und unerläßlich für ,,echte`` C++-Programmierer
[HN93]	Programmierrichtlinien für C++, empfehlenswert
[ISO95]	Draft ISO-Papier zum kommenden ISO-Standard, eine der Grundlagen für dieses Buch
[Lip91]	Gute und detaillierte Einführung, sehr empfehlenswert
[Pap95]	Einführung in die wesentlichen Konzepte von C++, setzt keine C-Kenntnisse voraus
[SB95]	Kurze und einfache Einführung in die wesentlichen Konzepte von C++, setzt C-Kenntnisse voraus
[Str92]	Eine ausführliche Darlegung der Sprache C++, leichter verständlich als das ARM, sehr empfehlenswert

[Str94]	Eines der wichtigsten C++-Bücher, beschreibt Entstehung der Sprache bis hin zum ISO-Standard und klärt über Hintergründe auf, sehr empfehlenswert

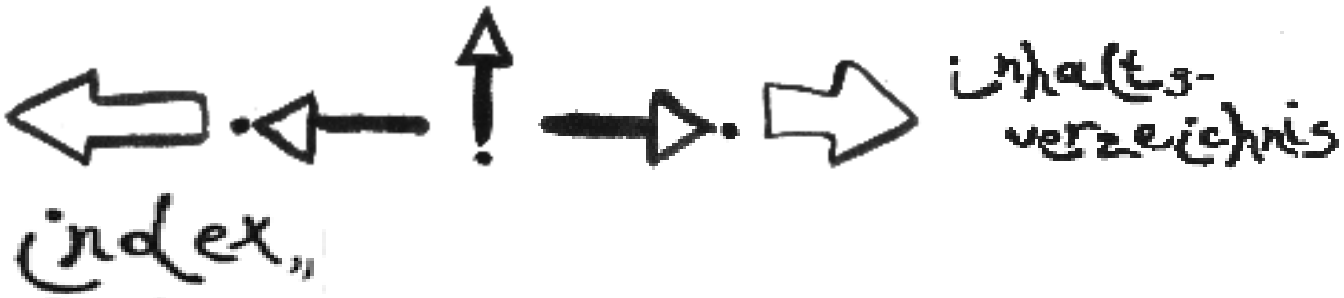
Tabelle E.1: C++-Literatur: Einführung & Referenz

Referenz	Kommentar
[Alg95]	Weiterführendes Buch, das Themen wie *Master Pointer*, Speichermanagement, *Scavenging* etc. behandelt
[Boo94]	Eines der wesentlichen Bücher zu den Grundlagen des objektorientierten Paradigmas
[CE95]	Gutes C++-Design Buch
[Car92]	Empfehlenswertes Buch mit vielen Tricks, zeigt Fehler und Schwächen anhand von konkreten Source-beispielen auf, gute Beispiele für Code-Inspektion
[Cop92]	*Das* (vielleicht einzige wirkliche) C++-Buch für Fortgeschrittene, sehr empfehlenswert
[GHJV95]	*Das* Buch zu Design Patterns, kein C++-spezifisches Buch, aber unerläßlich für weiterführende Arbeiten
[Hit92]	Gutes deutschsprachiges C++-Buch
[Lak95]	Umfangreiche und gute Sammlung von Tips und Richtlinien zur Abwicklung großer C++-Programme
[Mey92]	50 „Tips`` zu verschiedenen C++-Themen, eines der wichtigsten C++-Bücher, sehr empfehlenswert
[Mey96]	Weiterführung von [Mey92], sehr empfehlenswert

[Mur93]	Setzt einfache C++ Grundkenntnisse voraus, behandelt weiterführende Themen wie Handles, Ausnahmebehandlung etc., sehr empfehlenswert

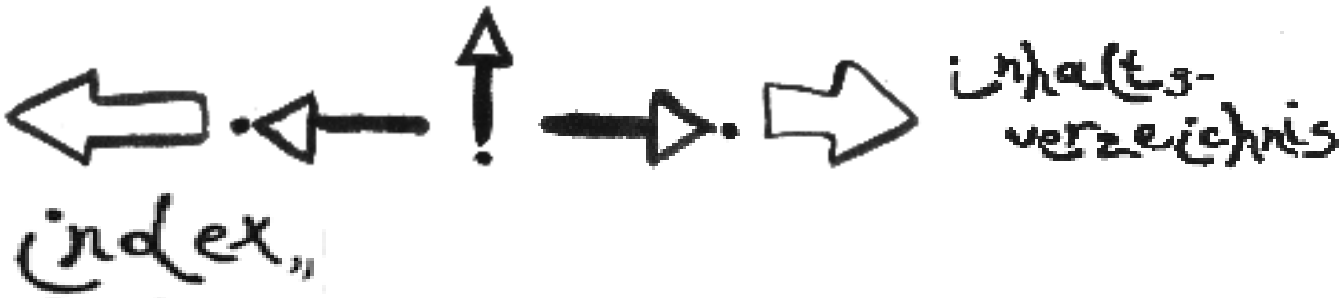
Tabelle E.2: Weiterführende C++-Literatur

Vorige Seite: D.6.2 Die Klasse `Rational` II **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** F Programmverzeichnis



Vorige Seite: E C++-Literatur **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** Literatur

F Programmverzeichnis



Vorige Seite: F Programmverzeichnis **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** Index

Literatur

Alg95

Jeff Alger. *Secrets of the C++ Masters*. Academic Press, London, 1995.

Boo87

Grady Booch. *Software Components with Ada: Structures, Tools, and Subsystems*. Benjamin/Cummings, Menlo Park, CA, 1987.

Boo94

Grady Booch. *Object-Oriented Analysis And Design with Applications*. Benjamin/Cummings, Redwood City, CA, 1994.

Car92

Tom Cargill. *C++ Programming Style*. Addison-Wesley, Reading, MA, 1992.

Car94

Tom Cargill. Exception handling: A false sense of security. *C++ Report*, 6(9), nov 1994.

CE95

Martin Carroll and Margaret Ellis. *Designing and Coding Reusable C++*. Addison-Wesley, Reading, MA, 1995.

Cla93

Ute Claussen. *Objektorientiertes Programmieren*. Springer-Verlag, Berlin, 1993.

Coh86

N. Cohen. *Ada as a Second Language*. McGraw-Hill, 1986.

Cop92

James Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, Reading, MA, 1992.

Dij68

Edsger Dijkstra. Goto statement considered harmful. *Communications of the ACM*, 11:147f, 1968.

ES90

Margaret Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

GHJV95

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

Goo73

Gerhard Goos. Hierarchies. In Friedrich Bauer, editor, *Advanced Course in Software Engineering*. Springer, Berlin, 1973.

GR89

Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley, Reading, MA, 1989.

Hit92

Werner Hitz. *C++ Grundlagen und Programmierung*. Springer-Verlag, Wien, 1992.

HN93

Mats Henricson and Erik Nyquist. Programming in c++: Rules and recommendations. Technical Report M 90 0118 Uen, Ellemtel Telecommunication System Laboratories, 125 25 Älvsjö, Schweden, 1993.

IBM94

IBM Corp. *SOMobjects Developers Toolkit 2.1, Programmers Reference Manual*, 1994.

ISO95

ISO. *Working Paper for Draft Proposed International Standard for Information Systems - Programming Language C++*. Number X3J16/95-0087, WG21/NO0687. 1995. ftp://research.att.com/dist/c++std/WP.

KR78

B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, 1978.

Lak95

John Lakos. *Large-Scale C++ Software Design*. Addison-Wesley, Reading, MA, 1995.

Lib93

Don Libes. *Obfuscated C and Other Mysteries*. John Wiley & Sons, New York, 1993.

Lip91

Stanley Lippman. *C++ Primer*. Addison-Wesley, Reading, MA, 2 edition, 1991.

Lip92

Stanley Lippman. *C++ Lernen und Beherrschen*. Addison-Wesley, Reading, MA, 2 edition, 1992.

LRM83

Department of Defense, USA, Washington. *Reference Manual for the Ada Programming Language (Proposed Standard Document)*, 1983.

Mey88

Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, 1988.

Mey92

Scott Meyers. *Effektive C++*. Addison-Wesley, Reading, MA, 1992.

Mey96

Scott Meyers. *More Effective C++*. Addison-Wesley, Reading, MA, 1996.

MS96

David Musser and Atul Saini. *STL Tutorial and Reference Guide : C++ Programming with the Standard Template library*. Addison-Wesley, Reading, MA, 1996.

Mur93

Robert Murray. *C++ Strategies and Tactics*. Addison-Wesley, Reading, MA, 1993.

Pap95

David Papurt. *Inside the Object Model: The Sensible Use of C++*. SIGS Books, New York, 1995.

PB96

G. Pomberger and G. Blaschek. *Software Engineering: Prototyping und objektorientierte Software-Enwicklung*. Carl Hanser Verlag, München, 2 edition, 1996.

Ree96

Jack Reeves. Ten guidelines for exception specification. *C++ Report*, 1:64ff, jul 1996.

SB95

Gregory Satir and Doug Brown. *C++: The Core Language*. O'Reilly, Sebastopol, CA, 1995.

Sch92

Arno Schulz. *Software-Entwurf: Methoden und Werkzeuge*. Oldenbourg Verlag, München, 3 edition, 1992.

Sed92

Robert Sedgewick. *Algorithmen in C++*. Addison-Wesley, Bonn, 1992.

Str92

B. Stroustrup. *Die C++-Programmiersprache*. Addison-Wesley, Bonn, 1992.

Str94

B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Reading, MA, 1994.

Vorige Seite: Literatur **Eine Ebene höher:** C++ Programmieren mit Stil **Nächste Seite:** Über dieses Dokument ...

Index

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Vorige Seite: Index **Eine Ebene höher:** C++ Programmieren mit Stil

Über dieses Dokument ...

Einige generelle Informationen zur Erstellung dieses Dokuments mit LaTeX2HTML:

> This document was generated using the **LaTeX**2HTML translator Version 97.1 (January 15th, 1997)
>
>
>
> The command line arguments were:
> **latex2html** `-split 5 -no_reuse0 -t C++ Programmieren mit Stil -html_version 3.0 cppbk41`.
>
> The translation was initiated by root on 3/1/1997
