TECHNISCHES PROGRAMMIEREN IN C++

Karl Riedling

Institut für Sensor- und Aktuatorsysteme,
TU Wien

7. Auflage
Wien, Frühjahr 2014

ISBN: 3-901578-05-6

Vorwort

Die objektorientierte Programmierung ist in der Entwicklung der modernen Software, sei es für graphische Benutzeroberflächen oder für komplexe Datenbanksysteme, nicht mehr wegzudenken. Unter den zahlreichen dafür verfügbaren Programmiersprachen ist C++ wahrscheinlich die am weitesten verbreitete, zumal es die Vorteile der Hardware-Nähe, die es von seinem Vorgänger, C, geerbt hat, mit den problemorientierten Funktionen einer objektorientierten Sprache verbindet. Die Lehrveranstaltung "Technisches Programmieren in C++" unternimmt den Versuch, Einblicke in diese — eben wegen ihrer Verbindung von maschinennahen und objektorientierten Eigenschaften — zwangsläufig komplexe und umfangreiche Sprache zu vermitteln.

Die vorliegenden Unterlagen enthalten eine vollständige Beschreibung von C++, also auch jene Sprachfunktionen, die unverändert aus C übernommen wurden. Um Lesern mit C-Vorkenntnissen das Studium dieser Unterlagen zu erleichtern, wurden jene Stellen und Abschnitte speziell gekennzeichnet, an denen in C++ neu eingeführte Funktionen oder Unterschiede zu Standard-C beschrieben werden. Wenn das Icon "C++-spezifisch" neben einer Kapitelüberschrift erscheint, ist das gesamte Kapitel als C++-spezifisch zu betrachten; erscheint es neben einem Absatz im fortlaufenden Text, so beinhaltet nur dieser Absatz Informationen, die für C++ neu sind.

C++-spezifisch

Die vorliegenden Unterlagen enthalten auch zahlreiche kurze Demo-Programme, die in erster Linie zur Illustration verschiedener Sprach-Funktionen dienen sollen und daher vielfach nicht als programmiertechnisch optimale Lösungen zu betrachten sind. Die meisten dieser Programme sind — zum Teil geringfügig erweitert und ausführbar gemacht — auch in elektronischer Form verfügbar; ein Icon am Beginn des Programmlistings gibt den Namen der Programmdatei an.

Demo-Programm 2_04_01.cc

Aus didaktischen und urheberrechtlichen Gründen sollen Programme im Zuge der Lehrveranstaltung ausschließlich mit Hilfe eines als *Freeware* verfügbaren Compilers, des GNU C/C++-Compilers, erstellt werden. Spezifische Eigenschaften (und *Bugs*) dieses Compilers werden mit dem Icon "CNU C/C++-spezifisch" gekennzeichnet.

GNU-C/C++-spezifisch

Die vorliegenden Unterlagen werden von einem allgemeinen Abschnitt über die Konzepte, die C++ zugrunde liegen eingeleitet. Darauf folgt ein Kapitel über Konventionen, wie Namen, reservierte Schlüsselworte, Operatoren und die Darstellung von Konstanten, sowie die Definition einiger in der Folge benötigter Begriffe. Im nächsten Abschnitt werden die zahlreichen von C++ unterstützten Daten-Typen kurz beschrieben; darauf folgt ein Überblick über Ausdrücke, Funktionen, Operatoren und Befehle in C++. Die Beschreibung der Datenstrukturen und ihrer Anwendungsmöglichkeiten wird im nächsten Abschnitt über die objektorientierte Programmierung vertieft; eine Aufteilung auf zwei getrennte Kapitel (3. und 5.) war notwendig, um die Lesbarkeit, insbesondere auch der Demo-Programme, nicht durch eine übergroße Anzahl von Vorwärtsreferenzen zu erschweren. Abschnitte über die Ein-/Ausgabefunktionen in C und C++ sowie über den Präprozessor und ein Literaturverzeichnis schließen diese Unterlagen ab.

Aus Platzgründen konnte leider keine vollständige Referenz der zahlreichen Bibliotheksfunktionen von C und C++ in diesen Unterlagen vorgesehen werden. Eine sehr gute Zusammenstellung der Standard-Bibliotheksfunktionen ist jedoch als Hypertext-On-Line-Dokumentation im GNU C/C++-Softwarepaket enthalten.

Typographische Konventionen

Schriftbild:	**Erklärung:**	**Beispiel:**
Normal	Fortlaufender Text	
Kursiv	(Englische) Fachausdrücke; Hervorhebungen	*Statements*
Fett, ***Kursiv/Fett***	Hervorhebungen	***Vorsicht:***
`Maschinenschrift`	Programm-Code; Namen von Variablen und Objekten; Ein- und Ausgabedaten	`int i = 0;` `class PUNKT`
`Kursiv`	Platzhalter (der durch einen geeigneten Namen oder Befehl zu ersetzen ist)	`for (`*`Init`*`; `*`Testen`*`; `*`Weiter`*`)`
`Fett`	Reserviertes C/C++-Schlüsselwort oder Operator (nicht in Programm-Beispielen); Hervorhebungen	**`int`** **`sizeof(`*`type`*`)`** (z.B. **`sizeof(int)`**)
`Kursiv/Fett`	Platzhalter für Schlüsselwort oder Operator	***`op`*`=`** (z.B. **`+=`**)

Diese Unterlagen wurden unter Verwendung von Microsoft *Word* erstellt; die verwendeten Schriftarten sind *Lucida Bright* (Text), *Lucida Sans* (Titel) und *Lucida Sans Typewriter* (Programm-Beispiele).

Inhalt

Demo-Programme

1. Das Konzept von C++

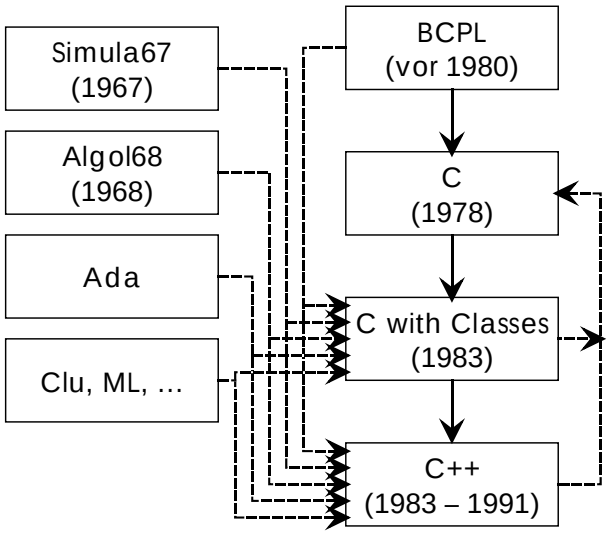
Dieser Abschnitt beschreibt die Geschichte der Entwicklung von C++ und die Konzepte, die hinter dieser Entwicklung stehen, in erster Linie das der objektorientierten Programmierung.

Anschließend werden allgemeine Aspekte für die Gestaltung technischer Programme diskutiert, die unabhängig von der für die Implementierung gewählten Programmiersprache gelten, aber unter Verwendung der von C++ gebotenen Möglichkeiten sehr effizient realisiert werden können.

1.1. Allgemeines

1.1.1. Die Entstehung von C++

C++ (ausgesprochen "*C plus plus*") ist eine Weiterentwicklung von C unter grundsätzlicher Beibehaltung der Funktionen von C als Teilmenge der Funktionen von C++. Die Wurzeln von C++ reichen bis in die Sechziger Jahre zurück; seine Entwicklungsgeschichte wird durch die folgende Graphik veranschaulicht:



Ebenso wie C wurde auch C++ bei AT&T entwickelt; als "Erfinder" von C++ gilt Bjarne STROUSTRUP. Als Basis von C++ wurde C gewählt, weil es

- flexibel, kompakt und verhältnismäßig hardwarenahe,
- auch für Systemprogrammierung geeignet,
- portabel (im Hinblick auf Hardware und Betriebssystem), und
- kompatibel mit der Programmierumgebung von UNIX

ist.

Die Abwärtskompatibilität von C++ zu C (also der Umstand, dass (fast) alle Funktionen und die Syntax von C auch von C++ in gleicher Weise unterstützt werden) bringt es mit sich, dass eine *Migration* von Programmen von C nach C++ in der Regel ohne größere Schwierigkeiten möglich ist. *De facto* gilt (nach STROUSTRUP) die Regel:

"Gute C-Programme **sind** C++-Programme"

Allerdings bietet C++ eine Fülle an Möglichkeiten, insbesondere an Datentypen, die im Vergleich zu C neu sind, und die viele Tricks ersetzen, die in C notwendig waren (z.B. *Type Casting*, Makros, *Unions*, Zeiger-Arithmetik,...). Grundsätzlich könnten alle Aufgabenstellungen, die mit C++ behandelt werden können, auch in C (oder auch in Assembler) programmiert werden; die neuen Features von C++ tragen aber zu einer erheblichen Erleichterung der Programmierarbeit und zu einer Reduktion von Fehlerquellen bei. (Frühe C++-Compiler verwendeten den Präprozessor eines C-Compilers (siehe Seite 10 und 215ff), um den C++-Quellcode in C-Quellcode umzusetzen, der anschließend wie ein gewöhnliches C-Programm übersetzt wurde. Jeder Compiler generiert wiederum letzten Endes eine Sequenz von binären Maschinenbefehlen, die ebenso gut, wenn auch um vieles mühsamer als durch das Hochsprachen-Programm, auch von einem menschlichen Programmierer in Assemblersprache erstellt werden könnte. Aus diesen Gründen ist die Aussage gerechtfertigt, dass *alles*, was (beispielsweise) in C++ programmiert werden kann, grundsätzlich, aber mit weit geringerem Komfort, ebenso auch in C oder Assembler codiert werden könnte.)

Die Entwicklung, die letztlich in der Definition von C++ resultierte, wirkte ihrerseits auch auf Standard-C zurück. In das heute standardisierte ANSI-C wurden Features von *C with Classes* und C++ aufgenommen, z.B. Funktions-Prototypen (siehe Seite 33 und 71).

1.1.2. Die Philosophie von C++

Dank seiner "Abstammung" von C verbindet C++ zwei ansonsten widersprüchliche Anforderungen:

- Maschinennähe durch C-spezifische Features;
- Problemnähe durch die neuen C++-spezifischen Eigenschaften.

Neu und zentral in der Philosophie von C++ ist die *objektorientierte Programmierung*, also die Abstraktion von Daten-*Objekten* in *Klassen* (*Classes*), in denen Datenstrukturen und die spezifischen Funktionen zu ihrer Behandlung zusammengefasst werden und die beliebig *hierarchisch* aufeinander aufgebaut sein können. Klassen, die von einer oder mehreren existierenden "Basisklassen" abgeleitet wurden, *erben* deren Eigenschaften und Datenstrukturen; damit wird ein konsistenter Zugriff auf Objekte unterschiedlicher Typen ermöglicht. In den meisten C++-Implementierungen ist eine weitere Abstraktion unter Verwendung von *Schablonen* (*Templates*) für verwandte Klassen möglich. Die Definition einer Klasse legt die *Struktur* und allenfalls die *Behandlung* der Objekte fest, die dieser Klasse angehören; als *Objekte* werden individuelle, voneinander unterscheidbare Agglomerate von Daten bezeichnet. Ein wesentlicher Aspekt dabei ist die *Einkapselung* von Daten und Funktionen, die auch verwendet werden können, ohne dass alle Details ihrer Implementierung bekannt zu sein brauchen.

Für die Definition von Klassen gelten die folgenden Grundregeln (nach B. STROUSTRUP):

- Wenn "*etwas*" als eigenständige Idee erscheint, definiere es als *Klasse*.
- Wenn "*etwas*" eine eigenständige Einheit darstellt, definiere es als *Objekt* einer Klasse.
- Wenn zwei Klassen "*etwas*" gemeinsam haben, mache es zu einer Basisklasse für beide.
- Tunlichst vermieden werden sollten:
  - Globale Daten;
  - Globale Funktionen;
  - Daten einer Klasse, die als **`public`** deklariert sind;
  - Direkte Zugriffe auf Daten anderer Objekte.

Ähnlich wie C oder Pascal ist C++ *funktionsorientiert*, d.h., der größte Teil der Funktionalität des Programms wird über (Bibliotheks-) Funktionen erfüllt, die vom Programm aus aufgerufen werden. Es existieren in C/C++ relativ wenige *Schlüsselworte* (*Keywords*), die fester Teil der Sprache sind (und daher nicht als Namen für Variable oder benutzerdefinierte Funktionen verwendet werden dürfen) (siehe Seite 13). (Im Gegensatz dazu hat beispielsweise BASIC eine große Anzahl von Schlüsselworten und wenige Funktionen.)

1.2. Programmier-Philosophie

Ein Programm sollte (mit abnehmender Priorität) die folgenden Anforderungen erfüllen:

➡ Funktionalität:

Die geforderten Funktionen (die zweckmäßigerweise *vor* Beginn der Programmierarbeit in einem Pflichtenheft definiert werden sollten) müssen einwandfrei und ohne unerwünschte oder im Widerspruch zur Definition stehende Nebenwirkungen erfüllt werden.

➡ Wartbarkeit:

Der Programmcode muss mit vertretbarem Aufwand erweiterbar (auf einen vergrößerten Funktionsumfang) oder korrigierbar (zur Behebung von logischen Programmfehlern) sein. Dies erfordert:

- Modularen Aufbau: Programmfunktionen müssen sinnvoll und nachvollziehbar auf *Unterprogramme* (in C++: *Unterprogramm = Funktion*) und die Funktionen auf *Module* (*Modul, Übersetzungseinheit = Datei*) aufgeteilt werden.
- *Einkapselung* von Funktionen: Jede Funktion hat *einen* exakt definierten Satz von Ein- und Ausgabedaten und *eine* exakt definierte Aufgabe. *Wie* diese Aufgabe im Detail erfüllt wird, hat den aufrufenden Funktionen gleichgültig zu sein. (Die Konsequenz dieser Forderung ist, dass beispielsweise von der Verwendung globaler Variablen abgesehen, Eingabedaten einer Funktion prinzipiell als Funktionsargumente übergeben, und Nebenwirkungen einer Funktion grundsätzlich nicht ausgenützt werden sollten.)
- Zusammenfassen zusammengehöriger Daten in Datenstrukturen (*Objekten*).
- Mnemotechnisch sinnvolle beschreibende Namen für Variable, Objekte und Funktionen.
- Großzügige Verwendung von sinnvollen *Kommentaren*: Es ist sinnlos, im Kommentar im Klartext zu wiederholen, was der Programmcode tut (z.B.: "*inkrementiere i*"); vielmehr sollte angegeben werden, was der *Zweck* der jeweiligen Operation ist (z.B.: "*nächstes Feldelement*").
- Vermeidung von undurchsichtigen Programmkonstruktionen: Konstrukte, die beispielsweise in einer Instruktion einen Zeiger inkrementieren, dereferenzieren und den Wert der Variablen testen, auf die der Zeiger zeigt, sind speziell in C/C++ möglich und werden häufig von "Profis" verwendet. Eine Aufteilung auf aufeinanderfolgende Instruktionen ist jedoch
  - übersichtlicher,
  - in ihren Wirkungen und Nebenwirkungen besser abschätzbar, und
  - führt im allgemeinen zum gleichen oder nicht signifikant längerem oder langsamerem *Object-Code*.
- Optische Gliederung des Programmcodes: Die Lesbarkeit eines Programms wird durch eine optische Gliederung deutlich verbessert, beispielsweise durch Einrückungen für jeden neuen Block, Zusammenfassung zusammengehöriger Instruktionen durch Einführung von "Absätzen" (= Leerzeilen), usw.

➡ Benutzerfreundlichkeit:

Selbst bei "kleinen" Programmen sollte die Benutzerschnittstelle logisch aufgebaut und intuitiv benutzbar gemacht werden. Die Benutzbarkeit wird durch eine On-Line-Hilfe immer verbessert. Grundsätzlich muss der Benutzer stets das Gefühl haben, zu wissen, *was* das Programm gerade tut oder beim nächsten Befehl tun wird. Kryptische Meldungen ("*Alle Dateien nicht löschen (J/N)?*") sind zu vermeiden. Um die Übersichtlichkeit der Benutzeroberfläche zu verbessern, sollten nicht regelmäßig benötigte Funktionen (leicht auffindbar) "versteckt" werden. Zur Vermeidung von Bedienungsfehlern (und zur Erleichterung der Bedienung des Programms) sollte die Eingabe von Befehlen und Daten (außer bei einfachen Hilfsprogrammen) grundsätzlich über Menüs erfolgen, wobei alle Optionen (zumindest alle nahe verwandten) gleichzeitig sichtbar sein sollten. Falsche numerische Eingaben oder Befehle sollten zur Ver-

meidung von Laufzeitfehlern abgefangen werden — das Programm darf auch von unfähigen oder bösartigen Benützern nicht zum Absturz gebracht werden können.

➡ Leistungsfähigkeit:

Für viele Anwendungen ist die Geschwindigkeit des Programmcodes *nicht* signifikant (vor allem dann, wenn die Eingabe von Befehlen und/oder Daten durch den Benützer die Geschwindigkeit bestimmt). Kritisch sind meist nur:

- Oft durchlaufene innere Programmschleifen;
- Hardware-naher Code (Interrupt-Handler).

Die Geschwindigkeit wird in der Regel durch den *Algorithmus* und *nicht* durch die verwendete Programmiersprache bestimmt; die Verwendung von Assembler-Code bringt daher nur in Ausnahmefällen etwas.

➡ Kompaktheit des Programmcodes:

Programme sollten möglichst so ausgelegt werden, dass die Größe der Programmdatei und der Arbeitsspeicherplatzbedarf des Programms selbst sich in Grenzen hält. (Dies gilt besonders für speicherresidente Programme, Gerätetreiber, ROM-residente Firmware u.ä.)

➡ Portabilität:

Wieweit die Portabilität des Programmcodes auf andere Betriebssystems- und/oder Hardware-Plattformen überhaupt ein berücksichtigenswürdiger Gesichtspunkt ist, hängt in erster Linie von der geforderten Funktion des Programms ab: Für hardwarenahen Code (z.B. Gerätetreiber) ist Portabilität kein Thema, für allgemein verwendbare Applikationen (z.B. Textverarbeitungsprogramme) schon. Ein hohes Maß an Portabilität des Programmcodes kann durch striktes Vermeiden von hardware- oder compilerspezifischen Erweiterungen der Programmiersprache und von "schmutzigen Tricks" aller Arten gewährleistet werden.

➡ "Schönheit" des Programmcodes:

Die Ästhetik des Programms wird von vielen Informatikern als das höchste Ziel der Programmierkunst gepriesen. "Schönheit" um ihrer selbst willen (wenn sie nicht zur Erreichung der zuvor genannten Punkte beiträgt) ist jedoch *sinnlos* und daher *in einem technischen Programm nicht anzustreben*. Die Konsequenz daraus ist:

- Bestimmte Programmiertechniken (z.B. objektorientiertes Programmieren oder rekursive Funktionsaufrufe) sind *dort und nur dort* sinnvoll, wo dies Vorteile im Hinblick auf die zuvor genannten Punkte bringt.
- Nicht das kürzeste, am "elegantesten" geschriebene Programm ist notwendigerweise auch das beste.
- In *Ausnahmefällen* (*nur* in Ausnahmefällen!) können auch "Verstöße" gegen die zuvor aufgestellten Regeln sinnvoll sein, um Anforderungen bezüglich Geschwindigkeit oder Speicherplatzbedarf (besser) erfüllen zu können (z.B. die Verwendung globaler Variablen, "Spaghetti-Code" oder ähnliches).

2. Konventionen

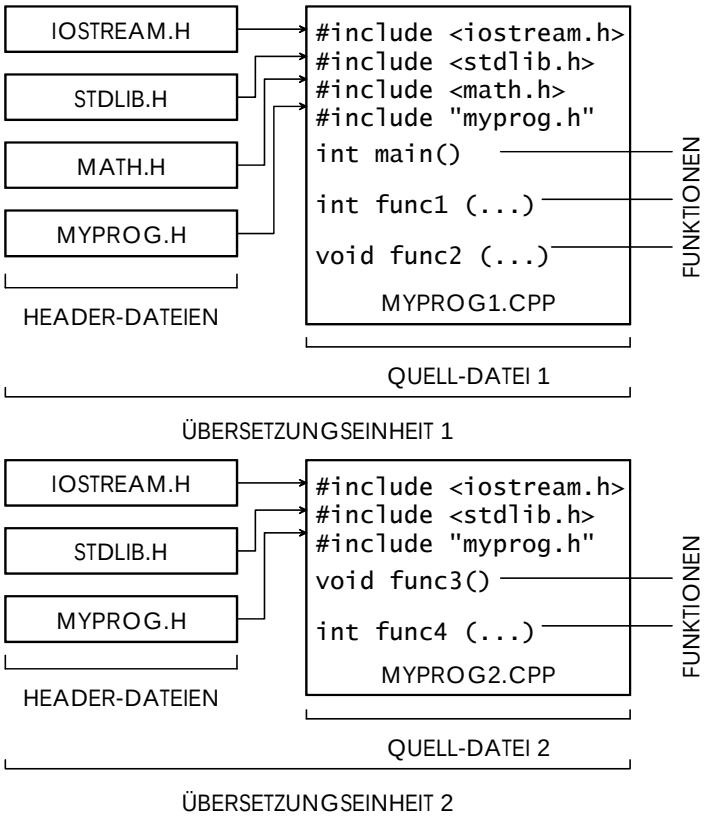
Diesen Abschnitt leitet eine Beschreibung der grundsätzlichen Struktur eines C++-Programms und der verschiedenen Phasen seiner Umsetzung in ausführbaren Code ein. Anschließend wird ein Überblick über die Elemente von C++ geboten, also über die Syntax der Namen von Funktionen und Daten-Objekten, reservierte Schlüsselworte, die Operatoren von C++ und die Struktur von Konstanten. Der Abschnitt wird mit einer Definition verschiedener in der Folge benötigter Begriffe abgeschlossen.

2.1. Struktur eines C++-Programms

Ein C++-Programm besteht aus einer oder mehreren *Übersetzungseinheiten*, die ihrerseits eine oder mehrere *Funktionen* enthalten können. Eine *Übersetzungseinheit* besteht im wesentlichen aus einer *Quell-Programmdatei*, in die aber (über **`#include`**-Präprozessordirektiven; siehe Seite 225) bei Bedarf *Header-Dateien* mit Funktionsdeklarationen, Definitionen von Konstanten oder Objekttypen usw. eingebettet werden können.

Für C++-Quell-Programmdateien haben sich die Dateinamenerweiterungen "`CC`", "`CPP`" oder "`CXX`" eingebürgert; für Header-Dateien zu C- oder C++-Programmen ist die Erweiterung "`H`" (wie "*Header*") üblich.

Jedes C++-Programm muss zumindest eine Funktion enthalten; genau eine der Funktionen des Programms muss den Namen `main` haben (mit Ausnahme von Programmen für *Microsoft Windows*; dort wird eine Funktion `WinMain()` erwartet). Die Funktion `main()` (oder `WinMain()`) ist diejenige, die nach den von den Laufzeit-Bibliotheksroutinen des C++-Compilers durchgeführten Initialisierungen aufgerufen wird; eine Rückkehr aus `main()` (oder `WinMain()`) entspricht einer Beendigung des Programms.



2.2. Übersetzung eines C++-Programms

Die Übersetzung von C++-Programmen in ausführbaren Code erfolgt in den folgenden Schritten:

- Umsetzung des Zeichensatzes des Quellprogramms in den internen Zeichensatz.
- Zusammenhängen von Fortsetzungszeilen (*Line Splicing*): Zeilen, in denen unmittelbar auf einen "\" ein Zeilenvorschub folgt, werden zusammengehängt.
- Umsetzung in *Tokens* (Ersetzen von Kommentaren durch Leerschritte; Aufteilung in Programm-Elemente und *White Space*).
- Vorverarbeitung (*Preprocessing*): Einbinden von Hilfsdateien über **`#include`**-Direktiven, Auflösung von Makros usw. mit einer "Übersetzungseinheit" als Ergebnis. Präprozessor-Befehle beginnen generell mit einem "**`#`**" als erstes Zeichen einer Zeile, das *kein White Space* (siehe Seite 11) ist.
- Umsetzung des Quell-Zeichensatzes in den Ausführungs-Zeichensatz.
- Zusammenhängen von Zeichenketten (*Strings*): Zeichenketten (unter doppelten Anführungszeichen), die nur durch *White Space* voneinander getrennt sind, werden zusammengefasst.
- Umsetzung in den *Object*-Code (mit syntaktischer und semantischer Prüfung des Quellcodes).
- Zusammenbinden verschiedener Objekt-Module und Bibliotheksfunktionen zu einem ausführbaren Programm.

2.3. Sprachelemente

2.3.1. *Tokens*

Tokens sind die kleinsten Elemente eines C++-Programms. *Tokens* können zu einer der folgenden Gruppen gehören:

- Namen (*Identifiers*);
- Schlüsselworte (*Keywords*);
- Numerische Konstanten und Textkonstanten (*Literals*);
- Operatoren;
- Trennzeichen.

Tokens werden durch Leerräume ("*White Space*") getrennt. Beliebige Kombinationen der folgenden Zeichen stellen Leerräume dar:

- Leerzeichen (*Spaces* — ASCII-Code `0x20` (20 hexadezimal = 32 dezimal));
- Horizontale und vertikale Tabulatoren (ASCII-Codes `0x09` und `0x0b`);
- Zeilenvorschübe (ASCII-Codes `0x0d` und `0x0a`);
- Seitenvorschübe (ASCII-Codes `0x0c`);
- Kommentare.

2.3.2. Kommentare

Kommentare werden vom Compiler (auch vom Präprozessor!) ignoriert. Sie dienen primär zur Erklärung des Programms für einen menschlichen Leser. Kommentare können auch verwendet werden, um Teile des Programmcodes temporär zu deaktivieren; das funktioniert aber dann nicht wunschgemäß, wenn der auskommentierte Programmteil selbst Kommentare enthält. (Günstiger ist für das Deaktivieren von Programmteilen die Verwendung von **`#if/#endif`**-Präprozessor-Direktiven; siehe Seite 226)

Es existieren in C++ (*nicht* in Standard-C) zwei verschiedene Formen von Kommentaren:

C++-spezifisch

- "Klassischer" ANSI-C-Stil: Der Kommentar wird zwischen "**`/*`**" und "**`*/`**" eingeklammert:

  ```
  /* Das ist ein Kommentar */
  ```

  ANSI-C-Kommentare dürfen sich über eine beliebige Zahl von Zeilen erstrecken; der Kommentar endet beim ersten "**`*/`**" nach dem einleitenden "**`/*`**" (deshalb sind keine geschachtelten Kommentare zulässig!).

- Einzelzeilen-Kommentare (*Single Line Comments*): Der Kommentar beginnt mit zwei unmittelbar aufeinanderfolgenden "**`//`**" und endet mit dem Zeilenende (sofern nicht *unmittelbar* vor dem Zeilenvorschubs-Zeichen ein "\" steht).

  ```
  // Das ist auch ein Kommentar
  ```

"**`/*`**", "**`*/`**" und "**`//`**" innerhalb von Zeichenketten werden nicht als Kommentareinleitung interpretiert.

2.3.3. Namen (*Identifiers*)

Die folgenden Sprachelemente werden mit *Namen* bezeichnet:

- *Objekte* oder *Variable*.
- *Klassen*, *Strukturen* und *Unionen*.
- *Aufzählungs-Typen*.
- Elemente einer Klasse, Struktur, Union oder Aufzählungs-Type.
- *Funktionen* (auch Klassenelement-Funktionen).
- Benutzerdefinierte Typen (**typedef**).
- Sprungmarken.
- *Makros*.
- Argumente von Makros.

Namen dürfen aus folgenden Zeichen bestehen:

`_ a b c … x y z A B C … X Y Z 0 1 2 … 8 9`

Sie dürfen jedoch *nicht* mit einer Ziffer (`0 … 9`) beginnen. Groß- und Kleinbuchstaben werden als *verschieden* betrachtet ("`DateiName`" ist nicht gleich "`Dateiname`"). Namen dürfen nicht identisch mit Schlüsselworten sein (sie dürfen sie aber enthalten — "`Pint`" oder "`Int`" ist ein legaler Name, "`int`" wäre es nicht, weil "**`int`**" ein reserviertes Schlüsselwort ist).

Die folgenden Konventionen haben sich in C/C++ eingebürgert (und wurden zum Teil in die ANSI-C-Norm übernommen):

- Namen von Variablen, Funktionen, und anderen vom *Compiler* aufzulösenden Einheiten sind in Kleinbuchstaben oder in einem Gemisch von Groß- und Kleinbuchstaben, in dem letztere überwiegen:

  `i; sin(x); DateiName; datei_name`

- Namen, die vom *Präprozessor* aufgelöst werden (also Namen von Makros, mit **`#define`** definierten Konstanten u.ä.) sind zur Gänze in Großbuchstaben:

  `PROGRAMM_VERSION; ZEILEN_PRO_SEITE`

Abkürzungen in "ungarischer Notation":

Abkürzung	Bedeutung
`c`	**`char`** (vorzeichenbehaftete 8-Bit-Zahl)
`by`	`BYTE` (**`unsigned char`**)
`n`	**`int, short`** (vorzeichenbehaftete 16-Bit-Zahl)
`i`	**`int`**
`x, y`	**`short`** (als Bildschirmkoordinate)
`cx, cy`	**`short`** (als Länge in x- oder y-Richtung)
`b`	`BOOL` (**`int`**)
`w`	`WORD` oder `UINT` (**`unsigned short`** oder **`unsigned int`**)
`l`	**`long`**
`dw`	`DWORD` (**`unsigned long`**)
`fn`	Funktion
`s`	*String*
`sz`	*String*, mit NULL-Byte abgeschlossen

- Gelegentlich wird die *Ungarische Notation* (nach dem legendären Microsoft-Programmierer Charles Simonyi; siehe vorige Seite) verwendet, bei der dem eigentlichen (Variablen-) Namen ein oder zwei Zeichen vorangestellt werden, die den Typ der Variablen (für den menschlichen Leser des Programms) andeuten, z.B.:

  `nZahl` – `n` für Integer.

  `szname` – `sz` für *String, Zero delimited* (Zeichenkette, die mit einem NULL-Byte abgeschlossen ist).

  `fnMyFunc` – `fn` für Funktion(s-Zeiger).

- Die Verwendung von "__" (zwei "_") oder einem "_", gefolgt von einem Großbuchstaben, am Anfang eines Namens ist für C++-Implementierungen reserviert. "_", gefolgt von einem Kleinbuchstaben, am Anfang eines Namens sollte für Namen vermieden werden, die über das aktuelle Modul hinaus gültig sind (z.B. globale Variable oder Funktionen), um Portabilitätsprobleme zu vermeiden.

2.3.4. Schlüsselworte (*Keywords*) in C++

Die folgenden Schlüsselworte sind in Standard-C++ reserviert und dürfen nicht als Namen für Objekte oder Funktionen verwendet werden (dazu kommen noch implementierungsspezifische Schlüsselworte):

C++-spezifisch

`asm`	`delete`	`if`	`return`	`try`
`auto`	`do`	`inline`	`short`	`typedef`
`break`	`double`	`int`	`signed`	`union`
`case`	`else`	`long`	`sizeof`	`unsigned`
`catch`	`enum`	`new`	`static`	`virtual`
`char`	`extern`	`operator`	`struct`	`void`
`class`	`float`	`private`	`switch`	`volatile`
`const`	`for`	`protected`	`template`	`while`
`continue`	`friend`	`public`	`this`	
`default`	`goto`	`register`	`throw`	

(C++-spezifische Schlüsselworte sind schattiert.)

2.3.5. Operatoren

Die folgende Tabelle listet die Operatoren von C++ mit abnehmender *Präzedenz* (siehe Seite 83) auf. Operatoren, die durch einen doppelten Rahmen in eine Gruppe zusammengefasst sind, haben jeweils gleiche Präzedenz.

Operator	Bedeutung	Beispiel	assoziiert
`::`	Gültigkeit	`Konto::Stand`	—
`::`	Globales Objekt	`::breite`	—
`[ ]`	Feldindex	`a[j]`	links → rechts
`( )`	Funktionsaufruf	`MyFunc(x)`	links → rechts
`( )`	Typkonversion	`unsigned long (i)`	—
`.`	Element-Auswahl (Objekt)	`time.day`	links → rechts
`->`	Element-Auswahl (Zeiger)	`timeptr->hour`	links → rechts
`++`	Postfix-Inkrement	`i++`	—
`--`	Postfix-Dekrement	`j--`	—

Operator	Bedeutung	Beispiel	assoziiert
new	Erstelle neues Objekt	`new window mywin`	—
delete	Zerstöre Objekt	`delete mywin`	—
delete[]	Zerstöre Objekt (Feld)	`delete[] myarray`	—
++	Präfix-Inkrement	`++i`	—
--	Präfix-Dekrement	`--i`	—
*	Dereferenzieren	`*buffer`	—
&	Adresse	`&i`	—
+	Unäres Plus	`+3`	—
-	Unäres Minus	`-k`	—
!	Logisches "Nicht"	`!(x > 3)` (= x ≤ 3)	—
~	Bitweises (Einer-) Komplement	`~0x8000` (= 0x7fff)	—
sizeof	Größe eines Objekts	`sizeof time`	—
sizeof()	Größe einer Type	`sizeof(long)` (= 4)	—
(*type***)**	*Type Cast*	`(int) 'A'`	rechts → links
.*	Zeiger auf ein Element einer Klasse auf Objekt anwenden	`Konto.*Stand`	links → rechts
->*	Zeiger auf ein Klassen-Element dereferenzieren	`Konto->*Stand`	links → rechts
*	Multiplikation	`a*b`	links → rechts
/	Division	`a/b`	links → rechts
%	Modulo	`a%b`	links → rechts
+	Addition	`a + b`	links → rechts
-	Subtraktion	`a - b`	links → rechts
<<	Verschieben nach links	`x << 3` (= x = x * 8)	links → rechts
>>	Verschieben nach rechts	`x >> 2` (= x = x/4)	links → rechts
<	kleiner als	`x < 3`	links → rechts
>	größer als	`x > 4`	links → rechts
<=	kleiner gleich	`y <= x`	links → rechts
>=	größer gleich	`y >= x`	links → rechts
==	gleich (im Vergleich)	`i == 1`	links → rechts
!=	ungleich	`j != 0`	links → rechts
&	bitweises UND	`i & 0x3f`	links → rechts
^	bitweises EX-OR	`j ^ 0x5a5a`	links → rechts
|	bitweises ODER	`j | 0x8000`	links → rechts
&&	logisches UND	`(x > 0)&&(y < 1)`	links → rechts
||	logisches ODER	`(x > 3)||(y > 3)`	links → rechts

Operator	Bedeutung	Beispiel	assoziiert
`? :`	bedingte Ausführung	`x > 0 ? x-- : i++`	rechts → links
`=`	Zuweisung	`x = 3`	rechts → links
`*=`	Multiplikations-Zuweisung	`x *= 3` (= x = x * 3)	rechts → links
`/=`	Divisions-Zuweisung	`y /= 4`	rechts → links
`%=`	Modulo-Zuweisung	`i %= 7`	rechts → links
`+=`	Additions-Zuweisung	`j += 2`	rechts → links
`-=`	Subtraktions-Zuweisung	`y -= 10`	rechts → links
`<<=`	Linksverschiebungs-Zuweisung	`i <<= 3`	rechts → links
`>>=`	Rechtsverschiebungs-Zuweisung	`x >>= 2`	rechts → links
`&=`	Zuweisung mit bitweisem UND	`i &= 0x3f`	rechts → links
`|=`	Zuweisung mit bitweisem ODER	`i |= 0x8000`	rechts → links
`^=`	Zuweisung mit bitweisem EX-OR	`j ^= 0x5a5a`	rechts → links
`,`	Komma	`x = 5*i, y = 3*i`	links → rechts

2.3.6. Befehle

Befehle (*Statements*) in C++ bestehen aus einem oder mehreren *Ausdrücken* (*Expressions*), die über Operatoren miteinander verknüpft sind. Befehle sind in der Regel immer mit einem Strichpunkt ("**;**") abgeschlossen:

```
a = b + c;

if (x > 0)
    i = 1;
else
    i = 0;

i = (x > 0) ? 1 : 0;              // äquivalent zum vorigen Beispiel
```

Eine beliebige Anzahl von Befehlen kann durch geschwungene Klammern ("**{ }**") zu einem "*Befehlsblock*" oder "*Verbundbefehl*" zusammengefasst werden, der so behandelt wird, als wäre er ein einziger Befehl. Befehle *innerhalb* eines Blocks sind jedoch *immer* mit Strichpunkten abzuschließen (auch der letzte!). Der "Körper" einer Funktion ist *immer* ein Block in geschwungenen Klammern, auch wenn er nur einen einzigen Befehl enthält:

```
if (x > 0)
{
    i = 1;
    j = -1;
}
else
{
    i = 0;
    k = 1;
}
```

Demo-Programm 2_03_01.cc

```
int faktorielle (unsigned n)
{
    int nfak;

    for (nfak = 1; n > 1; n--)
        nfak *= n;                      // nfak = nfak * n;

    return nfak;
}

// oder

int faktorielle (unsigned n)
{
    int nfak = 1;

    while (n > 1)
    {
        nfak *= n;
        n--;
    }

    return nfak;
}
```

2.3.7. Konstanten

2.3.7.1. Ganzzahlige (*Integer*) Konstanten

Integer-Konstanten beginnen grundsätzlich immer mit einer Ziffer. Folgende Formate werden unterstützt:

- Dezimale Konstanten: Beginnen mit einer der Ziffern zwischen 1 und 9 (*nicht* mit 0) und können alle Ziffern von 0 bis 9 enthalten.
- Oktale Konstanten: Beginnen immer mit "0" und enthalten nur Ziffern zwischen 0 und 7.
- Hexadezimale Konstanten: Beginnen mit "0x" oder "0X" und können neben den Ziffern von 0 bis 9 die hexadezimalen Ziffern "a" bis "f" (oder "A" bis "F") enthalten. (Hexadezimale Konstanten sind einer der ganz seltenen Fälle in C++, wo Groß- und Kleinbuchstaben äquivalent sind.)

Dezimal:	Oktal:	Hexadezimal:
123	0173	0x7b
32767	077777	0x7fff
2147483647	017777777777	0x7fffffff

C++-spezifisch

Optional kann durch ein nachgestelltes "u" oder "U" angedeutet werden, dass die Konstante als **unsigned** (nicht vorzeichenbehaftet) zu interpretieren ist. Konstanten, die als **long int**(eger) darzustellen sind, wird "l" oder "L" nachgestellt. "U" und "L" dürfen auch kombiniert werden. Grundsätzlich sollte der Typ einer Konstanten mit dem der Variablen übereinstimmen, der oder denen ihr Wert zugewiesen werden soll. C++ führt — im Gegensatz zu Standard-C — jedoch immer eine automatische Konversion in das Datenformat der Zielvariablen durch, wobei unter Umständen jedoch Vorzeichen und/oder Wert der Konstanten verändert werden können:

Das folgende Beispiel nimmt eine 16-Bit-Implementierung mit

- (*Short*) *Integer* = 16 Bit
- *Long Integer* = 32 Bit

an.

Damit ist der Wertebereich einer *Short Integer*: 2^{16} Werte —

- -32767 ... 32767 (*Signed Integer*) oder
- 0 ... 65535 (*Unsigned Integer*).

Die Zahl "40000" ist daher in dieser Implementierung keine gültige (vorzeichenbehaftete) (*Short*) *Integer*-Konstante, weil sie außerhalb des vorzeichenbehafteten Wertevorrats liegt, sie ist aber eine gültige *Unsigned Integer*-Konstante und daher als "40000u" oder "40000U" darzustellen. Sie könnte alternativ auch als 32-Bit *Long Integer* dargestellt werden: "40000l" oder "40000L".

```
int i = 40000U;        // gültig; der tatsächliche Wert von i ist aber -25536
                       // (40000 = 0x9c40 als Zweier-Komplement interpretiert)
unsigned j = 40000U;   // gültig
long k = 40000L;       // gültig
int l = 1234567L;      // gültig; der tatsächliche Wert von l ist aber -10617
                       // (1234567 = 0x12d687; die int-Variable wird 0xd687 =
                       // Zweier-Komplement von 10617)

unsigned long l = 4000000000UL;  // gültig
```

Ein häufiger Fehler ist eine versehentliche Interpretation einer dezimalen Konstanten als oktaler Wert. Ein derartiger Fehler kann nicht nur bei der Übersetzung eines Quellprogramms vorkommen, sondern auch bei der Ausführung des Programms, weil manche C/C++-Bibliotheksfunktionen für die Konversion von Eingabedaten die gleiche Notation für die Interpretation der Eingabedaten verwenden wie der Compiler. Wenn beispielsweise ein Uhrzeit-Wert vom Programm eingelesen und weiterbearbeitet werden soll, so kann dies typisch unter Verwendung des folgenden Funktionsaufrufs geschehen:

```
sscanf (buffer, "%i:%i:%i", &std, &min, &sec);
```

Die Funktion `sscanf` wird eine Zeitangabe in der Form von beispielsweise "11:47:32" problemlos auf die drei **int**-Variablen `std`, `min` und `sec` abbilden. Hingegen streikt die Umwandlung bei einer Eingabe von "08:15:00", weil "08" eine ungültige oktale Konstante ist (da ja nur die Ziffern von 0 bis 7 in einer oktalen Konstante zulässig sind). Wenn die standardmäßige Interpretation der Eingabedaten unterdrückt und diese *nur* als Dezimalzahlen interpretiert werden, indem anstelle der Formatangabe "%i" der Formatstring "%d" verwendet wird (siehe Seite 181), ist auch im zweiten Fall eine korrekte Interpretation möglich; die führende Null wird dann einfach ignoriert.

2.3.7.2. Zeichenkonstanten (*Character*-Konstanten)

In C++ wird unterschieden zwischen dem

- Quell-Zeichensatz (*Source Character Set*) — dem Zeichensatz, in dem das Programm geschrieben wurde —, und dem
- Ausführungs-Zeichensatz (*Execution Character Set*) — dem Zeichensatz, in dem das Programm mit seinen Benützern spricht.

Beide Zeichensätze brauchen nicht notwendigerweise identisch zu sein. Zeichenkonstanten gehören immer dem Quell-Zeichensatz an, stellen aber Zeichen des Ausführungs-Zeichensatzes dar.

Es existieren die folgenden Typen von Zeichenkonstanten:

- "Gewöhnliche" Zeichenkonstanten: Sie sind immer vom Typ **char**.

  ```
  char ch = 'a';
  ```

- Mehrfach-Zeichenkonstanten (*Multicharacter Constants*): Sie sind implementierungsabhängig und nicht portabel. Mehrfach-Zeichenkonstanten enthalten so viele Zeichen, wie eine Variable vom Typ **int** Variable vom Typ **char** enthält (`sizeof(int)`) (das sind in 16-Bit-Implementierungen 2, in 32-Bit-Implementierungen 4):

  ```
  int mbch = 'xy';
  ```

- "Weite" Zeichenkonstanten (*Wide Character Constants*) — z.B. *Unicode*: Sie haben den abgeleiteten Typ `wchar_t`:

  ```
  wchar_t unicode_char = L 'ab';
  ```

(Beachten Sie das Präfix "L" !)

Zeichenkonstanten können dargestellt werden durch:

- Ein oder mehrere Zeichen des Quell-Zeichensatzes (mit Ausnahme des einfachen Anführungszeichens ("'"), des Rückschrägers (*Backslash*) ("\") und des Zeilenvorschubs) in *einfachen* Anführungszeichen:

  ```
  'a', 'X', 'Ü'
  ```

- Einfache *Escape*-Sequenzen: Bestehen aus einem Backslash, gefolgt von einem anderen Zeichen:

Zeichen	**ASCII-Bezeichnung**	**ASCII-Wert**	**Escape-Sequenz**
Zeilenvorschub	NL oder LF	0x0a	\n
Horizontaler Tabulator	HT	0x09	\t
Vertikaler Tabulator	VT	0x0b	\v
Rückschritt	BS	0x08	\b
Zeilenrücklauf	CR	0x0d	\r
Seitenvorschub	FF	0x0c	\f
Alarm	BEL	0x07	\a
Null-Zeichen (*nicht* die Ziffer "0"!)	NUL	0x00	\0
Backslash	\	0x5c	\\
Fragezeichen	?	0x3f	\?
Einfaches Anführungszeichen	'	0x27	\'
Doppeltes Anführungszeichen	"	0x22	\"

- Oktale Zeichenkonstanten: Backslash, gefolgt von einer bis drei oktalen Ziffern:

  ```
  '\377'          // ASCII-Wert 255
  ```

- Hexadezimale Zeichenkonstanten: "\x", gefolgt von einer *beliebigen* Anzahl hexadezimaler Ziffern:

  ```
  '\xff'          // gleiche Konstante wie oben
  '\x00ff'        // gleiche Konstante
  ```

Oktale Konstanten enden mit dem ersten Zeichen, das keine oktale Zahl ist, oder nach spätestens drei Zeichen. Hexadezimale Konstanten enden mit dem ersten Zeichen, das keine hexadezimale Zahl ist.

Bei der Definition von String-Konstanten (siehe Seite 19) müssen allenfalls zur Vermeidung von Fehlinterpretationen Escape-Sequenzen getrennt in doppelten Anführungszeichen angeführt und der Mechanismus zum Zusammenhängen von Zeichenketten verwendet werden:

```
"\x7aha!"          // wird interpretiert: 0x7a (z)+ "ha!" = "zha!"
"\x7" "aha!"       // wird interpretiert: BEL + "aha!"
```

2.3.7.3. Gleitkommakonstanten

Gleitkommakonstanten müssen mindestens eine der folgenden Komponenten enthalten:

- Dezimalpunkt;
- Dezimalstellen;
- Exponent ("e" oder "E", gefolgt von einer ganzzahligen Konstanten).

Beispiele für Gleitkommakonstanten:

```
123                 // KEINE Gleitkommakonstante!
123.                // gültig
123.456             // gültig
0.456               // gültig
.456                // gültig
123e1               // gültig (= 1230.)
3.456e-9            // gültig (= 0.000000003456)
```

Gleitkommakonstanten haben standardmäßig den Typ **`double`**. Durch Nachstellen von "`f`" oder "`F`" können Konstanten vom Typ **`float`** angegeben werden, durch Nachstellen von "`l`" oder "`L`" Konstanten vom Typ **`long double`**.

2.3.7.4. Zeichenketten- (*String*-) Konstanten

Stringkonstanten bestehen aus einem oder mehreren Zeichen des Quellzeichensatzes in doppelten Anführungszeichen ("). Strings sind immer mit einem Null-Byte (NUL, `'\0'`) abgeschlossen.

Strings können dargestellt werden als

- Felder vom Typ **`char[`***n*`]`**, wobei *n* die Anzahl der Zeichen im String plus 1 (für das Null-Byte) ist:

  ```
  char msg[] = "Hello, world!";        // Feldgröße ist 14 (13 + 1)
  char msg[14] = "Hello, world!";      // Angabe der Feldgröße ist unnötig
  ```

- Felder vom Typ `wchar_t[`*n*`]`:

  ```
  wchar_t WideStr[] = L"\x1234\x2345"; // WideStr ist 6 Bytes lang (4+2)
  ```

Aneinanderstoßende Zeichenketten werden zusammengehängt:

```
"12" "34"                            // äquivalent zu "1234"
```

Das gilt auch, wenn die Strings durch Zeilenvorschübe getrennt sind:

```
"The quick brown fox jumps "
"over the lazy white dog."
```

Alternativ kann auch *Line Splicing* verwendet werden, um den gleichen Zweck zu erreichen:

```
"The quick brown fox jumps \
over the lazy white dog."
```

(Vorsicht: Im zweiten Fall werden alle Zeichen, die zur graphischen Gestaltung des Programmtextes verwendet werden (Tabulatoren, Einrückungen) als zum String gehörig betrachtet, während im ersten Fall nur die zwischen den doppelten Anführungszeichen stehenden Texte tatsächlich verwendet werden!)

Alle auf Seite 18 beschriebenen Escape-Sequenzen einschließlich oktaler oder hexadezimaler Konstanten dürfen in Strings verwendet werden.

2.4. Begriffe

2.4.1. Deklarationen und Definitionen

Deklarationen

Deklarationen führen *Namen* und *Typen* in einem Programm ein, ohne notwendigerweise ein zugehöriges Objekt oder eine Funktion zu erzeugen. (Vielfach sind aber Deklarationen gleichzeitig auch Definitionen.) Deklarationen, die *nicht* gleichzeitig auch Definitionen sind, dürfen auch öfter als einmal in einem Programm(modul) vorkommen, sofern sie einander nicht widersprechen (z.B.: Funktions-*Prototypen*, **`typedef`**-Befehle).

Definitionen

Definitionen enthalten Information, die es dem Compiler ermöglicht, Speicherplatz für *Objekte* zu reservieren oder Programmcode für *Funktionen* zu generieren. Ein Objekt oder eine Funktion muss genau *einmal* definiert werden (es gibt aber Ausnahmen, wo wohl eine Deklaration notwendig ist, aber keine Definition).

2.4.2. Gültigkeitsbereich (*Scope*)

Der Gültigkeitsbereich eines (Objekts- oder Funktions-) Namens gibt den Bereich des Programms an, innerhalb dessen auf dieses Objekt unter diesem Namen zugegriffen werden kann. Es gibt fünf Gültigkeitsbereiche:

- Lokale Gültigkeit: Namen, die *innerhalb* eines Blocks (d.h., innerhalb zweier zusammengehöriger geschweifter Klammern ("**`{ }`**")) deklariert werden, haben *lokale* Gültigkeit und können nur *innerhalb* dieses Blocks angesprochen werden. Dies gilt auch für die formalen Argumente von Funktionen, die so behandelt werden, als wären sie im äußersten Block der Funktion deklariert worden.
- Funktions-Gültigkeit: Sie existiert nur für Sprungmarken (*Labels*): Eine Sprungmarke kann von jedem Punkt einer Funktion aus angesprungen werden, aber nicht von außerhalb der Funktion.
- Datei-Gültigkeit: Namen, die *außerhalb* aller Blöcke oder Klassen deklariert wurden, haben Datei-Gültigkeit und können von jedem Punkt der Programmdatei aus angesprochen werden, der *nach* ihrer Deklaration liegt. (Wenn sie nicht *statische* Objekte angeben, werden Namen mit Datei-Gültigkeit oft auch als *globale* Namen bezeichnet.)

C++-spezifisch

- Klassen-Gültigkeit (*Class Scope*): Gilt für Namen der Elemente einer Klasse (Daten oder Funktionen), die nur mit Hilfe des Elementauswahl-Operators ("**`.`**", "**`->`**", "**`.*`**" oder "**`->*`**"), angewandt auf ein Objekt (oder einen Zeiger auf ein Objekt) dieser Klasse, angesprochen werden können. Beispiel:

```
class Point
{
    int x;
    int y;
};        // x und y haben Klassen-Gültigkeit.
```

- Prototypen-Gültigkeit: Gilt nur für die (optionalen) Namen in einem Funktions-Prototyp; die Gültigkeit endet mit dem Ende des Prototyps. Beispiel:

```
char *strcpy (char *szDest, const char *szSource);
          // szDest und szSource haben Prototypen-Gültigkeit.
```

Namen gelten unmittelbar *nach* ihrer Deklaration, aber noch *vor* ihrer allfälligen Initialisierung als deklariert.

Namen können *verborgen* werden, wenn sie innerhalb eines eingeschlossenen Blocks neu deklariert werden:

Demo-Programm 2_04_01.cc

```
#include <iostream.h>
    // benötigt für die Deklaration von cout (siehe unten)

int i = 1;                                      // hat Datei-Gültigkeit

void MyFunc (int i)                             // verbirgt obiges i
{
    cout << "i = " << i << "\n";
        // gibt den Wert des Arguments i aus (nicht unbedingt 1)
    {
        int i = 2;                              // verbirgt Argument i
        cout << "i = " << i << "\n";            // gibt aus "i = 2"
        {
            int i = 3;
            cout << "i = " << i << "\n";        // gibt aus "i = 3"
        }

        cout << "i = " << i << "\n";            // gibt aus "i = 2"
    }

    cout << "i = " << i << "\n";
        // gibt den Wert des Arguments i aus (nicht unbedingt 1)
}

void MyFunc1 ()
{
    cout << "i = " << i << "\n";
        // gibt den Wert der globalen Variable i aus ("i = 1")
}
```

In der hier demonstrierten Form ist dies aber ein Beispiel extrem schlechter Programmierpraxis!

C++-spezifisch

Das vorangegangene Beispiel verwendet `cout`, das in C++ als Standard-Ausgabe-*Stream* (siehe Seite 191 ff) vorgesehen ist. Die auszugebenden Daten werden verkettet durch den Operator "`<<`" hintereinander angegeben; der Compiler wählt automatisch das korrekte Ausgabeformat für jedes Datenelement. `cout` ist wohl mit der Sprache definiert, ist aber als Objekt einer Klasse (`ostream`) implementiert. (Die sonderbare Form des Aufrufs von `cout` ergibt sich aus der objektorientierten Implementierung.) Damit der Compiler den Namen `cout` und die für die Argumente von `cout` verwendete Syntax korrekt interpretieren kann, benötigt er Information aus einer *Header-Datei*, im konkreten Fall die Datei "`iostream.h`", die bei der Übersetzung des Programm-Moduls in die Quelldatei an der Stelle des Präprozessor-Befehls "`#include <iostream.h>`" eingefügt wird. Nähere Informationen zur C++ Ein-/Ausgabe und zur Verwendung von Header-Dateien finden Sie auf Seite 191 ff bzw. 225.

C++-spezifisch

In C++ (nicht in C) kann auf verborgene globale Objekte mit dem Operator "`::`" zugegriffen werden:

Demo-Programm 2_04_02.cc

```
#include <iostream.h>      // für cout

int i = 5;                 // globale Variable

int main ()
{
    int i = 3;             // i mit lokaler Gültigkeit verbirgt globales i

    cout << "Lokales i = " << i << "\n";      // gibt aus "Lokales i = 3"
    cout << "Globales i = " << ::i <<"\n";    // gibt aus "Globales i = 5"

    return 0;
}
```

2.4.3. Dateiübergreifende Gültigkeit (*Linkage*)

Ein Programm besteht üblicherweise aus mehreren *Modulen* (*Übersetzungseinheiten; Translation Units*), deren Zugriff auf die Namen von Objekten oder Funktionen geregelt sein muss. Namen können folgende dateiübergreifende Gültigkeitsattribute haben:

- Interne (lokale): Namen in einer Übersetzungseinheit sind unabhängig von allfälligen identischen Namen in irgendeiner anderen Übersetzungseinheit (d.h., der selbe Name kann in verschiedenen Modulen verschiedene Objekte kennzeichnen). Lokale Objekte sind solche mit Datei-Gültigkeit (d.h., außerhalb aller Blöcke deklarierte) mit dem Schlüsselwort **`static`**:

  ```
  static int i;
  ```

- Externe: Externe Objekte sind solche mit Datei-Gültigkeit, die *nicht* mit dem Schlüsselwort **`static`** deklariert wurden. Externe Objekte, die *nicht* in der aktuellen Übersetzungseinheit *definiert* werden, können optional mit dem Schlüsselwort **`extern`** *deklariert* werden. Die folgende Tabelle gibt an, in welchen Kombinationen mehrere Module das Schlüsselwort **`extern`** bzw. Initialisierungen verwenden dürfen:

aktuelles Modul:	**andere Module:**	**Bemerkungen**

`int i;`	`int i;`
	`extern int i;`
	`int i = 3;`
		Initialisierung nur in *einem* Modul

`extern int i;`	`int i;`
	`extern int i;`
	`int i = 3;`
		Initialisierung nur in *einem* Modul

`int i = 3;`	`int i;` `extern int i;`	
`extern int i = 3;`	`int i;` `extern int i;`	`extern` *und* Initialisierung ist unzulässig (wird aber toleriert)

```
// Datei A:

int i = 3;                      // globale Variable deklariert und definiert

// Datei B:

#include <iostream.h>           // für cout

extern int i;                   // deklariert, aber nicht definiert
...
{
    cout << "i = " << i << "\n";            // gibt "i = 3" aus
}
```

- Keine dateiübergreifende Gültigkeit: Solche Objekte sind einmalig vorhanden; auf sie kann von außerhalb ihres Gültigkeitsbereiches nicht zugegriffen werden. Beispiele:
  - Alle Objekte mit lokaler Gültigkeit, sofern sie nicht mit dem Schlüsselwort "**`extern`**" deklariert wurden;
  - Funktions-Argumente;
  - Aufzählungen;
  - **`typedef`**-Namen.

2.4.4. Verknüpfung mit Nicht-C++-Funktionen

C++-spezifisch

Funktionen, die mit einer anderen Programmiersprache als C++ erstellt wurden, können unterschiedliche Konventionen für die Übergabe von Argumenten oder für die Umsetzung der originalen globalen Namen in das in Object-Dateien verwendete Format verwenden. Solche Funktionen (oder globale Daten, die in ihren Quelldateien definiert wurden) müssen explizit als solche angesprochen werden. Diese Funktionalität ist in C++ ist für Standard-C-Funktionen unter Verwendung des Strings "**extern "C"**" fix eingebaut; für andere Programmiersprachen existieren implementierungsspezifische Lösungen:

```
extern "C" int printf (const char *fmt, ...);
```

Mehrere Funktionen oder Objekte, die in einer anderen Sprache definiert wurden, können in geschweiften Klammern deklariert werden:

```
extern "C"
{
    void *malloc (size_t size);
    void free (void *ptr);
    int errno;
}
```

(In C werden globale Namen durch Vorsetzen von "**_**" *dekoriert* und in dieser Form im Object-Modul abgelegt. In C++ wird eine allfällige Klassenzugehörigkeit sowie Anzahl und Typ der Argumente (letzteres in codierter Form) in den dekorierten Namen mit aufgenommen.)

2.4.5. Speicherklassen

Speicherklassen bestimmen die Lebensdauer, dateiübergreifende Gültigkeit (*Linkage*) und die Behandlung von Objekten und Variablen. Die folgenden Speicherklassen für Daten sind definiert:

- Automatisch: Objekte mit automatischer Speicherklasse sind lokal für jeden Aufruf des Blocks, innerhalb dessen sie definiert wurden. Bei paralleler (*multithreaded*) oder rekursiver Ausführung eines Blocks wird garantiert, dass für jeden Aufruf des Blocks individuelle Speicherplätze zugeordnet werden. Automatische Daten werden in der Regel am Stack abgelegt. Alle Objekte, die innerhalb eines Blocks definiert wurden, haben standardmäßig die Speicherklasse **auto**, sofern sie nicht unter Verwendung der Schlüsselworte **extern** oder **static** deklariert wurden. Die Verwendung des Schlüsselwortes **auto** ist optional. Automatische Objekte haben keine dateiübergreifende Gültigkeit; sie existieren bis zum Ende des Blocks, innerhalb dessen sie deklariert wurden.
- Statisch: Objekte und Variable, die unter Verwendung des Schlüsselwortes **static** deklariert wurden, werden auf fix festgelegten Speicherplätzen im Arbeitsspeicher abgelegt. Bei paralleler (*multithreaded*) oder rekursiver Ausführung eines Blocks wird garantiert, dass für jeden Aufruf des Blocks auf *denselben* Speicherplatz und daher auf denselben Zustand oder Wert des Objekts zugegriffen wird. Statische Variable und Objekte bestehen vom Zeitpunkt ihrer Definition an bis zum Ende des Programms. Objekte und Variable, die *außerhalb* aller Blöcke definiert wurden, haben statische Speicherklasse. Sie haben standardmäßig *externe* Gültigkeit, außer, wenn sie mit dem Schlüsselwort **static** deklariert wurden; in diesem Fall sind sie nur innerhalb der aktuellen Übersetzungseinheit gültig. Die Verwendung des Schlüsselwortes **static** ist auch für die Deklaration von Funktionen zulässig; diese sind dann ebenfalls nur innerhalb der aktuellen Übersetzungseinheit gültig.
- Register: Variable, die unter Verwendung des Schlüsselwortes **register** deklariert wurden, werden (vorzugsweise, und soweit möglich) in einem Register der CPU abgelegt. Sie verhalten sich ansonsten exakt wie automatische Variable. Die Speicherklasse **register** ist nur für lokale Variable und für die formalen Argumente einer Funktion zulässig.
- Extern: Objekte oder Funktionen, die unter Verwendung des Schlüsselwortes **extern** deklariert werden, deklarieren ein Objekt, das in einer anderen Quelldatei oder in einem höheren Block mit externer Gültigkeit definiert wurde:

Demo-Programm 2_04_03.cc

```
#include <iostream.h>

extern int Anderswo;                    // in anderer Datei definiert
int Hier = 5;                           // hier definiert

int main ()
{
    int Hier = 7;
    {
        cout << "Hier = " << Hier << "\n";                 // "Hier = 7"
        int Hier = 11;
        cout << "Hier = " << Hier << "\n";                 // "Hier = 11"
        {
            extern int Hier;            // bezieht sich auf globales "Hier"

            cout << "Hier = " << Hier << "\n";             // "Hier = 5"
        }
        cout << "Hier = " << Hier << "\n";                 // "Hier = 11"
    }
}
```

Lokale *automatische* Objekte oder Variable werden *immer* dann initialisiert, wenn der Programmablauf ihre Definition erreicht. Lokale *statische* Objekte werden initialisiert, wenn der Programmablauf *erstmals* ihre Definition erreicht.

Demo-Programm 2_04_04.cc

```
#include <iostream.h>                   // für cout

void myfunc ()
{
    static int j = 1;
    int k = 1;

    cout << "j = " << j << ", k = " << k << "\n";
    j++; k++;
}

int main ()
{
    for (int i = 0; i < 5; i++)
        myfunc ();
}
```

In diesem Programm wird in der Funktion `main` aus einer Schleife heraus die Funktion `myfunc` fünfmal aufgerufen. In `myfunc` werden zwei Integer-Variable definiert, `j` mit statischer Speicherklasse, und `k` mit automatischer. Beide werden mit dem Wert `1` initialisiert, ausgegeben und anschließend inkrementiert. Die automatische Variable `k` wird bei *jedem* Aufruf von `myfunc` neu mit `1` initialisiert, während die statische Variable `j` nur beim *ersten* Aufruf von `myfunc` initialisiert wird, dann aber ihren Wert zwischen den Aufrufen beibehält. Das Programm erzeugt daher die folgende Ausgabe:

```
j = 1, k = 1
j = 2, k = 1
j = 3, k = 1
j = 4, k = 1
j = 5, k = 1
```

Je nach Implementierung können *statische* Variable und Objekte auch ohne explizite Initialisierung einen definierten Wert (meist `0`) aufweisen; *automatische* Variable enthalten vor ihrer Initialisierung in der Regel nur zufällige Daten.

C++-spezifisch

In C++ können (im Gegensatz zu C) Definitionen von Variablen und Objekten an jeder beliebigen Stelle eines Blocks (also nicht nur vor der ersten ausführbaren Anweisung) stehen.

Externe Objekte dürfen nur in einer einzigen Übersetzungseinheit des Programms initialisiert (= *definiert*), aber beliebig oft *deklariert* werden.

2.4.6. `const`, `volatile` und `inline`

2.4.6.1. `const`

Das Schlüsselwort **`const`** deklariert die Behandlung des mit diesem Attribut versehenen Objekts als Konstante:

- Deklaration echter Konstanten:

  C++-spezifisch

  ```
  const double Pi = 3.141592654;
  ```

  In C++ sollten Konstanten grundsätzlich auf diese Art und Weise deklariert werden. Dies steht im Gegensatz zu der in C üblichen Verwendung von Präprozessor-Direktiven:

  ```
  #define PI 3.141592654
  ```

  Insbesondere bei häufiger Verwendung von Gleitkomma-Konstanten kann die letztere Methode zu einer erheblichen Verschwendung von Speicherplatz beitragen, weil der Compiler für jede Verwendung der über den Präprozessor definierten Konstanten `PI` eine neue Kopie der Konstanten im Datenbereich anlegt, während die mit **`const`** deklarierte Konstante `Pi` genau einmal im Speicher vorkommt.

- Schutz von Argumenten, die per Referenz (über einen Zeiger) an eine Funktion übergeben werden, vor unbeabsichtigten Änderungen:

  ```
  char *strcpy (char *target, const char *source);
  ```

In beiden Fällen verhindert der Compiler eine Veränderung dieser Objekte:

```
Pi = 6.28;                // erzeugt eine Compiler-Fehlermeldung
```

C++-spezifisch

Objekte mit Datei-Gültigkeit, die explizit als **`const`** und *nicht* explizit als **`extern`** deklariert wurden, sind lokal für ihre Übersetzungseinheit. (In vielen Fällen, speziell bei ganzzahligen Konstanten, legt der Compiler solche konstante Objekte nicht im Datenbereich an, sondern baut ihre Werte direkt in den Programmcode ein. Damit ist ein Zugriff auf sie aus einer anderen Übersetzungseinheit grundsätzlich nicht möglich.)

```
const int lokal = 2;      // nur in aktueller Übersetzungseinheit zugänglich
extern const int global = 3;
                          // auch aus anderen Übersetzungseinheiten zugänglich
```

2.4.6.2. `volatile`

Das Schlüsselwort **`volatile`** gibt an, dass das betreffende Objekt während der Ausführung des Programms asynchron verändert werden kann (z.B. eine Semaphor-Variable, die von einem Interrupt-Handler aus gesetzt wird).

```
volatile int ReadyFlag;
```

Dieses Attribut dient in erster Linie dazu, gewisse Optimierungen bei der Erstellung des Object-Codes zu unterbinden. Eine mit **`volatile`** deklarierte Variable wird bei jedem Zugriff neu aus dem Arbeitsspeicher geladen (anderenfalls kann der Compiler Variable, auf die öfters zugegriffen wird, in ein Prozessorregister laden, oder wiederholt unter Verwendung der Variablen vorgenommene Berechnungen nur einmal ausführen).

2.4.6.3. `inline`

C++-spezifisch

Das Schlüsselwort **`inline`** kann bei der Deklaration von Funktionen verwendet werden. Der Code solcher Funktionen wird nicht, wie sonst üblich, *einmal* für das gesamte Programm generiert und über Unterprogrammaufrufe von den verschiedenen Stellen aus angesprungen, an denen die Funktion verwendet wird; er wird vielmehr für *jeden* Aufruf der Funktion *separat* vorgesehen. Damit kann der Overhead eines Funktionsaufrufs vermieden werden; das Programm wird schneller. Funktionell ist eine **`inline`**-Funktion daher äquivalent zu einem Makro; im Gegensatz zu Makros hat aber der Compiler die Möglichkeit, die Typen der Argumente und des Ergebnisses zu prüfen. Nebenwirkungen, wie sie bei Makros auftreten können, werden vermieden. **`inline`**-Funktionen sind sinnvoll für einfache Anwendungen (z.B. in Zugriffsfunktionen von Klassen), wo der Code-Overhead eines Funktionsaufrufs größer wäre als die Expansion des Inline-Codes. Funktionen, die als **`inline`** deklariert wurden, sind immer lokal für ihre Übersetzungseinheit.

```
inline int square (int i)
{
    return i*i;
}
```

2.4.7. Zusammenfassung

	innerhalb eines Blocks	**als Funktions-argument**	**außerhalb aller Blöcke**
deklariert ohne Schlüsselwort	zulässig	zulässig	zulässig
Gültigkeit (*Scope*)	lokal	lokal (für gesamte Funktion)	global
dateiübergreifend (*Linkage*)	keine	keine	extern
Lebensdauer	Block	Funktionsaufruf	Programm
Speicherklasse	**`auto`, `register`**	**`auto`**	**`static`**
`auto`	Standard	unzulässig	unzulässig
Gültigkeit	lokal	—	—
Linkage	keine	—	—
Lebensdauer	Block	—	—
`register`	zulässig	zulässig	unzulässig
Gültigkeit	lokal	lokal	—
Linkage	keine	keine	—
Lebensdauer	Block	Funktion	—
`static`	zulässig	unzulässig	zulässig
Gültigkeit	lokal	—	Datei
Linkage	keine	—	intern
Lebensdauer	Programm	—	Programm
`extern`	zulässig	unzulässig	zulässig
Gültigkeit	extern	—	extern
Linkage	extern	—	extern
Lebensdauer	Block oder Programm	—	Programm

3. Objekt-Typen

Dieser Abschnitt befasst sich mit den verschiedenen in C++ verfügbaren Typen von Daten-Objekten, die von einfachen fundamentalen Variablen über Felder und die insbesondere in der objektorientierten Programmierung benötigten zusammengesetzten Klassen-Typen bis zu benutzerdefinierten Datentypen reichen. Dazu kommen die hier ebenfalls kurz abgehandelten Funktionen, Zeiger und die in C++ neu eingeführten Referenz-Typen. Den Abschluss des Kapitels bilden die Regeln für die Umwandlung zwischen unterschiedlichen Datentypen.

Die hier gebotenen Informationen, insbesondere über Funktionen und Klassen-Typen, sind nur als einleitender Überblick zu verstehen. Eine detailliertere Beschreibung wird in den Abschnitten 4. und 5. erfolgen.

3.1. Fundamentale Typen

Fundamentale Datentypen sind mit der Programmiersprache definiert. Daten, die diesen Typen entsprechen, werden üblicherweise als "Variable" bezeichnet.

3.1.1. Die Type **void**

Die Type **void** hat einen leeren Satz von Eigenschaften. Sie darf *nicht* für die Deklaration von Variablen verwendet werden. Für **void** existieren zwei Anwendungen:

- Deklaration von Funktionen, die kein Ergebnis zurückgeben:

```
void "C" abort(void);
```

- "Generische" Zeiger auf Daten mit beliebigen oder nicht festgelegten Typen:

```
void *buffer;
void *malloc(size_t size);
```

Unter bestimmten Voraussetzungen dürfen Ausdrücke in den Typ **void** umgewandelt werden.

3.1.2. Ganzzahlige Typen

char:

Die Type **char** ist so definiert, dass sie die Elemente des Ausführungs-Zeichensatzes (i.a. ASCII) darzustellen erlaubt. (**char** entspricht daher meistens, aber nicht notwendigerweise immer, einem Byte zu 8 Bit.) Die Type **char** kann durch Beifügen von **signed** oder **unsigned** modifiziert werden; **char**, **signed char** und **unsigned char** werden vom Compiler als drei *verschiedene* Typen betrachtet. In den meisten Implementierungen wird standardmäßig **char** wie **signed char** behandelt; Compiler-Optionen erlauben jedoch eine Behandlung von **char** wie **unsigned char**. Von der Interpretation als **signed** oder **unsigned** hängt das Ergebnis von Vergleichsoperationen und die Umwandlung in **int** ab:

Beispiel für 8-Bit-**char** und 16-Bit-**int**:

```
signed char ch1 = '\x01', ch2 = '\x81';
unsigned char ch3 = '\x01', ch4 = '\x81';
int i;

i = ch2;                        // i = 0xff81 = -127
i = ch4;                        // i = 0x0081 = 129

i = ch1 > ch2;                  // i = 1 ("true"), weil 1 > -127
i = ch3 > ch4;                  // i = 0 ("false"), weil 1 < 129
```

short (oder **short int**):

Die Type **short** ist größer als oder gleich groß wie die Type **char** und kleiner oder gleich groß wie die Type **int**. **short** kann erweitert werden zu **signed short** und **unsigned short**. **short** (ohne Zusatz) oder **short int** ist immer identisch zu **signed short**.

int:

Die Type **int** ist größer als oder gleich groß wie die Type **short int** und kleiner oder gleich groß wie die Type **long int**. **int** kann erweitert werden zu **signed int** und **unsigned int**. **int** (ohne Zusatz) ist immer identisch zu **signed int**.

➡ `long` (oder `long int`):

Die Type **`long`** ist größer als oder gleich groß wie die Type **`int`**. **`long`** kann erweitert werden zu **`signed long`** und **`unsigned long`**. **`long`** (ohne Zusatz) oder **`long int`** ist immer identisch zu **`signed long`**.

Die tatsächlichen Größen der ganzzahligen Typen und die mit ihnen darstellbaren Wertebereiche hängen von der jeweiligen Implementierung ab:

16-Bit-Implementierung:

Type	Bytes	Minimum		Maximum	
`char`	1	-127	0x81	127	0x7f
`signed char`	1	-127	0x81	127	0x7f
`unsigned char`	1	0	0x00	255	0xff
`short`	2	-32767	0x8001	32767	0x7fff
`unsigned short`	2	0	0x0000	65535	0xffff
`int`	2	-32767	0x8001	32767	0x7fff
`unsigned int`	2	0	0x0000	65535	0xffff
`long`	4	-2147483647	0x80000001	2147483647	0x7fffffff
`unsigned long`	4	0	0x00000000	4294967295	0xffffffff

32-Bit-Implementierung:

Type	Bytes	Minimum		Maximum	
`char`	1	-127	0x81	127	0x7f
`signed char`	1	-127	0x81	127	0x7f
`unsigned char`	1	0	0x00	255	0xff
`short`	2	-32767	0x8001	32767	0x7fff
`unsigned short`	2	0	0x0000	65535	0xffff
`int`	4	-2147483647	0x80000001	2147483647	0x7fffffff
`unsigned int`	4	0	0x00000000	4294967295	0xffffffff
`long`	4	-2147483647	0x80000001	2147483647	0x7fffffff
`unsigned long`	4	0	0x00000000	4294967295	0xffffffff

3.1.3. Gleitkomma-Typen

➡ `float`:

Die Type **`float`** ist die kleinste Gleitkomma-Type.

➡ `double`:

Die Type **`double`** ist größer oder gleich der Type **`float`**, aber kleiner oder gleich der Type **`long double`**.

➡ `long double`:

Die Type **`long double`** ist größer oder gleich der Type **`double`**.

Speicherplatzbedarf, Wertebereiche und Auflösung der Gleitkommatypen sind wiederum implementierungsspezifisch:

16-Bit-Implementierung:

Type	Bytes	Minimum	Maximum	Auflösung
float	4	1.2e-38	3.4e38	1.2e-7
double	8	2.2e-308	1.8e308	2.2e-16
long double	10	3.4e-4932	1.2e4932	1.1e-19

32-Bit-Implementierung:

Type	Bytes	Minimum	Maximum	Auflösung
float	4	1.2e-38	3.4e38	1.2e-7
double	8	2.2e-308	1.8e308	2.2e-16
long double	8	2.2e-308	1.8e308	2.2e-16

3.2. Abgeleitete Typen

3.2.1. Direkt abgeleitete Typen

3.2.1.1. Felder von Variablen oder Objekten

Felder (*Arrays*) enthalten eine bei ihrer Deklaration festgelegte Anzahl von Elementen eines bestimmten Typs, der entweder fundamental, abgeleitet oder benutzerdefiniert sein kann.

Beispiele für die Definition von Feldern:

```
int IntArray[5];          // Feld aus 5 ints
POINT polygon[6];         // Feld bestehend aus 6 Elementen vom Typ POINT
```

Die einzelnen Elemente eines Feldes werden über einen in eckigen Klammern stehenden Index (eine ganzzahlige Konstante, Variable, oder ein Ausdruck mit ganzzahligem Ergebnis) adressiert. Feldindizes beginnen immer bei `0`. Das oben deklarierte Feld `IntArray` besteht daher aus den Elementen

```
IntArray [0]
IntArray [1]
IntArray [2]
IntArray [3]
IntArray [4]
```

C++ prüft standardmäßig *nicht* die Indizes eines Feldes auf die Gültigkeit ihres Wertebereichs.

Bei der Definition von Feldern, deren Größe durch die Initialisierung vorgegeben ist, braucht die Feldgröße nicht explizit angegeben zu werden:

```
int primes_to_20[] = {2,3,5,7,11,13,17};

char meldung[] = "Alle Files nicht löschen (J/N)?"

char *Bundesland[] = {
       "Wien",            "Niederösterreich",       "Burgenland",
       "Oberösterreich",  "Salzburg",               "Tirol",
       "Vorarlberg",      "Steiermark",             "Kärnten"
};
```

Mehrdimensionale Felder werden als Felder von Feldern interpretiert:

```
int Feld2d [5][7];
```

0, 0	0, 1	0, 2	0, 3	0, 4	0, 5	0, 6
1, 0	1, 1	1, 2	1, 3	1, 4	1, 5	1, 6
2, 0	2, 1	2, 2	2, 3	2, 4	2, 5	2, 6
3, 0	3, 1	3, 2	3, 3	3, 4	3, 5	3, 6
4, 0	4, 1	4, 2	4, 3	4, 4	4, 5	4, 6

Dieses Feld würde in zeilenweiser Reihenfolge der Elemente im Arbeitsspeicher angelegt. Bei mehrdimensionalen Feldern variiert der letzte Index am schnellsten. Bei Initialisierung mehrdimensionaler Felder kann die erste Felddimension entfallen.

3.2.1.2. Funktionen

Funktionen übernehmen keines oder mehrere *Argumente* mit einem bestimmten Datentyp und geben ein Resultat zurück (oder keines, wenn sie vom Typ **void** sind). Der Typ einer Funktion ist der Typ des Resultats der Funktion. Funktionen können daher überall dort verwendet werden, wo auch eine Konstante mit dem entsprechenden Typ eingesetzt werden könnte. Das folgende Beispiel verwendet die Standard-Bibliotheksfunktionen `atof()` und `cos()`:

```
double atof (const char *);             // String -> Wert vom Typ double
double cos (double);                    // Cosinus-Funktion

const char *Wert = "3.141592654";

#include <iostream.h>                   // für cout

void myfunc ()
{
    double x = atof (Wert);
    cout << "cos (" << Wert << ") = " << cos(x) << "\n";
    // ein etwas umständlicher Weg, um "cos (3.141592654) = -1" auszugeben...
}
```

Eine Funktion, die *nicht* vom Typ **void** ist, sollte unter allen Umständen (und für alle möglichen Verzweigungen) einen ihrem Typ entsprechenden Wert mit einem **return**-Befehl zurückgeben. Manche Compiler prüfen dies nicht standardmäßig!

Das folgende Beispiel definiert eine Funktion, deren Ergebnis für ein positives Argument `+1.`, für ein negatives `-1.`, und für ein verschwindendes Argument `0.` ist:

Demo-Programm 3_02_01.cc

```
double sign (double x)
{
    if (x > 0.)                         // positives Argument
        return 1.;

    else
        if (x < 0.)                     // negatives Argument
            return -1.;

        else                            // Argument == 0
            return 0.;
}
```

Die Funktion wäre auch eleganter (?) folgendermaßen zu schreiben:

```
double sign (double x)
{
    return x > 0. ? 1. : x < 0. ? -1. : 0.;
}
```

(Übrigens: Der Compiler generiert in beiden Fällen praktisch den gleichen *Object*-Code.)

Sowohl bei der *Deklaration* einer Funktion (*Funktions-Prototyp*) als auch bei ihrer *Definition* müssen die Anzahl der Argumente und ihre Typen sowie der Typ des Resultats konsistent angegeben werden:

Demo-Programm 3_02_02.cc

```
// Deklaration; erforderlich, wenn auf die Funktion vor ihrer Definition
// zugegriffen werden soll:

double Power (double Arg, unsigned Exp);
    // Funktions-Prototyp; die Namen der Argumente können auch weggelassen
    // werden:
    // double Power (double, unsigned);
```

```
// Definition:

double Power (double Arg, unsigned Exp)
{
    double Resultat;

    for (Resultat = 1.; Exp > 0; Exp--)
        Resultat *= Arg;

    return Resultat;
}
```

C++-spezifisch

Der Compiler prüft bei jedem Aufruf einer Funktion, ob die Typen der Argumente und des Ergebnisses mit der Deklaration übereinstimmen. Während in Standard-C bei Verwendung falscher Datentypen eine Fehlermeldung erfolgt, versucht der C++-Compiler im allgemeinen eine Konversion der Argument- oder Ergebnistypen.

```
double xq = Power (3.14, 2);     // Ok
int iq = Power (xq, -3);  // Fehler in Standard-C, nicht in C++:
                          // iq ist nicht double, und -3 ist nicht unsigned
```

Unter Verwendung der Funktion `Power` könnten wir ein Programm erstellen, das einen Gleitkommawert und einen ganzzahligen Exponenten vom Benutzer anfordert und den Gleitkommawert mit dem Exponenten potenziert:

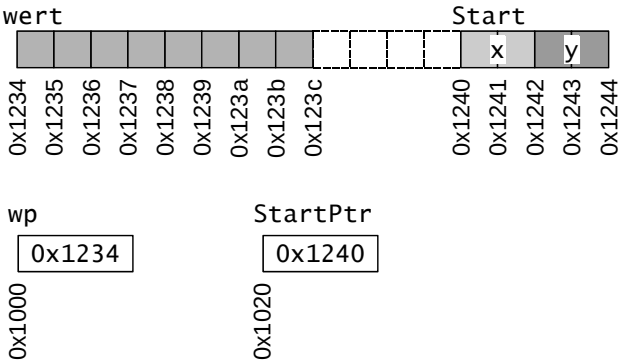
```
#include <iostream.h>            // für cout und cin

int main ()
{
    double x = 0;
        // muss hier deklariert werden, weil es außerhalb des folgenden
        // Blocks (in "while ...") benötigt wird
    do
    {
        int exp;    // kann auch innerhalb der Schleife deklariert werden

        cout << "\nx = ";                      // Ausgabe ("x = ")
        cin >> x;                             // Eingabe von x
        cout << "Exp = ";
        cin >> exp;

        cout << x << "^" << exp << " = " << Power(x,exp) << "\n";
    }
    while (x != 0.);
}
```

(`cin` ist der Konsol-Eingabe-*Stream* in C++; siehe Seite 196.)

3.2.1.3. Zeiger

Zeiger (*Pointer*) geben die *Adresse* eines Objekts oder einer Variablen vom entsprechenden Typ an. Die Adresse einer fundamentalen Variablen oder eines Objekts kann einem Zeiger unter Verwendung des *Adress*-Operators "**&**" zugewiesen werden:

```
double wert;
POINT Start;

double *wp = & wert;
    // wp ist ein Zeiger auf eine Variable vom Typ double.
POINT *StartPtr = & Start;
    // StartPtr ist ein Zeiger auf ein (beliebig komplexes) Objekt vom Typ
    // POINT.
```



Zeiger erlauben (unter anderem) die effiziente Übergabe von umfangreicheren Objekten als Argumente (Parameter) von Unterprogrammen (es muss nicht, wie bei der standardmäßigen Übergabe des Objekts als Wert, das gesamte Objekt auf den Stack kopiert werden, sondern nur ein *Zeiger* auf das Objekt). Die Übergabe eines Zeigers als Argument erlaubt auch die Änderung des Wertes des Arguments durch die Funktion (was aber tunlichst vermieden werden sollte!).

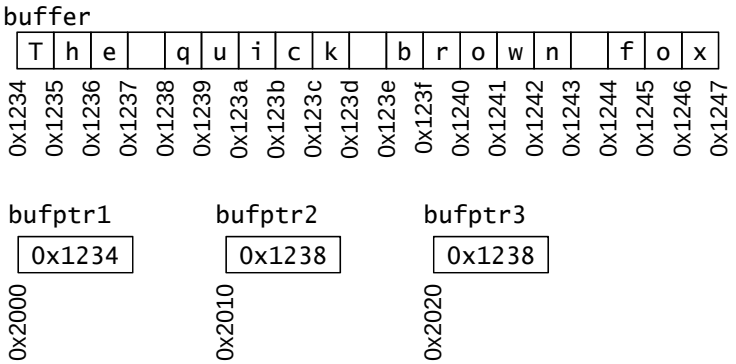
Wenn ein Zeiger auf den Beginn eines Feldes (z.B. eines Strings) zeigen soll, erübrigt sich die Angabe des "**&**"-Operators:

```
char buffer[80];
char *bufptr1 = buffer;
```

Ein Zeiger auf ein *beliebiges* Element eines Feldes kann folgendermaßen erhalten werden:

```
char *bufptr2 = & (buffer[4]);   // zeigt auf das fünfte Element des Feldes
char *bufptr3 = buffer + 4;      // alternativ und äquivalent zu obigem
```

Bei Feldern von Variablen und Objekten, die größer als ein Byte sind, multipliziert der Compiler den angegebenen Offset mit der Größe des Objekts (`sizeof(…)`) und addiert ihn zur Basisadresse, um die tatsächliche Adresse im Arbeitsspeicher zu ermitteln.



Um den *Wert* der Variablen oder des Objekts zu erhalten, auf die bzw. das der Zeiger weist, muss der Zeiger mit dem *Indirektions*-Operator "*****" *dereferenziert* werden.

```
char ch1 = *bufptr1;       // ch1 = 'T'
char ch2 = *bufptr2;       // ch2 = 'q'
```

Das folgende Programm gibt zeichenweise den Inhalt des **char**-Feldes buffer aus:

Demo-Programm 3_02_03.cc

```
#include <iostream.h>                    // für cout

char buffer[80]="The quick brown fox jumps over the lazy white dog";

int main ()
{
    char *bufptr = buffer;               // zeigt auf 1. Zeichen in buffer
```

```
    while (*bufptr)                     // solange das abschließende Null-Byte
                                        // noch nicht erreicht ist
        cout << *bufptr++;
                    // gib das Byte aus, auf das bufptr zeigt, und zeige
                    // anschließend auf das nächste Element
    return 0;
}
```

Zeiger gleichen Typs können voneinander subtrahiert werden (z.B. um die Länge eines Strings zu erhalten); zu einem Zeiger kann ein ganzzahliger Ausdruck addiert oder von ihm subtrahiert werden, um die Adresse, auf die er zeigt, zu modifizieren. Die Einheit dabei ist immer in Elementen der Type des Zeigers:

```
int IntFeld [100];
int *intptr = IntFeld;                  // weist auf IntFeld[0]

intptr += 3;                            // weist nun auf IntFeld[3]
```

Je nach der Größe von **int** (die mit `sizeof(int)` erhalten werden kann), erhöht sich der Wert von `intptr` im obigen Beispiel um 6 oder 12!

Zeiger und Felder werden daher vielfach äquivalent behandelt. De facto *ist* der Name eines Feldes ein Zeiger auf das erste Element des Feldes. Die folgenden zwei Funktionen `func1()` und `func2()` tun exakt dasselbe: sie weisen den Elementen von `IntFeld` die natürlichen Zahlen von 1 bis 100 zu:

Demo-Programm 3_02_04.cc

```
int IntFeld[100];                       // Feld mit 100 ints

void func1 ()
{
    register int i;                     // Index-Zählvariable

    for (i = 0; i < 100; i++)
        IntFeld [i] = i + 1;            // oder: *(IntFeld + i) = i + 1;
}

void func2 ()
{
    register int i;                     // Index-Zählvariable
    register int *intptr;               // Zeiger in IntFeld

    for (intptr = IntFeld, i = 0; i < 100; i++, intptr++)
        *intptr = i;
}
```

Beachten Sie aber bitte, dass der Compiler diese Äquivalenz zwischen Feldern und Zeigern *nur dann* korrekt behandeln kann, wenn eine korrekte Deklaration erfolgte. Dies gilt besonders bei Verwendung globaler (externer) Objekte:

```
// Modul 1:

int "C" puts (const char *);            // String-Ausgabe auf Konsole

char Meldung[] = "Datei nicht gefunden!";
...
    puts (Meldung);                     // funktioniert problemlos

// Modul 2:

int "C" puts (const char *);            // String-Ausgabe auf Konsole

extern char *Meldung;
...
    puts (Meldung);                     // schreibt Mist
```

Im ersten Fall *weiß* der Compiler, dass `Meldung` ein Feld vom Typ **char** ist, und übergibt seine *Adresse* an `puts`. Im zweiten Fall würde der Compiler die ersten *Zeichen* der Meldung als Adresse

interpretieren und an `puts` übergeben; `puts` würde ausgeben, was immer an dieser Adresse steht. Korrekt muss daher das zweite Programmodul lauten:

```
// Modul 2:

int "C" puts (const char *);           // String-Ausgabe auf Konsole

extern char Meldung[];
...
    puts (Meldung);                    // funktioniert wie erwartet
```

Derartige Fehler können in C überhaupt nicht vom Compiler entdeckt werden; in C++ kann aufgrund der *Dekoration* globaler Namen eine widersprüchliche Deklaration aufgezeigt werden.

C++-spezifisch

Über einen Zeiger kann grundsätzlich das Objekt, auf das er zeigt, geändert werden. Um *unerwünschte* Änderungen zu vermeiden, sollte das Schlüsselwort **`const`** bei der Deklaration des Zeigers verwendet werden:

```
const char *cptr1;              // Zeiger auf eine konstante Variable
char * const cptr2;             // Konstanter Zeiger (zeigt immer auf die
                                // gleiche char-Variable)
const char * const cptr3;       // Konstanter Zeiger auf konstante Variable
```

Der Compiler lässt nur Zuweisungen zu, die der Deklaration mit **`const`** nicht widersprechen:

```
const char cch = 'A';           // bleibt immer 'A'
char ch = 'B';                  // kann geändert werden
char ch1 = 'C';                 // kann geändert werden

      char *       pch1 = &ch;   // erlaubt
const char *       pch2 = &ch;   // erlaubt
      char * const pch3 = &ch;   // erlaubt
const char * const pch4 = &ch;   // erlaubt
      char *       pch5 = &cch;  // VERBOTEN
const char *       pch6 = &cch;  // erlaubt
      char * const pch7 = &cch;  // VERBOTEN
const char * const pch8 = &cch;  // erlaubt

*pch1 = 'A';        // Ok; normaler Zeiger
pch1 = &ch1;        // Ok; normaler Zeiger
*pch2 = 'A';        // Fehler; konstantes Objekt
pch2 = &ch1;        // Ok; Zeiger nicht konstant
*pch3 = 'A';        // Ok; Objekt nicht konstant
pch3 = &ch1;        // Fehler; konstanter Zeiger
*pch4 = 'A';        // Fehler; konstantes Objekt
pch4 = &ch1;        // Fehler; konstanter Zeiger

pch1 = pch3;        // Ok; ein nicht konstanter Zeiger darf einem konstanten
                    // gleichgesetzt werden
pch3 = pch1;        // Fehler; ein konstanter Zeiger darf nicht einem nicht
                    // konstanten gleich gesetzt werden
```

Analog kann das Schlüsselwort **`const`** verwendet werden, um die Veränderung eines Objektes zu verhindern, auf das ein als Funktionsargument übergebener Zeiger zeigt:

```
void func1 (char *buf1, const char *buf2)
{
    *buf1 = 'A';          // legal; buf1 ist nicht const
    *buf2 = 'B';          // Fehler: Objekt, auf das buf2 weist, darf nicht
                          // geändert werden
}

// Aufrufe:

char buffer[80];
func1 (buffer, buffer);         // Ok; nicht konstantes Objekt darf in ein
                                // konstantes umgewandelt werden
```

```
func1 ("Hello, world", buffer);  // Fehler; konstantes Objekt darf nicht in
                                 // ein nicht konstantes umgewandelt werden

func1 (buffer, "Hello, world");  // Ok
```

In C++ (und auch in C) existieren auch *Zeiger auf Funktionen* (die die Einsprungadresse der Funktion beinhalten). Funktions-Zeiger sind unersetzlich, wenn beispielsweise eine Bibliotheksfunktion ihrerseits eine benutzerdefinierte Funktion aufrufen soll, oder wenn über einen ganzzahligen Index-Wert schnell eine bestimmte Funktion ausgewählt und ausgeführt werden soll:

Demo-Programm 3_02_05.cc

```
int func1 (double, int, char *);
    // Deklaration (Prototyp) einer Funktion, die drei Argumente vom Typ
    // double, int, und Zeiger auf char übernimmt und ein Resultat vom Typ
    // int zurückgibt

int func2 (double, int, char *);       // detto
int func3 (double, int, char *);       // detto

int (*funcptr) (double, int, char *) = func1;
    // Zeiger namens "funcptr", der auf eine Funktion weist, die drei
    // Argumente vom Typ double, int, und Zeiger auf char übernimmt und ein
    // int-Resultat hat

int (*fparray[]) (double, int, char *) =
{
    func1,
    func2,
    func3
};          // Feld namens "fparray" von drei Funktions-Zeigern wie oben
```

Beachten Sie bitte die Klammern bei der Definition von `funcptr` und `fparray`:

```
int (*funcptr) (double, int, char *);  // ist ein FUNKTIONSZEIGER
int *(funcptr) (double, int, char *);  // ist eine FUNKTION mit Resultat int *
int *funcptr (double, int, char *);    // ebenso FUNKTION mit Resultat int *
```

Unter Verwendung der Definitionen von `funcptr` und `fparray` kann man beispielsweise schreiben:

```
int libfunc (int, int (*)(double, int, char*));
    // Prototyp einer Bibliotheksfunktion "libfunc", deren zweites Argument
    // ein Zeiger auf eine Funktion ist, wie sie oben definiert wurde

int irgendwas (int nPar)
{
    double Wert = 3.14;
    int Schalter = 12;
    char *Meldg = "Hello, world!";
    int i;

    // Funktion func1 über Funktions-Zeiger ausführen:
    i = funcptr (Wert, Schalter, Meldg);
        // äquivalent zu: i = func1 (Wert, Schalter, Meldg);

    // alternative Syntax:
    i = (*funcptr) (Wert, Schalter, Meldg);

    // Funktions-Zeiger als Argument übergeben:
    i = libfunc (4711, funcptr);

    // Funktion aus Feld der Funktions-Zeiger auswählen und ausführen:
    i = fparray [nPar] (Wert, Schalter, Meldg);
        // ist speziell bei zahlreichen Funktionen viel effizienter
        // als Programmverzweigungen
}
```

Die Bibliotheksfunktion `libfunc` könnte etwa so aussehen:

```
int libfunc (int nPar1, int (*fpUser) (double, int, char*))
{
    return fpUser (6.28, nPar1, "Error");
}
```

3.2.1.4. Referenzen (*References*)

C++-spezifisch

Referenzen (*References*) sind eigentlich Zeiger, die aber syntaktisch wie gewöhnliche Objekte behandelt werden. Besonders zweckmäßig ist ihre Verwendung im Zusammenhang mit komplexen Objekten.

```
int Wert;                       // gewöhnliche int-Variable
int& WertRef = Wert;            // Referenz auf Wert
int *WertPtr = &Wert;           // Zeiger auf Wert
```

Obwohl programmintern an der durch `WertRef` bestimmten Adresse im obigen Beispiel das Gleiche gespeichert wird wie in `WertPtr` (nämlich die Adresse von `Wert`), ist der Zugriff auf die eigentliche Variable über die Referenz einfacher als über den Zeiger. Die folgenden drei Zuweisungen haben identische Wirkung (`Wert` wird auf `3` gesetzt):

```
Wert = 3;
WertRef = 3;
*WertPtr = 3;
```

Bei Verwendung einer Referenz als Funktionsargument wird — ebenso wie bei der Verwendung eines Zeigers — die Übergabe komplexer Objekte erleichtert, weil nur die Adresse des Objekts und nicht das Objekt selbst übergeben werden muss. Der Zugriff auf ein Objekt ist aber über eine Referenz einfacher als über einen Zeiger.

Funktionen können auch Referenzen als *Resultat* zurückgeben. In diesem Fall kann das Resultat der Funktion verwendet werden, um ein Objekt zu verändern (was wichtig für objektorientierte Programmierung ist, wo auf Objekte nur über Funktionen zugegriffen werden soll). Damit ist es in C++ auch möglich, eine Funktion auf der *linken* Seite einer Zuweisung zu verwenden.

Das folgende Beispiel wurde bewusst *nicht* objektorientiert gewählt:

Demo-Programm 3_02_06.cc

```
int& WertFunc ()
{
    static int Wert;            // von außen nicht zugänglich
    return Wert;
}

// Setzen von Wert:
WertFunc () = 69;               // setzt Wert auf 69

// Auslesen von Wert:
int i = WertFunc ();
```

Die Deklaration einer Referenz-Type muss eine Initialisierung beinhalten, außer

- bei expliziter Deklaration als **`extern`**;
- bei der Deklaration eines Funktionsarguments oder -ergebnisses;
- bei der Deklaration eines Elements einer Klasse; und
- bei der Deklaration *innerhalb* einer Klasse.

3.2.2. Zusammengesetzte abgeleitete Typen

C++-spezifisch

Daten mit zusammengesetztem abgeleitetem Typ werden in C++ im allgemeinen als *Objekte* bezeichnet. Sie können nicht nur eine beliebige Anzahl von Datenelementen von beliebigem fundamentalem oder abgeleitetem Typ enthalten, sondern auch Funktionen, die für den jeweiligen Datentyp spezifisch sind.

3.2.2.1. Strukturen

Strukturen (*Structures*) vereinigen eine beliebige Anzahl von Objekten mit beliebiger Type, die (sinnvoller Weise) in einem logischen Kontext zueinander stehen. Das folgende Beispiel zeigt die Definition einer Struktur für Datum und Zeit-Information:

```
struct DateTime
{
    short Jahr;
    short Monat;
    short Tag;
    short Stunde;
    short Minute;
    short Sekunde;
} Heute;
```

Der Zugriff auf die einzelnen Elemente der Struktur erfolgt mittels des Operators "**.**". Jedes der so referenzierten Elemente kann genau so behandelt werden wie ein Objekt der betreffenden Type (z.B. `Heute.Jahr` wie eine Variable vom Typ **`short`**):

```
    Heute.Jahr = 1999;
    Heute.Monat = 4;
    Heute.Tag = 21;
```

Die Deklaration von Strukturen kann so wie im obigen Beispiel als gleichzeitige Deklaration der *Type* und eines *Objekts* dieser Type erfolgen; in der Regel werden aber Type und Objekte separat deklariert:

```
struct DateTime {
    short Jahr;
    short Monat;
    short Tag;
    short Stunde;
    short Minute;
    short Sekunde;
    };

struct DateTime Gestern;
DateTime Heute;                  // "struct" ist optional
```

C++-spezifisch

Im Gegensatz zu C ist in C++ die Verwendung des Schlüsselwortes **`struct`** optional, wenn die Struktur einmal deklariert wurde. Strukturen können als Argumente an eine Funktion übergeben oder von dieser zurückgegeben werden. Bei der Übergabe der Struktur als solche wird (wie auch sonst bei der Übergabe von Funktionsargumenten) eine *Kopie* des Objekts über den Stack übergeben; analog wird das mit **`return`** zurückgegebene Resultat in das Zielobjekt kopiert:

```
DateTime Vorjahr (DateTime Jetzt)
{
    Jetzt.Jahr--;
    return Jetzt;
}

DateTime LJahr = Vorjahr (Heute);
```

Günstiger ist dagegen die Übergabe von Zeigern an eine bzw. von einer Funktion oder die Verwendung von Referenzen:

```
// Zeiger auf eine Struktur:

DateTime *Vorjahr1 (DateTime *Jetzt)
{
    Jetzt->Jahr--;
    return Jetzt;
}

LJahr = *Vorjahr1 (&Heute);

// Referenz:

DateTime& Vorjahr2 (DateTime& Jetzt)
{
    Jetzt.Jahr--;
    return Jetzt;
}

LJahr = Vorjahr2 (Heute);
```

Mit Ausnahme der Deklaration der Funktion `Vorjahr2` resultiert bei Verwendung von Referenzen exakt der gleiche Programm-Quellcode wie bei Verwendung der Objekte selbst (Funktion `Vorjahr`). Hingegen müssen Zeiger auf Strukturen explizit dereferenziert werden. Dies geschieht mit dem Operator "`->`":

```
DateTime Heute;
DateTime *HeutePtr = & Heute;    // Zeiger auf die Struktur Heute

HeutePtr->Jahr = 1999;
HeutePtr->Monat = 4;
HeutePtr->Tag = 21;              // weist der Struktur Heute die gleichen
                                 // Werte zu wie im vorhergehenden Beispiel
```

Alternativ zum Operator "`->`" ("`HeutePtr->Jahr = 1999;`") hätten wir auch schreiben können:

```
 (*HeutePtr).Jahr = 1999;
```

Strukturen können ihrerseits auch Strukturen, Zeiger oder Felder enthalten; es gelten dann die folgenden Regeln für die Zugriffe auf diese Elemente:

```
struct POINT {
    unsigned x;           // x-Koordinate
    unsigned y;           // y-Koordinate
    };

struct Polygon {
    char *name;           // Bezeichnung
    short ecken;          // Eckenzahl (<= 16)
    POINT poly[16];       // Eckpunkte - Feld von Objekten vom Typ POINT
    };

Polygon Fuenfeck;         // Objekt der Type Polygon
Polygon *pFuenfeck;       // Zeiger auf ein Objekt der Type Polygon

pFuenfeck = & Fuenfeck;

// die folgenden Zeilen sind paarweise äquivalent in ihrer Wirkung

Fuenfeck.name = "Pentagramma";
pFuenfeck->name = "Pentagramma";
```

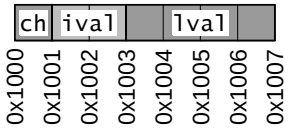
```
cout << Fuenfeck.name;
cout << pFuenfeck->name;

char ch1 = *Fuenfeck.name;       // ch1 = 'P'
char ch2 = *pFuenfeck->name;     // ch2 = 'P'

Fuenfeck.ecken = 5;
pFuenfeck->ecken = 5;

Fuenfeck.poly[1].x = 2;          // x-Wert Ecke 2
pFuenfeck->poly[1].x = 2;        // x-Wert Ecke 2

Fuenfeck.poly[1].y = 3;          // y-Wert Ecke 2
pFuenfeck->poly[1].y = 3;        // y-Wert Ecke 2

// die folgenden vier Zeilen setzen jeweils den x-Wert von Ecke 3.
// Beachten Sie bitte die Klammern!

((Fuenfeck.poly)+2)->x = 4;
((pFuenfeck->poly)+2)->x = 4;
(*((Fuenfeck.poly)+2)).x = 4;
(*((pFuenfeck->poly)+2)).x = 4;
```

Strukturen werden von manchen Compilern standardmäßig so im Arbeitsspeicher angelegt, dass der Zugriff der CPU auf die einzelnen Elemente besonders effizient erfolgen kann. Elemente, die größer als ein Byte sind, werden daher auf Adressen gelegt, die bei 16-Bit-Rechnern ein Vielfaches von 2 und bei 32-Bit-Rechnern ein Vielfaches von 4 sind. Das kann die Konsequenz haben, dass Strukturen, die binär in eine Datei kopiert wurden (siehe z.B. Seite 186, 201 und 205), nicht notwendigerweise von einem Programm gelesen werden können, das mit einem anderen Compiler erstellt wurde als das Ausgabeprogramm. Das Standard-Verhalten kann mit der Präprozessor-Direktive **`#pragma pack(`*n*`)`** bei Bedarf geändert werden. ***Vorsicht***: Diese Direktive ist nicht bei allen Compilern implementiert (zum Beispiel nicht bei GNU C/C++). Ihre Verwendung kann daher Portabilitätsprobleme verursachen!

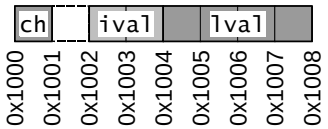
Die Konsequenz der unterschiedlichen Parameter von **`#pragma pack`** soll anhand des folgenden Beispiels illustriert werden:

```
struct test
{
    char ch;
    short ival;
    long lval;
};
```

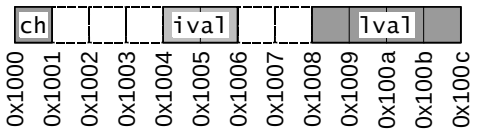
`#pragma pack (1)`



`#pragma pack (2)`



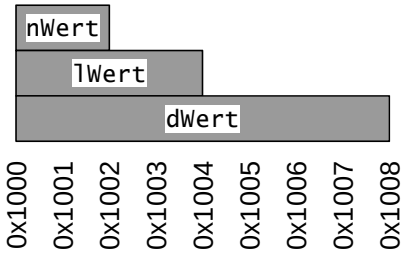
`#pragma pack (4)`



3.2.2.2. Unions

Im Gegensatz zu Strukturen, deren Elemente auf individuellen Speicherplätzen angelegt werden, und die daher zu einem gegebenen Zeitpunkt *mehrere* Elemente enthalten können, verwenden *Unions* den *gleichen* Speicherbereich für ihre Elemente und können daher zu einem gegebenen Zeitpunkt nur *ein* Datenelement (allerdings unter unterschiedlichen Interpretationen) enthalten:

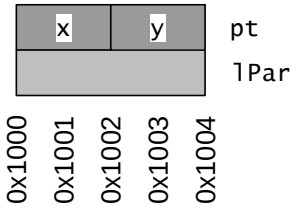
```
union Zahlentype
{
    short  nWert;
    long   lWert;
    double dWert;
};
```



Die Verwendung einer Union erlaubt *nicht* die Umwandlung von Werten zwischen verschiedenen Zahlenformaten; der gleiche Speicherplatz wird nur mehrfach belegt.

Zweckmäßig ist die Verwendung einer Union dann, wenn z.B. zwei Werte vom Typ **`short`** als ein Wert vom Typ **`long`** übergeben werden sollen:

```
struct POINT
{
    short x;
    short y;
}

union LongPar
{
    POINT pt;
    long  lPar;
};
```



Der Zugriff auf ein Element einer Union erfolgt analog wie bei einer Struktur. Das folgende Beispiel weist den x- und y-Koordinaten eines Punktes Werte zu und übergibt die resultierende Datenkombination als Funktionsargument vom Typ **`long`**:

```
int WinFunc (int Switch, long Param);
    // Switch bestimmt, wie Param zu interpretieren ist

LongPar Punkt;

Punkt.pt.x = 2;
Punkt.pt.y = 3;
WinFunc (WF_POINT, Punkt.lPar);
```

Vorsicht: Die Verwendung von Unions zum "Bauen" von Daten, die tatsächlich z.B. als **long** interpretiert werden sollen, ist *nicht portabel*, weil die Anordnung der Bytes in Werten, die aus mehreren Bytes bestehen, von der verwendeten CPU abhängt (höchstwertiges oder niedrigstwertiges Byte an der niedrigsten Adresse — *Big* oder *Small Endian*).

C++-spezifisch

3.2.2.3. Klassen

Klassen (*Classes*) stellen eine Erweiterung der Funktionalität von Strukturen dar. Strukturen *sind* Klassen; ein Objekt, das als **struct** deklariert wurde, kann anschließend als **class** definiert werden, oder umgekehrt:

```
struct A;           // Voraus-Deklaration

class A             // Definition
{
public:
    int i;
};
```

Der wesentliche *Unterschied* zwischen Strukturen und Klassen liegt darin, dass auf die Elemente einer Struktur standardmäßig von beliebigen Funktionen zugegriffen werden kann, während auf die Elemente einer Klasse standardmäßig nur Funktionen zugreifen können, die dieser Klasse angehören (*Klassenelement-Funktionen*). Mit den Schlüsselworten **public** und **private** kann dieses Verhalten geändert werden:

```
class X
{
public:             // alle folgenden Elemente sind frei zugänglich
    int i;
    int j;

private:            // alle folgenden Elemente sind nur für Funktionen
    int k;          // dieser Klasse zugänglich
    int l;
};
```

Der Umstand, dass Datenelemente nicht frei zugänglich sind, erfordert *Zugriffsfunktionen*, die Bestandteil der Klasse sind und mit dieser definiert werden. Zwei spezielle Klassenelement-Funktionen, die *Konstruktor*- und die *Destruktor*-Funktion, erlauben zudem die Erstellung und Initialisierung eines neuen Objekts der Klasse bzw. das "Aufräumen", wenn ein Objekt nicht mehr gültig ist. Der Name des Konstruktors ist immer gleich dem Namen der Klasse, der Name des Destruktors ist der Klassenname mit vorangestellter Tilde "~".

Demo-Programm 3_02_07.cc

```
#include <iostream.h>                       // für cout

class Konto                                 // Klassenname "Konto"
{
public:
    Konto ()                                // Konstruktor
        { Kontostand = 0.; }

    void Einlage (double Betrag)            // Zugriffsfunktion
        { Kontostand += Betrag; }

    void Abhebung (double Betrag)           // Zugriffsfunktion
        { Kontostand -= Betrag; }

    double Stand ()                         // Zugriffsfunktion
        { return Kontostand; }

private:
    double Kontostand;                      // nicht direkt zugänglich
};
```

```
int main ()
{
    Konto Sparbuch; // Definition des Objekts "Sparbuch" der Klasse "Konto"

    // Mit der Definition eines Objekts einer Klasse wird die Konstruktor-
    // Funktion aufgerufen, die in diesem Fall den Kontostand auf 0 setzt.

    cout << "Saldo: " << Sparbuch.Stand() << "\n";         // gibt "0" aus

    Sparbuch.Einlage (500.);
    Sparbuch.Abhebung (200.);

    cout << "Saldo: " << Sparbuch.Stand() << "\n";         // gibt "300" aus
}
```

Da die Klassenelement-Funktionen logischer Teil der Klasse sind, müssen sie (analog zu einem Element einer Struktur) in der Form `Objekt.Funktionsname` aufgerufen werden. Sie können entweder wie im obigen Beispiel bei der Deklaration der Klasse oder aber separat definiert werden. Jedes *Objekt* einer Klasse belegt individuell Speicherplatz für seine Datenelemente; die Klassenelement-*Funktionen* existieren hingegen grundsätzlich nur *einmal* in einem Programm (außer, sie wurden als **`inline`**-Funktionen definiert). Da aber Klassenelement-Funktionen im allgemeinen Zugriff auf Datenelemente der Klasse benötigen, benötigt der Compiler die Angabe des *Objekts*, dessen Datenelemente die Funktion bearbeiten soll. Wir hätten im obigen Demo-Programm beispielsweise mehrere Objekte der Klasse `Konto` definieren können (z.B. `Sparbuch`, `Girokonto`, `Kredit`); ein Aufruf von `Girokonto.Einlage()` hätte dann nur das Datenelement `Kontostand` des Objekts `Girokonto` (`Girokonto.Kontostand`) betroffen, aber nicht die Datenelemente `Sparbuch.Kontostand` oder `Kredit.Kontostand`.

Klassenelement-Funktionen, die *mit* der Klasse definiert wurden, werden in manchen Implementierungen immer als **`inline`**-Funktionen angelegt. Separat definierte Klassenelement-Funktionen sind standardmäßig "normale" Funktionen; bei ihrer Definition muss ihrem Namen der Name der Klasse, gefolgt vom Operator "`::`" (z.B. "`Konto::`"), vorangestellt werden. Alternativ wäre für die Deklaration der Klasse `Konto` daher auch die folgende Vorgangsweise möglich:

Demo-Programm 3_02_08.cc

```
class Konto                                     // Klassenname "Konto"
{
public:
    // Hier nur DEKLARATIONEN der Klassenelement-Funktionen:

    Konto ();                                   // Konstruktor

    void Einlage (double Betrag);               // Zugriffsfunktion
    void Abhebung (double);                     // Zugriffsfunktion
    double Stand ();                            // Zugriffsfunktion

private:
    double Kontostand;                          // nicht direkt zugänglich
};

// DEFINITIONEN der Klassenelement-Funktionen:

Konto::Konto ()                                 // Konstruktor
    { Kontostand = 0.; }

void Konto::Einlage (double Betrag)             // Zugriffsfunktion
    { Kontostand += Betrag; }

void Konto::Abhebung (double Betrag)            // Zugriffsfunktion
    { Kontostand -= Betrag; }

double Konto::Stand ()                          // Zugriffsfunktion
    { return Kontostand; }
```

Die Datenelemente einer Klasse sind lokal für jedes Objekt dieser Klasse, d.h., für jedes Objekt der Klasse `Konto` existiert genau ein Datenelement `Kontostand`. Wird ein Element einer Klasse jedoch als **`static`** deklariert, wird *genau ein* derartiges *statisches Klassenelement* (mit externer Gültigkeit) angelegt. Solche statische Elemente werden mit der Deklaration einer Klasse zwar

deklariert, sie müssen aber explizit *definiert* werden. Analog können statische Klassenelement-Funktionen vorgesehen werden, die nicht auf Datenelemente eines bestimmten Objekts zuzugreifen brauchen. Da statische Datenelemente und Funktionen unabhängig von allen allfälligen Objekten dieser Klasse existieren, können sie auch dann verwendet werden, wenn noch kein Objekt dieser Klasse existiert. Das folgende Beispiel illustriert dieses Verhalten anhand eines Zählers für Objekte der Klasse `Punkt`:

Demo-Programm 3_02_09.cc

```
#include <iostream.h>                          // für cout

class Punkt
{
public:
    Punkt ()                                   // Konstruktor
        { PunkteZahl++; }
    ~Punkt ()                                  // Destruktor
        { PunkteZahl--; }

    // Zugriffsfunktionen:
    unsigned& x()
        { return xKoord; }
    unsigned& y()
        { return yKoord; }

    static int Anzahl()                        // statische Zugriffsfunktion
        { return PunkteZahl; }

    static int PunkteZahl;                     // statisches Datenelement

private:
    unsigned xKoord;
    unsigned yKoord;
};

int Punkt::PunkteZahl = 0;       // Definition von Punkt::PunkteZahl

int main ()
{
    cout << "Anzahl der Punkte: " << Punkt::Anzahl() << "\n";
                                  // "Punkt::Anzahl()", weil statische Funktion
    Punkt p1;                     // neues Objekt
    cout << "Anzahl der Punkte: " << Punkt::Anzahl() << "\n";
    {
        Punkt p2;
        cout << "Anzahl der Punkte: "<< Punkt::Anzahl() << "\n";
        {
            Punkt p3;
            cout <<"Anzahl der Punkte: " << Punkt::Anzahl () << "\n";

            // Koordinaten zuweisen - Zugriffsfunktion verwendet Referenzen
            // für ihr Ergebnis!
            p1.x() = 20;
            p1.y() = 30;
        }

        // Punkt p3 existiert nicht mehr

        cout << "Anzahl der Punkte: "<< Punkt::Anzahl() << "\n";
    }

    // Punkt p2 existiert nicht mehr

    cout << "Anzahl der Punkte: " << Punkt::Anzahl() << "\n";

    // Koordinaten von Punkt p1 ausgeben

    cout << p1.x() << ", " << p1.y() << "\n";
}
```

Das Demo-Programm zeigt die Verwendung einer statischen Funktion (`Anzahl()`) als Element der Klasse `Punkt`, die den Wert der statischen Variablen `PunkteZahl` als Ergebnis zurückgibt. Statische Funktionen als Element einer Klasse können jederzeit aufgerufen werden, auch ohne dass ein Objekt dieser Klasse existiert. Ihrem Namen muss, ebenso wie auch bei Zugriffen auf statische Datenelemente einer Klasse, entweder der Name der Klasse, gefolgt vom Operator "`::`" (z.B. "`Punkt::`"), oder der Name eines *Objekts* der Klasse, gefolgt vom Operator "`.`" bzw. "`->`", vorangestellt werden (siehe Seite 104).

Im Demo-Programm werden drei Objekte vom Typ `Punkt` in drei ineinandergeschachtelten Blöcken definiert. Da es sich um automatische Variable mit Blockgültigkeit handelt, existiert jede dieser Variablen nur, bis der Block, innerhalb dessen sie deklariert wurde, wieder verlassen wird. Bei der Definition der Objekte wird die Konstruktor-Funktion (`Punkt()`) ausgeführt, die die Variable `Punkt::PunkteZahl` inkrementiert. Beim Verlassen des Blocks wird für die ungültig werdenden Objekte die Destruktor-Funktion (`~Punkt()`) automatisch ausgeführt, die `Punkt::PunkteZahl` wieder dekrementiert. Das Programm erzeugt also die Ausgabe:

```
Anzahl der Punkte: 0
Anzahl der Punkte: 1
Anzahl der Punkte: 2
Anzahl der Punkte: 3
Anzahl der Punkte: 2
Anzahl der Punkte: 1
20, 30
```

Ebenso wie bei Strukturen können *Zeiger* auf Objekte einer Klasse definiert und verwendet werden. Zusätzlich besteht in C++ die Möglichkeit der Verwendung von Zeigern auf ein *Element* der Klasse: Im Gegensatz zu einem "gewöhnlichen" Zeiger auf ein Element eines Klassenobjekts, der für jedes Objekt gesondert definiert werden müsste, gilt die Definition eines Zeigers auf ein Element einer Klasse für *alle* Objekte dieser Klasse (und der daraus abgeleiteten Klassen). Ein Zeiger auf ein Element einer Klasse wird unter Verwendung des Klassennamens, gefolgt von "`::*`", deklariert. Ein solcher Zeiger kann unter Voranstellung des Namens des Objektes, gefolgt von "`.*`", dereferenziert werden (das Ergebnis ist der Wert des entsprechenden Elements des Objekts, auf das er zeigt), oder durch Voranstellen eines Zeigers auf das Objekt, gefolgt von "`->*`".

Das folgende Beispiel veranschaulicht die Verwendung dieser Operatoren:

Demo-Programm 3_02_10.cc

```
// Klasse "AClass" definieren

#include <iostream.h>                          // für cout

class AClass
{
public:
    int Wert;
    void Anzeigen () {cout << Wert << "\n";}
};

// Abgeleitete Type pDAT, die auf Elemente "Wert" von AClass zeigt:

int AClass::*pDAT = & AClass::Wert;

int main ()
{
    AClass A;                   // Objekte der Klasse "AClass"
    AClass B;
    AClass *pA = &A;            // Zeiger auf "A"
    AClass *pB = &B;            // Zeiger auf "B"

    A.*pDAT = 1234;             // Zuweisung, de facto auf A.Wert
    B.*pDAT = 2345;             // Zuweisung, de facto auf B.Wert

    A.Anzeigen ();              // A.Wert ausgeben
    B.Anzeigen ();              // B.Wert ausgeben

    pA->*pDAT = 4321;           // Dereferenzieren des Zeigers auf A
    pB->*pDAT = 5432;
```

```
    pA->Anzeigen ();            // Zeiger
    A.Anzeigen ();              // Objekt
    pB->Anzeigen ();            // Zeiger
    B.Anzeigen ();              // Objekt
}
```

Ein Zeiger auf ein Element einer Klasse ist eine *Type*, kein *Objekt*! Dieses Konstrukt erspart eine individuelle Deklaration von Zeigern auf die Elemente *jedes* Objekts, etwa für das obige Beispiel:

```
int *pAw = & A.Wert;
int *pBw = & B.Wert;
int *ppAw = & pA->Wert;
int *ppBw = & pB->Wert;
```

3.2.2.4. Bitfelder

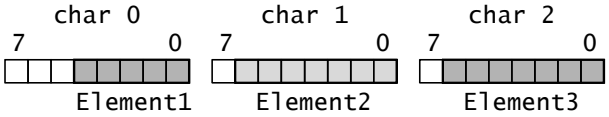
Objekte, die mit **`struct`** oder **`class`** deklariert wurden, können Elemente (Bitfelder, *Bit Fields)* enthalten, die kleiner als eine ganzzahlige Datentype sind. Solche Elemente werden mit ganzzahliger Type, gefolgt von einem Doppelpunkt ("**`:`**") und einem ganzzahligen konstanten Ausdruck deklariert, der die Anzahl der Bits in dem Bitfeld angibt. *Anonyme* (also namenlose) Bitfelder können als Platzhalter verwendet werden. Ein Element der Größe `0` erzwingt den Beginn des darauffolgend deklarierten Elements in einem neuen Element der bei der Definition des Bitfelds verwendeten ganzzahligen Type. Man wählt üblicherweise **`unsigned`**-Typen für Bitfelder, obwohl dies nicht von der Syntax verlangt wird. Typisch werden Bitfelder für hardwarenahe Problemstellungen verwendet, bei denen die einzelnen Bits eines Bytes, Wortes oder Doppelwortes unabhängig voneinander behandelt werden müssen. Die Größe des Basistype des Bitfelds (**`unsigned char`**, **`unsigned short`** oder **`unsigned long`**) bestimmt die Anordnung der Bitfelder mit; man wird sie zweckmäßigerweise entsprechend der von der Anwendung festgelegten Datengröße (8, 16 oder 32 Bit) wählen.

Das folgende Beispiel definiert vier verschiedene Bitfeld-Typen und zeigt, wie Bitfelder angeordnet werden, die über die Grenzen ihrer Basistype hinausreichen würden:

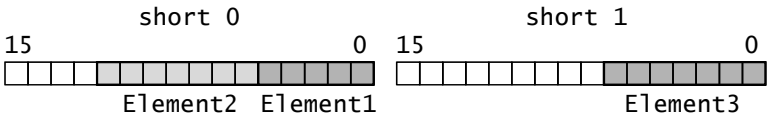
```
struct CharFeld
{
    unsigned char Element1 : 5;
    unsigned char Element2 : 7;
    unsigned char Element3 : 7;
};

struct ShortFeld
{
    unsigned short Element1 : 5;
    unsigned short Element2 : 7;
    unsigned short Element3 : 7;
};

struct LongFeld
{
    unsigned long Element1 : 5;
    unsigned long Element2 : 7;
    unsigned long Element3 : 7;
};

struct AlignedShortFeld
{
    unsigned short Element1 : 5;
    unsigned short          : 0;
                          // zwingt Element2 in nächstes unsigned short
    unsigned short Element2 : 7;
    unsigned short Element3 : 7;
};
```

```
// Objekte dieser Bitfeld-Typen definieren

CharFeld cf;
ShortFeld sf;
LongFeld lf;
AlignedShortFeld asf;
```

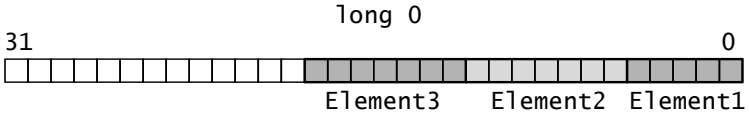
CharFeld:



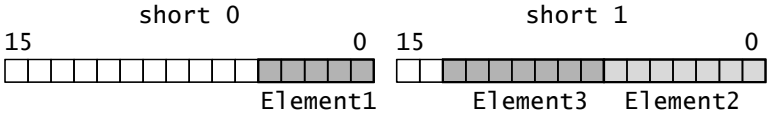
ShortFeld:



LongFeld:



AlignedShortFeld:



Unzulässige Operationen für Bitfelder sind:

- Adresse eines Bitfeld-Elements ermitteln (z.B. "`&sf.Element2`");
- Eine Referenz mit einem Bitfeld-Element initialisieren.

3.3. Benutzerdefinierte Typen

3.3.1. Aufzählungen

Aufzählungen (*Enumerations*) sind ganzzahlige Datentypen, die *benannte Konstanten* definieren:

```
enum Tag
{
    Sonntag,
    Montag,
    Dienstag,
    Mittwoch,
    Donnerstag,
    Freitag,
    Samstag
};

int main ()
{
    Tag Heute;             // Deklaration eines Objekts "Heute" vom Typ "Tag"
    Heute = Mittwoch;
}
```

Aufzählungstypen werden intern als ganze Zahlen dargestellt, wobei standardmäßig der erste Name (*Enumerator*) in der Liste ("`Sonntag`") als "0" dargestellt wird, der zweite als "1", usw. Sie können überall dort verwendet werden, wo Konstanten verwendet werden können (also *nicht* als Ergebnis eines Ausdrucks!):

```
Heute = Samstag;                        // Ok
if (Heute == Montag)...                 // Ok

Tag Morgen;
Morgen = Heute++;                       // unzulässig!
```

(Die Möglichkeit, in C++ für eine Klasse die Bedeutung eines Operators neu zu definieren, erlaubt einen Ausweg aus dieser Einschränkung. Ein anderer Weg ist die Verwendung einer expliziten Typumwandlung.)

Wenn die numerischen Werte einer Aufzählung von der Standardeinstellung abweichen sollen, ist eine explizite Zuweisung möglich:

```
enum Tag1
{
    Sonntag,
    Montag = 5,
    Dienstag,
    Mittwoch,
    Donnerstag = 3,
    Freitag,
    Samstag
};
```

Damit gilt für die obigen Enumeratoren:

```
Sonntag = 0,
Montag = 5,
Dienstag = 6,
Mittwoch = 7,
Donnerstag = 3,
```

```
Freitag = 4,
Samstag = 5.
```

Der Name eines Enumerators muss eindeutig sein, darf also für kein anderes Objekt im Gültigkeitsbereich des Enumerators verwendet werden.

Enumeratoren haben keine dateiübergreifende Gültigkeit (*Linkage*). Es ist also nicht möglich, sich in einem Modul auf einen in einem anderen Modul definierten Enumerator zu beziehen. (Das impliziert, dass in verschiedenen Modulen verschiedene Definitionen der gleichen Aufzählung existieren können, was Ursache von Fehlern sein kann.)

Aufzählungen ersetzen einen *Zahlenwert* durch einen logischen *Namen*. Die Verwendung eines Enumerators ist aber auf Objekte beschränkt, die als Aufzählung (unter Verwendung eben dieses Enumerators) definiert wurden.

```
int i = Montag;                    // unzulässig
```

Alternative und universell verwendbare Möglichkeiten, Konstante durch Namen zu ersetzen, sind:

- Präprozessor-Makros:

```
#define MONTAG 1
int i = MONTAG;                    // Ok; ist "int i = 1;"
```

- Definition mit dem Schlüsselwort **const**:

```
const int montag = 1;
int i = montag;                    // Ok; ist "int i = 1;"
```

3.3.2. Synonyme (**typedef**)

Das Schlüsselwort **typedef** erlaubt die Zuweisung eines Namens an beliebige (und beliebig komplexe) Daten*typen*. Der mit **typedef** angegebene Name wird also zum Namen einer Datentype und kann überall dort verwendet werden, wo fundamentale oder abgeleitete Typen verwendet werden können:

```
typedef unsigned char BYTE;
typedef unsigned short WORD;
typedef unsigned long DWORD;

typedef BYTE * PBYTE;                          // entspricht "unsigned char *"

typedef int(*)(double,int,char*) MyType;       // Funktions-Zeiger von Seite 38
```

typedef-Namen können in (nachfolgenden) **typedef**s vorkommen. Wie sehr benutzerdefinierte Synonyme Deklarationen vereinfachen können, beweist die Verwendung der Type `MyType` in der Deklaration der Bibliotheksfunktion von Seite 38:

```
int libfunc (int, MyType);
```

steht für:

```
int libfunc (int, int (*)(double, int, char*));
```

Mit **typedef** definierte Synonyme sind nur innerhalb der Datei gültig, in der sie definiert wurden; sie haben keine dateiübergreifende Gültigkeit (*Linkage*). In C++ (nicht aber in C) sind mehrere *in ihrem Wesen* identische Definitionen mit **typedef** in einer Datei zulässig; sie dürfen einander aber nicht widersprechen:

C++-spezifisch

```
typedef char CHAR;

typedef char CHAR;                 // Ok; identische Definition
```

```
typedef CHAR CHAR;              // Ok; kein Widerspruch

typedef signed char CHAR;       // Fehler! signed char != char !
```

Die Namen von **typedef**s müssen sich von denen von Variablen, Funktionen und anderen Objekten unterscheiden; sie können aber durch Deklaration eines Objekts mit gleichem Namen in einem eingeschlossenen Block verborgen werden:

```
typedef unsigned long ULONG;

void func1 ()
{
    int ULONG;              // Ok; verbirgt typedef
    ULONG ul;               // Fehler: ULONG ist jetzt Objekt vom Typ int
}
```

C++-spezifisch

Von dem in C gern geübten Brauch, **typedef**s für die Deklaration von Klassen (Strukturen, **union**s) zu verwenden, sollte in C++ abgesehen werden, obwohl der folgende Code aus Kompatibilitätsgründen auch in C++ akzeptiert wird:

```
typedef struct              // namenlose Struktur
{
    short x;
    short y;
} PUNKT;                    // benannt als "PUNKT"

PUNKT p1, p2;               // Definition zweier Objekte der obigen Struktur
```

3.4. Umwandlungen zwischen Typen

3.4.1. Implizite Umwandlungen

Im Zuge von arithmetischen Ausdrücken, bei der Übergabe von Argumenten an Funktionen und bei der Zuweisung eines Ergebnisses eines Ausdrucks sieht der Compiler eine automatische Umwandlung in andere Datentypen vor, um Verknüpfungen zwischen Daten gleichen Typs vornehmen oder Daten mit korrektem Typ übergeben zu können.

Ein als *Integral Promotion* bezeichneter Vorgang erlaubt die Verwendung der folgenden Objekte überall dort, wo ganzzahlige Datentypen verwendet werden dürfen:

- Objekte vom Typ **char** und **short int**;
- Aufzählungs-Typen;
- Bitfelder in ganzzahligen Variablen;
- Enumeratoren.

Bei der *Integral Promotion* bleibt im Gegensatz zu echten Typenkonversionen das Bitmuster der zu konvertierenden Variablen unverändert; es werden allenfalls zusätzliche Füll-Bits eingefügt. Die Konversion erfolgt in die Type **int**, sofern **int** geeignet ist, den gesamten Wertevorrat darzustellen; ansonsten erfolgt sie in die Type **unsigned int**. Dabei wird der *Wert*, nicht aber unbedingt das *Vorzeichen* erhalten. Variable vom Typ **unsigned int** werden in 16-Bit-Systemen in **long** umgewandelt (weil 32-Bit **long**s jeden 16-Bit **unsigned int**-Wert darstellen können); in 32-Bit-Systemen erfolgt ihre Umwandlung in **unsigned long**.

Dieses Verhalten kann zu unterschiedlichen Resultaten bei Vergleichsoperationen auf 16- und 32-Bit-Systemen führen: So ergibt auf einem 16-Bit-System der Ausdruck "`-1L < 1U`" den Wert 1 (`true`), weil beide Seiten in den Typ **signed long** umgewandelt werden und `-1L` tatsächlich kleiner als `+1L` ist. Hingegen ergibt der gleiche Ausdruck auf einem 32-Bit-System den Wert 0 (`false`), weil beide Seiten in **unsigned long** konvertiert werden müssen und `-1UL` (= 2^{32} - 1) mit `+1UL` (= 1) verglichen wird.

Bei der Konversion ganzzahliger Datentypen von **signed** in **unsigned** bleibt grundsätzlich das Bitmuster der Variablen erhalten, nur die Interpretation ändert sich:

Demo-Programm 3_04_01.cc

```
#include <iostream.h>

int main ()
{
    short i = -3;
    unsigned short u;
    cout << (u = i) << "\n";            // Wichtig: Klammer um "(u = i)"
    return 0;
}
```

Das Programm gibt die Zahl `65533` (2^{16} - 3) aus.

Bei der Konversion von **unsigned** in **signed** kann es zu ähnlichen Fehlinterpretationen kommen, wenn der **unsigned**-Wert außerhalb des Wertebereichs von **signed** liegt:

Demo-Programm 3_04_02.cc

```
#include <iostream.h>

int main ()
{
    short i;
    unsigned short u = 65533;
    cout << (i = u) << "\n";
    return 0;
}
```

Das Programm gibt die Zahl `-3` aus.

Konversionen zwischen Gleitkomma-Typen sind ohne Verlust von Genauigkeit und Wert möglich, solange die Konversion in eine "höhere" Type erfolgt. In umgekehrter Richtung wird die Genauigkeit auf die der Zieltype reduziert; wenn der Wertebereich des ursprünglichen Wertes außerhalb des darstellbaren Wertebereichs der Zieltype liegt, kann das Ergebnis der Konversion Null oder Unendlich (`1.#INF` oder `-1.#INF`) sein.

Bei der Konversion von Gleitkommawerten in ganzzahlige Werte wird stets der nichtganzzahlige Teil abgeschnitten (nicht gerundet!). Aus `1.9` wird daher `1`, und aus `-1.9` wird `-1`.

Bei arithmetischen Operationen mit zwei Operanden erfolgt stets eine Konversion in die "höhere" der beiden beteiligten Typen, wie in der folgenden Tabelle dargestellt:

Ein Operand vom Typ:	Zweiter Operand vom Typ:	Konversion in:
`long double`	gleichgültig	**`long double`**
`double`	gleichgültig	**`double`**
`float`	gleichgültig	**`float`**
`unsigned long`	gleichgültig	**`unsigned long`**
`long`	**`unsigned int`**	**`long`** (16 Bit) **`unsigned long`** (32 Bit)
`long`	*nicht* **`unsigned int`**	**`long`**
`unsigned int`	gleichgültig	**`unsigned int`**
alle anderen	gleichgültig	**`int`**

Zeiger beliebiger Typen können implizit in einen Zeiger der Type **`void *`** konvertiert werden. Die umgekehrte Konversion ist nur explizit möglich.

3.4.2. Explizite Umwandlungen

3.4.2.1. *Type Casts*

Diese Form der expliziten Typenumwandlung wurde aus C (aus Kompatibilitätsgründen) übernommen; sie kann in der Regel mit identischen Ergebnissen durch die Verwendung eines Typkonversions-Operators (siehe Seite 55) ersetzt werden. Ebenso wie die implizite Typenumwandlung erhalten auch *Type Casts* den Wert der Objekte, auf die sie angewendet werden (sofern dies möglich ist); sie stellen *kein* Mittel dar, etwa die binäre Darstellung einer Gleitkommavariablen zu erforschen (dazu muss diese in eine **`union`** mit einem ganzzahligen Objekt geeigneter Größe gebracht werden).

Type Casts bestehen aus dem Namen der Type in runden Klammern, der dem zu konvertierenden Objekt vorangestellt wird:

```
int i = 7;
double d;

d = (double) i;
    // Hier wäre in C++ der Type Cast eigentlich gar nicht nötig gewesen
```

Explizite Typenumwandlungen sind unbedingt erforderlich für die Konversion von Zeigern unterschiedlicher Typen.

Im folgenden Beispiel soll für einen Speicherbereich `buffer`, der irgendwelche Daten vom Typ **`char`** enthält, eine 16-Bit-Prüfsumme berechnet werden:

Demo-Programm 3_04_03.cc

```
unsigned short checksum (char *buffer, unsigned int size)
{
    unsigned short sum = 0;                  // Prüfsumme
    unsigned int i = size / sizeof(short);   // Anzahl der shorts in buffer
```

```
    while (i)
    {
        sum += *(unsigned short *) buffer;
        ((unsigned short *) buffer)++;
        i--;
    }

    return sum;
}
```

Die *Type Casts* innerhalb der **`while`**-Schleife gewährleisten, dass

- jeweils *zwei* Bytes des Puffers an der aktuellen Position des Puffer-Zeigers `buffer` als **`short int`** interpretiert und zu `sum` addiert werden, und dass
- die Adresse im Puffer-Zeiger nicht um eins, sondern um zwei erhöht wird (unter der Annahme, `sizeof(short)` sei 2).

Damit wird das gleiche Ergebnis erhalten, wie wenn `buffer` als **`unsigned short *`** `buffer` definiert worden wäre.

Beachten Sie die Klammern in der Zeile:

```
((unsigned short *) buffer)++;
```

Ohne die äußeren Klammern würde für den Inkrement-Operator ("`++`") der Zeiger `buffer` als das erscheinen, als was er definiert wurde, nämlich als **`char *`**`buffer`; die Adresse im Zeiger würde um 1 statt um 2 erhöht. Erst *anschließend* würde der Zeiger (dann aber für nichts und wieder nichts) als **`unsigned short*`** interpretiert.

Die Schleife hätte übrigens noch kürzer geschrieben werden können:

```
while (i--)
    sum += *((unsigned short *) buffer)++;
```

Demo-Programm 3_04_04.cc

Generell ist bei der Verwendung von expliziten Typkonversionen größte Vorsicht angeraten, weil sie die Typenprüfung des Compilers unterlaufen. Insbesondere die unbedachte Konversion von Zeigern kann zu unerwarteten und nur in Ausnahmefällen brauchbaren Ergebnissen führen!

3.4.2.2. Typkonversions-Operator

C++-spezifisch

Die in C++ bevorzugte Form der expliziten Typenkonversion verwendet den *Typkonversions-Operator*, der ähnlich einem Funktionsaufruf aus dem Namen einer Type und dem zu konvertierenden Objekt in einfachen Klammern besteht:

```
int i = 7;
double d;

d = double (i);
```

Der Vorteil gegenüber der Verwendung von *Type Casts* liegt darin, dass mehr als ein Argument für die Konversion verwendet werden kann. Dies erlaubt auch die Verwendung von komplexen Typen (z.B. Klassen) mit der gleichen Syntax:

```
struct Point
{
    Point (short x, short y)
      {_x = x; _y = y }                        // Konstruktor

    short _x, _y;
}

short i, j;
...

Point pt = Point (i, j);
```

3.5. Der Operator **sizeof**

Insbesondere im Zusammenhang mit Typenkonversionen von Zeigern ist es notwendig, die Größe der bearbeiteten Objekte (in Bytes) zu kennen. Diese sollte *grundsätzlich* nicht als numerische Konstante angegeben werden ("die Größe eines **int** ist 2 Bytes"), weil diese Größen implementierungsspezifisch sind (und auf 32-Bit-Maschinen die Größe eines **int** nicht 2, sondern 4 Bytes ist) und Programme, in denen die Größen von Objekten als numerische Konstanten codiert sind, nicht portabel sind. Daher sollte grundsätzlich der Operator **sizeof** verwendet werden. **sizeof** wird *immer* bei der *Übersetzung* des Programms in eine ganzzahlige Konstante mit dem korrekten Wert umgewandelt; die Verwendung dieses Operators bedeutet also in keinem Fall einen Nachteil gegenüber numerischen Konstanten. Das Resultat des Operators **sizeof**, angewendet auf den Namen einer Type in Klammern, oder auf den Namen eines Objekts ohne Klammern, ist die Größe der Type bzw. des Objekts in Einheiten der Type **char**:

```
int i = sizeof (int);            // in 16-Bit-Systemen == "int i = 2;",
                                 // in 32-Bit-Systemen == "int i = 4;"

char szHello[] = "Hello, world!";
int i = sizeof szHello;          // immer == "int i = 14;"
```

Die Verwendung von **sizeof** erlaubt die Bestimmung der Größe von Feldern (als Anzahl der Elemente):

```
int Groesse = sizeof Feld/sizeof Feld[0];
```

Der Operator **sizeof** darf nicht angewendet werden auf

- Funktionen (wohl aber auf Zeiger auf Funktionen),
- Bitfelder,
- undefinierte Klassen,
- die Type **void**.

Wenn der Operator **sizeof** auf eine Referenz angewendet wird, ist das Resultat dasselbe, wie wenn **sizeof** auf das Objekt selbst angewendet worden wäre.

4. Ausdrücke und Befehle

In diesem Abschnitt setzen wir uns mit den folgenden Baublöcken eines C++-Programms auseinander:

- Initialisierungen verschiedener Datentypen;
- Ausdrücke;
- Funktionen;
- Operatoren und
- Befehle, insbesondere für Programmverzweigungen.

4.1. Initialisierungen

4.1.1. Allgemeines

Die Initialisierung von Variablen und Objekten wird beeinflusst durch:

- Speicherklasse (statisch oder automatisch):

  Statische Objekte, also alle, die außerhalb aller Blöcke definiert wurden, und solche, die mit dem Schlüsselwort **`static`** definiert wurden, werden an einem fixen, von ihnen exklusiv verwendeten Platz im Arbeitsspeicher angelegt. Diese Objekte können in der Programmdatei mit ihren korrekten Anfangswerten abgespeichert werden. Automatische Objekte und Objekte, die mit dem Operator **`new`** aus einem Speicher-Pool allokiert werden, können im Gegensatz zu statischen Objekten nur dynamisch initialisiert werden.

- Klassen- oder Nicht-Klassen-Objekt:

  Objekte einer Klasse, für die ein *Konstruktor* definiert wurde, werden immer dynamisch unter Verwendung des Konstruktors initialisiert.

  C++-spezifisch

- Initialisierung mit einer Konstanten oder mit einem nicht-konstanten Ausdruck:

  Bei einer Initialisierung mit einer Konstanten wird (außer bei Klassen-Objekten, die einen Konstruktor haben), der korrekte Anfangswert statischer Objekte beim Laden des Programms festgelegt; ansonsten erfolgt eine dynamische Initialisierung.

Speicherklasse	Konstruktor?	Initialisierung mit Konstante?	Zeitpunkt und Art der Initialisierung
statisch	nein	nein	Vor Eintritt in den Block, in dem das Objekt erstmals gültig wird; dynamisch
automatisch	nein	nein	wie oben
statisch	ja	nein	wie oben
automatisch	ja	nein	wie oben
statisch	nein	ja	Beim Laden des Programms
automatisch	nein	ja	Vor Eintritt in den Block, in dem das Objekt erstmals gültig wird; dynamisch
statisch	ja	ja	wie oben
automatisch	ja	ja	wie oben

Die in der obigen Tabelle schattierten Kombinationen existieren in Standard-C entweder nicht, oder sie wären unzulässig.

C++-spezifisch

Alle statischen Objekte, für die kein Konstruktor existiert und die *nicht* explizit initialisiert werden, werden (vom Startup-Code des Programms) automatisch auf "0" gesetzt.

Globale Objekte, für die kein Konstruktor existiert, können grundsätzlich nur in *einem* Modul mit einer Konstanten initialisiert werden. Initialisierung in *mehreren* Modulen führt zu einem fatalen Fehler beim Zusammenbinden der Module.

C++ erlaubt jedoch auch eine Initialisierung globaler Objekte mit einem Ausdruck oder mittels des Konstruktors einer Klasse; in diesem Fall versagen die Schutzmechanismen beim Zusammenbinden. Um undefinierte Ergebnisse bei der Definition globaler Objekte zu vermeiden, sollte folgendermaßen vorgegangen werden:

C++-spezifisch

- Globale Variable sollten weitestgehend vermieden werden. Alternativen sind die Zusammenfassung global benötigter Programmdaten in einem Klassen-Objekt (Struktur oder Klasse); auch eine Definition statischer Daten mit **`static`**, die sie in ihrer Gültigkeit zumindest auf die vorliegende Übersetzungseinheit beschränkt, bietet gewissen Schutz.

➡ Globale Objekte sollten grundsätzlich in *einer* Übersetzungseinheit definiert und *explizit*, möglichst mit einer Konstanten, initialisiert werden; in allen anderen Übersetzungseinheiten sollten sie grundsätzlich mit **extern** nur *deklariert* werden (auch wenn dies vom Compiler nicht erzwungen wird).

4.1.2. Initialisierung von fundamentalen Variablen

C++-spezifisch

Objekte der in C++ eingebauten fundamentalen Typen können bei ihrer Deklaration initialisiert werden, wobei auch bei statischen Objekten (im Gegensatz zu ANSI-C) eine Initialisierung mit nicht-konstanten Ausdrücken möglich ist. Diese müssen jedoch zu dem Zeitpunkt, zu dem die Initialisierung gültig wird, definiert sein. Das folgende Programm ist daher in C++ korrekt (nicht aber in ANSI-C):

Demo-Programm 4_01_01.cc

```
#include <iostream.h>

int i = 3;    // Initialisierung mit einer Konstanten
int j = i;    // Initialisierung mit Variablen wäre in ANSI-C verboten

int main ()
{
    int k = 5;                  // Initialisierung mit einer Konstanten
    int l = 7 * i;              // Ausdruck ist auch in ANSI-C Ok

    static int m = 11;          // auch in ANSI-C Ok
    static int n = 13 * i;      // in ANSI-C verboten

    cout << "i = " << i << ", j = " << j << ", k = " << k << ", l = " << l
        << ", m = " << m << ", n = " << n << "\n";
}
```

Die Ausgabe des Programms ist, wie erwartet:

```
i = 3, j = 3, k = 5, l = 21, m = 11, n = 39
```

GNU-C/C++-spezifisch

In diesem Programm werden (vom GNU C++-Compiler) die sechs Variablen wie folgt abgelegt:

`i` Statisch und global, initialisiert mit dem Wert `3`. `i` darf in keinem anderen Modul *definiert* (wohl aber deklariert) werden.

`j` Statisch und *communal* (d.h., `j` darf in einem anderen Modul definiert und mit einer Konstanten initialisiert werden). `j` wird mit einer Routine, die *vor* der Ausführung von `main()` aufgerufen wird, auf den Wert von `i` gesetzt.

`k` Am Stack allokiert und am Anfang von `main()` auf `5` gesetzt.

`l` Am Stack allokiert und am Anfang von `main()` auf das Ergebnis von `i*7` gesetzt.

`m` Statisch, initialisiert mit dem Wert `11`. Wird behandelt wie eine globale Variable, jedoch mit einem "dekorierten" Namen, sodass eine andere Variable `m` in einer anderen Übersetzungseinheit ein unabhängiges globales Objekt darstellen würde.

`n` Statisch, initialisiert mit Hilfe einer Hilfsvariablen: Eine Hilfsvariable, die beim Laden des Programms auf `0` initialisiert wird, wird getestet; wenn sie gleich `0` ist, wird sie auf `1` gesetzt und `n` initialisiert, ansonsten wird die Initialisierung von `n` übersprungen. Diese Maßnahme sichert auch bei mehrmaligem Aufruf der Funktion eine einmalige Initialisierung.

Die Konsequenz der Behandlung einer mit einem nicht-konstanten Ausdruck initialisierten globalen (und daher statisch abgespeicherten) Variablen ist, dass diese Variable in einem anderen Modul mit einem anderen Wert initialisiert werden könnte, der aber vor dem Aufruf von `main()` — in unserem Fall `j` mit dem Wert von `i` — überschrieben wird. Eine derartige mehrfache Initialisierung wird (zumindest von GNU C++) nicht entdeckt.

Die *Initialisierung* eines Objekts ist konzeptuell von der *Zuweisung* eines Wertes verschieden; die folgenden zwei Code-Fragmente sind ihrer Philosophie nach verschieden, obwohl sie vermutlich auf allen Compilern identischen Code erzeugen:

```
int main ()
{
    int i = 5;
    ...
}

int main ()
{
    int i;
    i = 5;
    ...
}
```

In C++ werden auch bei der Initialisierung von Objekten alle erforderlichen automatischen Typkonversionen vorgenommen; `int i = 3.14;` ist daher kein Fehler (sondern identisch zu `int i = 3;`).

C++-spezifisch

4.1.3. Initialisierung von Feldern und Klassen-Typen ohne Konstruktoren

Die folgenden Aussagen gelten für Felder und für jene Klassentypen (Strukturen oder Klassen), die

- keinen Konstruktor haben;
- keine Elemente haben, die *nicht* **`public`** sind;
- keine Basisklassen und virtuellen Funktionen haben.

Für alle diese Objekte kann eine Liste von Anfangswerten (mit korrekten Typen) für alle Elemente des Objekts in geschwungenen Klammern ("**`{ }`**"), getrennt durch Kommas ("`,`"), definiert werden. Die Liste darf auch weniger Anfangswerte enthalten als Elemente im Objekt vorhanden sind; in diesem Fall werden die nicht explizit initialisierten Elemente auf "`0`" gesetzt. Es dürfen jedoch nicht *mehr* Anfangswerte angegeben werden, als Elemente vorhanden sind. Diese Form der Initialisierung wurde unverändert von Standard-C übernommen.

```
struct RCPrompt
{
    short nZeile;
    short nSpalte;
    char *szPrompt;
};

RCPrompt rcWeiter = { 24, 0, "Weiter (J/N)?" };

int nPrim[5] = { 1, 2, 3, 5, 7 };
```

Für ein eindimensionales Feld, für das eine Initialisierung vorliegt, braucht die Dimension nicht angegeben zu werden. Das obige Beispiel könnte auch geschrieben werden:

```
int nPrim[] = { 1, 2, 3, 5, 7 };
```

Mehrdimensionale Felder, Felder von Klassen, Felder innerhalb einer Klasse oder ineinandergeschachtelte Klassen-Typen sollten mit ineinandergeschachtelten geschwungenen Klammern ("**`{ }`**") initialisiert werden:

```
RCPrompt Menu[] = {
    { 4, 7, "Vorhandene Optionen:" },
    { 5, 9, "1. Hauptmenü" },
    { 6, 9, "2. Druckerausgabe" },
    { 7, 9, "3. Programm beenden" }
    };
```

(Die inneren Klammern sind syntaktisch nicht unbedingt notwendig, sie verbessern aber die Klarheit des Quellcodes. Manche Compiler wünschen sie.)

Felder vom Typ **char** können entweder durch eine Liste von Zeichenkonstanten oder durch einen String initialisiert werden:

```
char abc[4] = { 'a', 'b', 'c', 'd' };
char ABC[5] = "abcd";
```

Beachten Sie, dass im zweiten Fall das Feld wegen des abschließenden Null-Bytes um ein Element größer ist!

C++-spezifisch

Im Gegensatz zu ANSI-C kann in C++ eine Initialisierung auch für Objekte und Felder mit automatischer Speicherklasse (also für nicht-statische Objekte und Felder) erfolgen. Das folgende Beispiel ist daher in C++ korrekt, wäre aber in ANSI-C nicht zulässig:

Demo-Programm 4_01_02.cc

```
#include <iostream.h>

int main ()
{
    int ai[5] = { 3, 5, 7, 11, 13 };
    char fox[] = "The quick brown fox\n";

    for (int i = 0; i < 5; i++)
        cout << "ai [" << i << "] = " << ai[i] << "\n";

    cout << fox;
}
```

Die Ausgabe des Programms ist:

```
ai [0] = 3
ai [1] = 5
ai [2] = 7
ai [3] = 11
ai [4] = 13
The quick brown fox
```

unions können entweder mit einer **union** gleichen Typs initialisiert werden, oder mit einem Initialisierungs-Ausdruck in geschwungenen Klammern ("**{ }**") für das *erste* Element der **union**.

C++-spezifisch

4.1.4. Initialisierung von Klassen-Typen mit Konstruktoren

Konstruktoren sind Funktionen, die bei der Definition einer Klasse festgelegt werden; ihr Name ist definitionsgemäß identisch mit dem Namen der Klasse. Ein als *Function Overloading* bezeichneter Mechanismus in C++ (siehe Seite 73) erlaubt die Definition unterschiedlicher Funktionen mit gleichem Namen, die sich durch die Anzahl und/oder Type ihrer Argumente voneinander unterscheiden müssen. Wenn für eine Klasse mindestens ein Konstruktor definiert wurde, sollten auch ein *Default*-Konstruktor und ein *Kopier*-Konstruktor definiert werden; anderenfalls generiert der Compiler rudimentäre Default- und Kopier-Konstruktoren:

- Ein *Default-Konstruktor* kann ohne Argumente aufgerufen werden; seine Aufgabe ist, ein Standard-Objekt der Klasse zu generieren.
- Ein *Kopier-Konstruktor* hat ein einziges Argument, dessen Typ eine Referenz auf die selbe Klassen-Type sein muss. Er dient dazu, Klassen-Objekte zu kopieren.

De facto prüft C++ nicht, *was* ein bestimmter Konstruktor *wirklich* tut. Es sind daher auch zusätzliche oder abweichende Funktionsmechanismen möglich, was auch das folgende Beispiel illustriert:

Demo-Programm 4_01_03.cc

```
// ******** Übersetzungseinheit A: ********

#include <iostream.h>

class Point
{
public:
    Point ()                                    // Default-Konstruktor (#1):
        { _x = 13; _y = 17; }

    Point (const Point& pt)                     // Kopier-Konstruktor (#2):
        { _x = 2*pt._x; _y = 2*pt._y; }

    Point (short x, short y)                    // Konstruktor #3:
        { _x = x; _y = y; };

    Show ()
        { cout << "x = " << _x << ", y = " << _y << "\n"; }

private:
    short _x, _y;
};

void func ();                       // Vorwärts-Deklaration

extern Point pt1;                   // HIER wird kein Konstruktor aufgerufen
Point pt2;                          // Konstruktor #1
Point pt3 (3, 5);                   // Konstruktor #3

int main ()
{
    Point pt4;                      // Konstruktor #1
    Point pt5 (7, 11);              // Konstruktor #3

    pt1.Show ();
    pt2.Show ();
    pt3.Show ();
    pt4.Show ();
    pt5.Show ();

    pt4 = pt3;                      // bitweise Kopie, KEIN Konstruktor
    Point pt6 = pt3;                // Konstruktor #2

    pt4.Show ();
    pt6.Show ();

    func ();
}

// ******** Übersetzungseinheit B: ********

#include <iostream.h>

class Point
{
    // Definition der Klasse wie in Übersetzungseinheit A
};

Point pt1 (23, 29);                 // Konstruktor #3
Point pt2 (31, 37);                 // Konstruktor #3

void func (void)
{
    cout << "*****************\n";
    pt1.Show ();
    pt2.Show ();
}
```

Das Programm erzeugt die Ausgabe:

```
x = 23, y = 29                  (pt1 – aus #3, Modul B)
x = 13, y = 17                  (pt2 – aus #1, Modul A)
x = 3, y = 5                    (pt3 – aus #3, Modul A)
x = 13, y = 17                  (pt4 – aus #1, Modul A)
x = 7, y = 11                   (pt5 – aus #3, Modul A)
x = 3, y = 5                    (pt4 – Kopie von pt3)
x = 6, y = 10                   (pt6 – aus #2, Modul A)
******************
x = 23, y = 29                  (pt1 – aus #3, Modul B)
x = 13, y = 17                  (pt2 – aus #1, Modul A)
```

Anhand dieses Beispiels erkennt man die folgenden Regeln:

- Wenn bei der Definition eines Objekts keine Initialisierung angegeben wird, wird der Default-Konstruktor aufgerufen (`pt4`), ansonsten der Konstruktor mit der entsprechenden Anzahl und Type der Argumente (`pt3` oder `pt5`).
- Initialisierung und Zuweisung sind konzeptuell verschieden: für den Ausdruck

  ```
  pt4 = pt3;
  ```

  wird *nicht* der Kopier-Konstruktor aufgerufen, sondern der Inhalt von `pt3` binär auf `pt4` kopiert; `pt4` *existiert* ja schon. (Die Klasse hätte allerdings eine eigene Definition für den Operator "`=`" vorsehen können, die dann an dieser Stelle verwendet worden wäre; siehe Seite 163.)

  Dagegen wird für

  ```
  Point pt6 = pt3;
  ```

  der Kopier-Konstruktor verwendet; `pt6` existiert ja noch nicht.
- Die Objekte `pt1`, `pt2` und `pt3` sind *global*, wobei `pt1` und `pt2` auch in einer anderen Übersetzungseinheit definiert und initialisiert werden. `pt1` ist in Modul A nur *deklariert*; dieses Objekt scheint auch in beiden Modulen mit der korrekten (in Modul B erfolgten) Initialisierung auf. `pt2` hingegen ist in *beiden* Modulen (widersprüchlich) definiert; es "gewinnt" (in unserem Fall) die in Modul A erfolgte Initialisierung mit dem Default-Konstruktor.
- Unter Verwendung des Kopier-Konstruktors ist *nur* eine Initialisierung mit einem Objekt gleichen Typs möglich. Bei Klassen, die mehr als einen numerischen Parameter für ihre Initialisierung benötigen, ist daher nur eine Initialisierung in der "funktionsartigen" Syntax

  ```
  Point pt5 (7, 11);
  ```

  möglich. Eine Initialisierung unter Verwendung des Kopier-Konstruktors, etwa mit

  ```
  Point pt5 = { 7, 11 };
  ```

  würde eine Fehlermeldung bewirken.

Eine Initialisierung wie bei fundamentalen Variablen ist daher nur im folgenden Fall möglich:

Demo-Programm 4_01_04.cc

```
#include <iostream.h>

class Konto
{
public:
    Konto () { Stand = 0; }                    // Default-Konstruktor (#1)

    Konto (const double& s) { Stand = s; }     // Kopier-Konstruktor (#2)

    Konto (double s) { Stand = s; };           // Konstruktor #3:

    Show () { cout << "Kontostand: " << Stand << "\n"; }

private:
    double Stand;
};
```

```
Konto k1;                                   // Konstruktor #1
Konto k2 = 123.45;                          // Konstruktor #2
Konto k3 (234.56);                          // Konstruktor #3

int main ()
{
    Konto k4;                               // Konstruktor #1
    Konto k5 = 654.32;                      // Konstruktor #2
    Konto k6 (765.43);                      // Konstruktor #3

    k1.Show (); k2.Show (); k3.Show ();
    k4.Show (); k5.Show (); k6.Show ();
}
```

Das Programm erzeugt die Ausgabe:

```
Kontostand: 0
Kontostand: 123.45
Kontostand: 234.56
Kontostand: 0
Kontostand: 654.32
Kontostand: 765.43
```

Wenn nur *ein* Wert für die Initialisierung benötigt wird, ist also die "klassische" Initialisierung (mit "**=**") und die Initialisierung mit funktionsartiger Syntax ("**( )**") äquivalent.

4.1.5. Initialisierung von Referenzen

C++-spezifisch

Variable vom Referenz-Typ müssen bei ihrer *Definition* (nicht aber bei einer reinen *Deklaration*) grundsätzlich immer mit einem Objekt initialisiert werden, dessen Type der Basistype der Referenz entspricht, oder das sich in die Type der Referenz umwandeln lässt. Im letzteren Fall ist jedoch nur eine Initialisierung einer Referenz-Variablen mit dem Attribut **const** möglich, weil der Compiler das ursprüngliche Objekt in die gewünschte Type umwandeln und in einem internen Hilfsobjekt speichern muss; eine Änderung dieses Hilfsobjekts würde das ursprüngliche Objekt aber nicht mehr beeinflussen:

Demo-Programm 4_01_05.cc

```
#include <iostream.h>

short sVar = 3;
long lVar = 5;

int main ()
{
    short& sRef1 = sVar;                    // "normale" Referenz
    long& lRef1 = lVar;                     // "normale" Referenz

    const short& sRef2 = lVar;              // Referenzen auf Hilfsobjekte;
    const long& lRef2 = sVar;               // ohne "const" nicht zulässig

    cout << "sVar = " << sVar << ", lVar = " << lVar << "\n";

    cout << "sRef1 = " << sRef1 << ", lRef1 = " << lRef1 <<
        ", sRef2 = " << sRef2 << ", lRef2 = " << lRef2 << "\n";

    sRef1++;                                // inkrementiere sVar
    lRef1++;                                // inkrementiere lVar

    // sRef2 und lRef2 dürfen wegen "const" nicht geändert werden

    cout << "sVar = " << sVar << ", lVar = " << lVar << "\n";

    cout << "sRef1 = " << sRef1 << ", lRef1 = " << lRef1 <<
        ", sRef2 = " << sRef2 << ", lRef2 = " << lRef2 << "\n";
}
```

Die Ausgabe des Programms lautet:

```
sVar = 3, lVar = 5
sRef1 = 3, lRef1 = 5, sRef2 = 5, lRef2 = 3
sVar = 4, lVar = 6
sRef1 = 4, lRef1 = 6, sRef2 = 5, lRef2 = 3
```

Änderungen, die an `sRef1` und `lRef1` vorgenommen wurden, beeinflussen die Variablen `sVar` bzw. `lVar`, auf welche die Referenzen zeigen. Wenn `sRef2` und `lRef2` auf die selben Objekte (`sVar` bzw. `lVar`) zeigen würden, hätten sie die gleichen Werte wie `lRef1` bzw. `sRef1`. Tatsächlich zeigen aber `sRef2` und `lRef2` auf *Kopien* von `sVar` bzw. `lVar`, die nicht verändert werden können und daher auch nicht verändert wurden.

Analog werden konstante Werte, die an eine Funktion mit Referenztyp-Argumenten übergeben werden, zunächst in eine interne Hilfsvariable kopiert, deren Adresse dann an die Funktion übergeben wird:

Demo-Programm 4_01_06.cc

```
#include <iostream.h>

void func (int& iRef);

int main ()
{
    int i = 5;

    func (i);              // Referenz auf i wird übergeben
    func (7);              // Referenz auf Hilfsvariable wird übergeben

    cout << "i = " << i << "\n";
}

void func (int& iRef)
{
    iRef *= 3;             // iRef = iRef * 3;

    cout << "iRef = " << iRef << "\n";
}
```

Das Programm hat die Ausgabe:

```
iRef = 15
iRef = 21
i = 15
```

4.1.6. Programmverzweigungen und Initialisierungen

Bei der Definition von — initialisierten und uninitialisierten — Objekten ist sicherzustellen, dass eine Initialisierung immer dann erfolgt, wenn *tatsächlich* ein Zugriff auf das betreffende Objekt stattfindet:

Demo-Programm 4_01_07.cc

```
#include <iostream.h>

void func (int param);

int main ()
{
    func (0);
    func (1);
}
```

```
void func (int param)
{
    int i;

    if (param)
        i = 7;

    cout << "i = " << i << "\n";
}
```

In diesem Fall erfolgt eine Initialisierung von `i` nicht bei der Definition der Variablen, sondern durch Zuweisung, jedoch nur dann, wenn der Wert von `param` von `0` verschieden ist. Wenn also `func()` mit dem Argument `0` aufgerufen wird, ist `i` wohl *definiert*, jedoch nicht *initialisiert*.

GNU-C/C++-spezifisch

Je nach der Umgebung, in der das Programm ausgeführt wird (und je nachdem, ob und wie der Compiler den erzeugten Code optimiert), sind also andere Werte für den beim ersten Aufruf von `func()` ausgegebenen Wert von `i` zu erwarten. Wenn dieses Programm mit GNU C++ ohne Optimierung übersetzt und ausgeführt wird, ist seine Ausgabe:

```
i = 0
i = 7
```

Bei Ausführung im GNU-Debugger wird — je nach den zuvor durchgeführten Debug-Operationen — beispielsweise erhalten:

```
i = 66050
i = 7
```

Bei maximaler Optimierung generiert der GNU-C++-Compiler eine Warnung

```
test0.cc: In function 'void func(int)':
test0.cc:14: warning: 'int i' may be used uninitialized in this function
```

und setzt auf jeden Fall die Konstante 7 als Argument von `cout` ein:

```
i = 7
i = 7
```

Im obigen Beispiel ist `i` jedenfalls *definiert*, wenn auch nicht notwendigerweise *initialisiert*, unabhängig davon, welcher Weg durch das Programm genommen wird.

Im folgenden Beispiel wird die Variable `i` de facto überhaupt nicht definiert, weil der Fluss des Programms mit dem **`goto`**-Befehl um ihre Definition und Initialisierung herumgeleitet wird:

```
#include <iostream.h>

int main ()
{
    goto Umweg;

    int i = 7;

Umweg:
    cout << "i = " << i << "\n";
}
```

GNU-C/C++-spezifisch

Wenn dieses Programm mit dem GNU C++-Compiler übersetzt wird, erkennt der Compiler das Problem, meldet

```
test2.cc: In function 'int main()':
test2.cc:11: crosses initialization of 'i'
```

und erstellt kein übersetztes Programm.

4.2. Typen von Ausdrücken

- Primäre Ausdrücke:
  - Konstanten;
  - einfache Namen;
  - Ausdrücke in Klammern;
  - das Schlüsselwort **`this`**; **C++-spezifisch**
  - Operator "**`::`**", gefolgt von einfachen Namen, Operator-Namen oder qualifizierten Namen (z.B.: **`Konto::Kontostand`**).
- Postfix-Ausdrücke:
  - Feldindizes ("**`[ ]`**");
  - Funktions-Operator ("**`( )`**");
  - Explizite Typkonversion ("***`type`*`()`**"); **C++-spezifisch**
  - Elementauswahl-Operator ("**`.`**" oder "**`->`**");
  - Inkrement- und Dekrement-Operator ("**`++`**" bzw. "**`--`**").
- Ausdrücke mit *unären* Operatoren:
  - *Indirektions*-Operator (Dereferenzieren eines Zeigers) ("**`*`**");
  - Adress-Operator ("**`&`**");
  - Unäres "**`+`**" und "**`-`**";
  - Logische Negation ("**`!`**");
  - Einer-Komplement (bitweise Negation) ("**`~`**");
  - Präfix-Inkrement- und Dekrement-Operator ("**`++`**" bzw. "**`--`**");
  - **`sizeof`**-Operator;
  - **`new`**-Operator; **C++-spezifisch**
  - **`delete`**-Operator.
- Ausdrücke mit expliziter Typkonversion durch *Type Casts;* **C++-spezifisch**
- Ausdrücke mit Zeigern auf ein Element einer Klasse ("**`.*`**" und "**`->*`**");
- Ausdrücke mit binären Operatoren:
  - Multiplikation, Division und Modulo-Funktion ("**`*`**", "**`/`**" und "**`%`**");
  - Addition und Subtraktion ("**`+`**" und "**`-`**");
  - Links- und Rechtsverschiebung ("**`<<`**", "**`>>`**");
  - kleiner, kleiner und gleich, größer, größer und gleich ("**`<`**", "**`<=`**", "**`>`**", "**`>=`**");
  - gleich, ungleich (im Vergleich) ("**`==`**", "**`!=`**");
  - Bitweises UND, EX-OR und ODER ("**`&`**", "**`^`**", "**`|`**");
  - Logisches UND ("**`&&`**");
  - Logisches ODER ("**`||`**").
- Ausdrücke mit dem Operator für bedingte Programmausführung ("**`? :`**");
- Ausdrücke mit Zuweisungs-Operatoren:
  - "gewöhnliche" Zuweisung ("**`=`**");
  - arithmetische Operation am Zielausdruck ("**`*=`**", "**`/=`**", "**`%=`**", "**`+=`**", "**`-=`**", "**`<<=`**", "**`>>=`**", "**`&=`**", "**`^=`**", "**`|=`**").

4.3. *L-Values* und *R-Values*

Die Zuweisung des Ergebnisses eines Ausdruck an einen anderen Ausdruck setzt voraus, dass eine Veränderung des Wertes des zweiten Ausdrucks physikalisch und semantisch *möglich* sein muss, dass es sich also um eine *Variable* oder ein *nichtkonstantes Objekt* handeln muss. Nur solche Ausdrücke dürfen auf der *linken* Seite eines Zuweisungsoperators vorkommen, die eine andere Type als **`void`** haben, und die eine Variable beschreiben (daher "*L-Value*").

Alle anderen Ausdrücke, insbesondere Konstanten, die auf der *rechten* Seite eines Zuweisungsoperators vorkommen können, werden gelegentlich als "*R-Values*" bezeichnet. Alle *L-Values* sind auch *R-Values*, aber nicht alle *R-Values* sind *L-Values*.

Die folgende Tabelle gibt an, welche Ausdrücke in C++ *L-Values* darstellen. Die Unterschiede gegenüber Standard-C sind wieder durch eine Schattierung hervorgehoben.

C++-spezifisch

Ausdruck:	***L-Value:***
Variable, Element eines Objekts	ja
`const` Variable oder Objekt	nein
Funktion mit Nicht-Referenztyp	nein
Referenz-Typen (auch Funktionen)	ja
arithmetische oder logische Ausdrücke	nein
Ausdrücke mit dem Operator für bedingte Programmausführung	ja

4.4. Funktionen

4.4.1. Formale und aktuelle Argumente

Funktionen können eine beliebige Anzahl von *Argumenten* (oder *Parametern*) erfordern (auch keine), die innerhalb der Funktionsklammern, getrennt durch Kommas ("**,**"), aufgelistet werden müssen. Die Argumente einer Funktion, die bei ihrer Deklaration und Definition verwendet wurden, sind die *formalen* Argumente der Funktion; die Ausdrücke, die an ihrer Stelle beim *Aufruf* der Funktion stehen, sind ihre *aktuellen* Argumente. Beim Aufruf einer Funktion werden die aktuellen Argumente ausgewertet, und die korrespondierenden formalen Argumente werden entsprechend den Ergebnissen (unter Berücksichtigung impliziter oder expliziter Typkonversionen) initialisiert.

C++-spezifisch

Wichtig: Es existiert in C++ keine festgelegte Reihenfolge, in der die Argumente ausgewertet werden müssen. Es wird nur sichergestellt, dass alle Argumente ausgewertet und alle Nebenwirkungen der Auswertung abgeschlossen sind, bevor der Eintritt in die Funktion erfolgt!

Grundsätzlich werden an die Funktion *Kopien* der aktuellen Argumente übergeben, wobei Argumente vom Referenz-Typ eine Ausnahme darstellen. Die Funktion kann daher den Wert eines Arguments beliebig verändern, ohne dass dies Rückwirkungen auf die aufrufende Funktion hat. Allerdings können über Zeiger- oder Referenz-Argumente sehr wohl Objekte der aufrufenden Funktion beeinflusst werden.

Das folgende Beispiel zeigt den Unterschied zwischen den formalen und den aktuellen Argumenten einer Funktion auf und demonstriert, unter welchen Voraussetzungen die Objekte innerhalb der aufrufenden Funktion durch einen Funktionsaufruf verändert werden können:

Demo-Programm 4_04_01.cc

```
#include <iostream.h>

// Deklaration ("Prototyp") der Funktion - ";" am Ende

int myfunc (int nP, int *npP, int& nrP);

int main ()
{
    int i = 1;
    int j = 2;
    int k = 3;

    // i, &j und k sind aktuelle Argumente

    int l = myfunc (i, &j, k);

    cout << "i = " << i << ", j = " << j << ", k = " << k << ", l = " << l
        << "\n";

    return 0;
}

// Definition ("Körper") der Funktion - "{ }" am Ende
// nP, npP und nrP sind formale Argumente

int myfunc (int nP, int *npP, int& nrP)
{
    nP++;                       // inkrementiere die Kopie des Objekts
    (*npP)++;                   // inkrementiere das Original des Objekts
    nrP++;                      // inkrementiere das Original des Objekts

    cout << "nP = " << nP << ", *npP = " << *npP << ", nrP = " << nrP << "\n";

    return *npP++;  // gib Wert des Objekts zurück und inkrementiere Zeiger
}
```

Die Ausgabe dieses Programms lautet:

```
nP = 2, *npP = 3, nrP = 4
i = 1, j = 3, k = 4, l = 3
```

Die direkt als Argument übergebene Variable `i` ("*Pass by Value*") wird also in der aufrufenden Funktion nicht verändert (wohl aber der Wert des ihr entsprechenden formalen Arguments `nP`). Änderungen von Objekten, deren Adresse übergeben wurde ("*Pass by Reference*") (also als Zeiger oder über eine Referenz) sind hingegen auch in der aufrufenden Funktion wirksam (Variable `j` und `k` bzw. formale Argumente `*npP` und `nrP`). Beachten Sie die Bedeutung der Klammern im Zusammenhang mit dem Indirektions- und Inkrement-Operator (siehe Seite 83)!

4.4.2. Deklaration und Definition

4.4.2.1. Allgemeines

Die *Definition* einer Funktion erfolgt durch die Angabe ihres *Funktionskörpers*, also des eigentlichen Programmcodes, der die Funktion ausmacht. Zum Zeitpunkt des Aufrufes einer Funktion muss diese zwar *deklariert*, nicht notwendigerweise aber *definiert* sein. (Viele Funktionen werden überhaupt nicht in der aktuellen Übersetzungseinheit oder im eigentlichen Programm definiert, sondern aus Bibliotheken eingebunden.)

Die *Deklaration* einer Funktion erfolgt über *Funktions-Prototypen*, die vielfach in Header-Dateien zusammengefasst und über **`#include`**-Befehle in die Übersetzungseinheit eingebunden werden.

In der *Deklaration* (*nicht* in der *Definition*) einer Funktion ist die Verwendung *abstrakter Deklaratoren* zulässig (d.h. Typenbezeichnungen ohne Objektnamen). Die folgenden beiden Prototypen sind zulässig und (für den Compiler) äquivalent:

```
int myfunc (int nP, int *npP, int& nrP);
int myfunc (int, int *, int&);
```

Die Verwendung deskriptiver Namen für die formalen Argumente in Funktions-Prototypen trägt aber zur Lesbarkeit des Programmcodes wesentlich bei.

4.4.2.2. Funktionen ohne Argumente

Funktionen, die keine Argumente benötigen, können auf zwei äquivalente Weisen deklariert und definiert werden:

```
int func1 ();

int func1 ()
{
    ...
}
```

oder

```
int func2 (void);

int func2 (void)
{
    ...
}
```

Der Aufruf muss in jedem Fall in der Form

```
resultat1 = func1();
resultat2 = func2();
```

erfolgen (die Zuweisung des Ergebnisses einer Funktion ist aber optional).

Das Schlüsselwort **void** darf nur als einziges formales Argument in der Deklaration oder Definition einer Funktion vorkommen.

4.4.2.3. Funktionen mit variabler Argumente-Zahl

Insbesondere für Schnittstellen-Anwendungen ist es zweckmäßig, einer Funktion (z.B. für die Konsolausgabe) eine beliebige Anzahl von Argumenten übergeben zu können. In C++ sind dafür zwei Möglichkeiten vorgesehen:

➡ Standard-C-Methode: Auslassungspunkte ("...")

Auslassungspunkte am Ende einer Argumentliste geben an, dass weitere Argumente mit beliebigen Typen folgen *können*. Eine derartige Deklaration unterläuft die Typenprüfung von C++ und ist daher möglichst zu vermeiden. Die weiteren Argumente werden den folgenden automatischen Typenumwandlungen unterworfen:

- **float** → **double**;
- **char**, **short**, Aufzählungen, Bitfelder → **int** oder **unsigned int**;
- Klassen-Objekte (**struct**, **union**, **class**) werden binär kopiert als Datenstrukturen übergeben.

Beispiel für eine Funktion mit variabler Anzahl von Argumenten (siehe Seite 177):

```
extern "C" int printf (const char *, ...);
```

C++-spezifisch

➡ Voreingestellte Argumente:

Funktionsargumente, für die fast immer die gleichen Werte verwendet werden, können bei der Deklaration einer Funktion mit einer Konstanten voreingestellt werden. Dieser konstante Wert wird dann bei Funktionsaufrufen verwendet, für die kein aktuelles Argument übergeben wurde.

Demo-Programm 4_02_02.cc

```
#include <iostream.h>

void showval (int i, int j = 10, int k = 20);

int main ()
{
    showval ( 1, 2, 3 );
    showval ( 4, 5 );
    showval ( 6 );
}

void showval (int i, int j, int k)
{
    cout << "i = " << i << ", j = " << j << ", k = " << k << "\n";
}
```

Das Programm erzeugt die Ausgabe:

```
i = 1, j = 2, k = 3
i = 4, j = 5, k = 20
i = 6, j = 10, k = 20
```

Für voreingestellte Argumente gelten die folgenden Regeln:

- Sie müssen immer die letzten Elemente der Argumentliste sein.
- Sie dürfen nur genau einmal in einer Übersetzungseinheit definiert werden. Jede nochmalige Definition (auch mit dem selben Wert) wird als Fehler betrachtet.
- In späteren Deklarationen dürfen jedoch allenfalls zusätzliche voreingestellte Argumente am Ende der Liste angefügt werden.
- Voreingestellte Argumente sind auch bei der Deklaration von Zeigern auf Funktionen zulässig:

```
int (*fpFunc)(int i = 0);
```

4.4.3. *Function Overloading*

C++-spezifisch

Eine als *Function Overloading* bezeichneter Mechanismus von C++ erlaubt die Definition von Funktionen mit gleichen Namen, aber unterschiedlichen Argumentsätzen. Damit ist es möglich, gewisse (fundamentale) Funktionen unabhängig von der Zahl und Type der Argumente durch einen *einzigen* Namen zu repräsentieren. (Die interne Unterscheidung zwischen den verschiedenen für die unterschiedlichen Argumentsätze benötigten Funktionen erfolgt durch das *Dekorieren* des Funktionsnamens im compilergenerierten Object-Code.)

Das folgende Beispiel demonstriert *Function Overloading* anhand eines Programms zur Ausgabe von Strings, ganzen Zahlen und Gleitkommazahlen mit einer generischen Funktion `print`:

Demo-Programm 4_04_03.cc

```
#include <stdio.h>                // Standard-C-Ein-/Ausgabe-Funktionen

                                  // Separate Deklarationen:
void print (char *);              // String-Ausgabe
void print (int);                 // int-Ausgabe
void print (double);              // double-Ausgabe

int main ()
{
    print ("Hello, world!");
    print (4711);
    print (3.141592);
}

void print (char *buffer)         // print-Funktion für Strings
{
    puts (buffer);                // Standard-C-String-Ausgabe
}

void print (int i)                // print-Funktion für ganze Zahlen
{
    printf ("%i\n", i);           // Standard-C formatierte Ausgabe
}

void print (double x)             // print-Funktion für Gleitkomma-Zahlen
{
    printf ("%lf\n", x);          // Standard-C formatierte Ausgabe
}
```

Das Programm erstellt, wie gewünscht, die Ausgabe:

```
Hello, world!
4711
3.14159200000000
```

Der Compiler erzeugt drei verschiedene Funktionsnamen für die drei Funktionen mit identischen Quellnamen. Mit dem GNU C++-Compiler erhält man:

```
_print__FPc          für    void print (char *);
_print__Fi           für    void print (int);
```

GNU-C/C++-spezifisch

`_print__Fd` für `void print (double);`

Funktionen, die den Mechanismus des *Function Overloading* verwenden sollen, müssen sich in mindestens einem der folgenden Charakteristika voneinander unterscheiden:

- Anzahl der Argumente;
- Typen der Argumente;
- Vorhandensein oder Fehlen von Auslassungspunkten ("`...`");
- **`const`**, **`volatile`** oder implementierungsspezifische Attribute in der Argumentliste (z.B. **`__near`**, **`__far`**).

Die folgenden Charakteristika können *nicht* für die Unterscheidung zweier Funktionen im Zusammenhang mit *Function Overloading* verwendet werden. Wenn zwei Funktionen mit gleichem Namen definiert werden, die sich *nur* durch eine oder mehrere der folgenden Charakteristika voneinander unterscheiden, so stellt dies einen Fehler dar:

- Typ des *Resultats* der Funktion;
- Verwendung von Namen, die mit **`typedef`** definiert wurden, und die gleiche Basistypen ergeben;
- Unspezifizierte Feldgrößen.

4.4.4. Funktionen, Makros und Nebenwirkungen

Bei der Ermittlung der aktuellen Argumente einer Funktion können beabsichtigte oder unbeabsichtigte *Nebenwirkungen* auftreten, beispielsweise dann, wenn Variable der aufrufenden Funktion in ihrem Wert verändert oder neue Daten eingelesen werden:

```
#include <conio.h>

int func1 (int);
char func2 (char);

int main ()
{
    int i = 1, j;
    char ch;

    j = func1 (i++);
        // Wert von i wird als aktuelles Argument verwendet, und i wird
        // anschließend inkrementiert

    ch = func2 (getch());
        // getch liest ein Zeichen von der Konsole ein (Standard-C-
        // Bibliotheksfunktion)
}
```

Beim Aufruf von *Funktionen* wird garantiert, dass jedes aktuelle Argument *genau einmal* ausgeführt wird, d.h., `i` wird *genau einmal* inkrementiert, und `getch()` *genau einmal* aufgerufen.

Dies gilt auch für **`inline`**-Funktionen. Im Gegensatz dazu werden beim Aufruf von *Makros* die Argumente des Makros unter Umständen auch öfter als einmal oder überhaupt nicht ausgewertet:

```
// Makro für die Konversion von Klein- in Großbuchstaben;
// Standardausführung:

#define toupper(c) ((islower(c)) ? ((c)-'a'+'A') : (c))
    // islower(c) ist ein Makro, das einen Wert != 0 ergibt, wenn sein
    // Argument zwischen 'a' und 'z' liegt. In diesem Fall erfolgt eine
    // Umwandlung in Großbuchstaben ((c)-'a'+'A')
```

Beim Aufruf des Makros werden alle Stellen, an denen eines seiner formalen Argumente vorkommt (hier "`(c)`") durch das aktuelle Argument ersetzt. Der Aufruf

```
char ch = toupper (getch());
```

wird daher umgesetzt in

```
ch = ((islower(getch())) ? ((getch())-'a'+'A') : (getch()))
```

`getch()` wird einmal bei der Auswertung von `islower(getch())` aufgerufen, und ein zweites Mal bei der bedingten Ausführung von `(getch())-'a'+'A'` bzw. `getch()`. Statt *eines* Zeichens fordert das Programm also zwei an. Die Definition von `toupper(c)` in GNU C++ vermeidet dieses Problem:

GNU-C/C++-spezifisch

```
#define toupper(c) ({int _c=(c); islower(_c) ? (_c-'a'+'A') : _c; })
```

Beachten Sie bitte die Klammern um die Argumente eines Makros in seiner Definition! Sie sollen sicherstellen, dass ein allenfalls als Argument des Makros verwendeter Ausdruck vollständig ausgewertet wird, bevor er vom Makro übernommen wird:

```
#define CONST1    10                          // Konstanten über
#define CONST2    2                           // Makros definiert

#define FALSCH    CONST1 + CONST2             // falsche Definition
#define RICHTIG   (CONST1 + CONST2)           // korrekte Definition

void *ptr = malloc (FALSCH*sizeof (long));
    // wird ausgewertet als:
    //     void *ptr = malloc (CONST1 + CONST2*sizeof (long));
    // Es werden also 10 + 2*4 = 18 Bytes an Speicher allokiert

void *ptr = malloc (RICHTIG*sizeof (long));
    // wird ausgewertet als:
    //     void *ptr = malloc ((CONST1 + CONST2)*sizeof (long));
    // Es werden also (10+2)*4 = 48 Bytes an Speicher allokiert
```

Die beschriebenen Nebenwirkungen und Gefahren unterbleiben, wenn anstelle von Makros die C++-Mechanismen von **`inline`**-Funktionen und **`const`**-Typen verwendet werden.

C++-spezifisch

4.5. Operatoren

4.5.1. Arithmetische Operatoren

4.5.1.1. Inkrement- und Dekrement-Operatoren

Inkrement-Operatoren ("++") bewirken die Erhöhung des Wertes der Variablen, auf die sie angewendet werden, um eine Einheit der entsprechenden Type; Dekrement-Operatoren ("--") bewirken eine Verringerung um eine Einheit. (Das muss nicht immer der Wert `1` sein; der Wert eines Zeigers auf **`long`** ändert sich beispielsweise um `4`, weil `sizeof(long) == 4` ist.) In C/C++ existieren zwei Typen von Inkrement- und Dekrement-Operatoren, die sich durch ihre Funktion und ihre *Präzedenz* (*Operator Precedence*; siehe Seite 13 und 83) voneinander unterscheiden:

➡ Postfix-Inkrement und -Dekrement:

Diese Operatoren erhöhen bzw. verringern den Wert der Variablen, auf die sie angewendet werden, *nachdem* deren *ursprünglicher* Wert in einer allfälligen anderen Operation verwendet wurde:

```
int i = 1, j;
j = i++;          // j wird auf 1 gesetzt; i wird anschließend auf 2 erhöht
```

Die Postfix-Inkrement- und Dekrement-Operatoren haben von allen arithmetischen Operatoren die höchste Präzedenz, d.h., sie beziehen sich immer auf das ihnen unmittelbar voranstehende Objekt.

➡ Präfix-Inkrement und Dekrement:

Diese Operatoren erhöhen bzw. verringern den Wert der Variablen, auf die sie angewendet werden, *bevor* ihr Wert in einer allfälligen anderen Operation verwendet wird:

```
int i = 1, j;
j = ++i;          // i wird auf 2 erhöht; j wird auf 2 gesetzt
```

4.5.1.2. Verschiebungs-Operatoren

Die Operatoren "<<" und ">>" bewirken eine bitweise Verschiebung des linken Operanden um die im rechten Operanden angegebene Anzahl von Bits nach links bzw. rechts. Beide mit diesen Operatoren verwendeten Operanden müssen ganzzahlig sein. Negative rechte Operanden oder rechte Operanden, die größer sind als die Bitzahl des linken Operanden, haben undefinierte Ergebnisse zur Folge.

Bei der Linksverschiebung mit "<<" werden die freiwerdenden niederwertigen Bits mit "0" aufgefüllt. Eine Linksverschiebung um n Bits entspricht einer Multiplikation mit 2^n.

Bei der Rechtsverschiebung mit ">>" werden die freiwerdenden höherwertigen Bits je nach der Deklaration des linken Operanden aufgefüllt: Bei linken Operanden, die als **`unsigned`** deklariert wurden, werden diese Bits immer mit "0" aufgefüllt (*logische Verschiebung*); bei **`signed`** Operanden wird der Wert des höchstwertigen (äußerst linken) Bits dupliziert (*arithmetische Verschiebung*). Damit entspricht eine Rechtsverschiebung um n Bits bei CPUs, die mit Zweier-Komplement-Darstellung negativer Zahlen arbeiten, in allen Fällen einer Division durch 2^n.

Bits, die aus dem Bereich des linken Operanden nach links oder rechts herausgeschoben werden, sind verloren; ihr Wert hat keinen Effekt auf nachfolgende Operationen.

4.5.2. Vergleichs-Operatoren

Die Operatoren "<", "<=", ">", ">=", "==" und "!=" dienen zum Vergleich ihrer linken und rechten Operanden. Diese werden nach den in C++ üblichen Regeln in kompatible Typen umgewandelt, bevor der Vergleich durchgeführt wird. Das Ergebnis einer Vergleichs-Operation ist immer 1, wenn der Vergleichs-Ausdruck eine wahre Aussage darstellt; ansonsten ist es 0. Das Ergebnis der Vergleichs-Operation ist immer vom Typ **int**. Häufig wird für die Manipulation logischer Aussagen eine Type BOOL definiert:

```
typedef int BOOL;
```

4.5.3. Bitweise und logische Operatoren

Die Funktion der *bitweisen* und der *logischen* Operatoren ist trotz mancher Ähnlichkeiten deutlich unterschiedlich:

- Die *bitweisen* Operatoren ("~" — NOT, "&" — AND, "^" — EX-OR und "|" — OR) beeinflussen *alle* Bits ihrer Operanden. Die *binären* Operatoren "&", "^" und "|" verknüpfen *jedes* Bit ihres ersten Operanden mit dem korrespondierenden Bit des zweiten Operanden; die bitweise Negation "~" invertiert *alle* Bits. Es gilt daher:

  ```
  ~0x5a5a == 0xa5a5;
  0x5555 & 0x00ff == 0x0055;
  0x5555 ^ 0x00ff == 0x55aa;
  0x5555 | 0x00ff == 0x55ff;
  ```

- Die *logischen* Operatoren ("!" — NOT, "&&" — AND und "||" — OR) interpretieren ihre Operanden als Boole'sche Variable, wobei alle Werte ungleich 0 als TRUE und der Wert 0 als FALSE interpretiert werden. Man kann dieses Verhalten dazu verwenden, um Ergebnisse zu "säubern", die als Boole'sche Variable verwendet werden sollen:

  ```
  typedef int BOOL;
  char *buffer;
  BOOL i = (*buffer) && 1;
      // ist dann == 1, wenn buffer Daten enthält, sonst == 0
  ```

4.5.4. Zuweisungs-Operatoren

Zuweisungs-Operatoren ("=" sowie "***op***=") weisen ihrem linken Operanden das Ergebnis eines Ausdrucks zu; der Wert eines Ausdrucks mit dem Zuweisungs-Operator ist identisch mit dem Wert des linken Operanden nach der Zuweisung. Der linke Operand einer Zuweisungsoperation muss immer ein *L-Value* sein. Der folgende Ausdruck ist daher illegal:

```
(a += b) += c;              // Fehler: (a += b) ist kein L-Value
a += (b += c);              // Ok
```

Die Verbund-Zuweisungsoperatoren "***op***=" weisen dem linken Operanden das Ergebnis der mit dem Operator *op* und dem rechten Operanden durchgeführten Operation zu. Ein Ausdruck der Form

```
a op= b;
```

ist also äquivalent zu

```
a = a op b;
```

op kann jeder binäre arithmetische und bitweise Operator sein.

4.5.5. Der Operator für bedingte Ausführung

Der Operator "**? :** " ist der einzige *ternäre* Operator in C/C++. Er hat die folgende Funktion:

- Der erste Operand (vor dem "**?**") wird ausgewertet, und alle Nebenwirkungen dieses Ausdrucks werden abgeschlossen.
- Wenn das Ergebnis des ersten Operanden ein von Null verschiedener Wert war, wird der zweite Operand ausgewertet.
- Wenn das Ergebnis des ersten Operanden Null war, wird der dritte Operand ausgewertet.

Das *Ergebnis* eines bedingten Ausdrucks ist das Ergebnis jenes Operanden, der ausgewertet wurde. Die folgenden zwei Ausdrücke sind daher äquivalent:

```
x = (i > 0) ? 5 : 3;

i > 0 ? (x = 5) : (x = 3);
    // Klammern um "(x = ...)" wegen der Operator Precedence!
```

Der erste Operand muss von einer ganzzahligen oder Zeiger-Type sein; der zweite und dritte Operand müssen kompatible Typen haben. Die folgenden Kombinationen sind legal, wobei es keine Rolle spielt, ob der zweite Operand von Type 1 und der dritte von Type 2 ist oder umgekehrt. Alle in der nachfolgenden Tabelle *nicht* angeführten Kombinationen sind illegal:

Type 1:	**Type 2:**	**Resultat:**
gleiche Type		gleiche Type
arithmetische Type		Type aus Standard-Konversion
Zeiger	Zeiger oder NULL	Type aus Zeigerkonversion
Referenz-Type		Type aus Referenz-Konversion
void	**void**	**void**
gleiche Klasse		gleiche Klasse

C++-spezifisch

Im Gegensatz zu ANSI-C sind in C++ Ausdrücke mit dem Operator für die bedingte Ausführung *L-Values*. Der folgende Ausdruck wird in C++ akzeptiert, würde aber in ANSI-C eine Fehlermeldung zur Folge haben:

```
((a < 5)? b : c) = 9;      // äquivalent zu: a < 5 ? (b = 9) : (c = 9);
```

C++-spezifisch

4.5.6. Die Operatoren **new** und **delete**

Die beiden Operatoren **new** und **delete** dienen zur Verwaltung von Speicherblöcken, die dynamisch (zur Laufzeit) aus einem als "*Free Store*" bezeichneten Speicherbereich (entspricht dem *Heap* in C) bezogen werden können. Sie ersetzen damit die in C für den gleichen Zweck verwendeten Bibliotheksfunktionen `malloc()` bzw. `free()`.

4.5.6.1. Der Operator **new**

Dieser Operator bezieht einen Speicherblock, dessen Größe durch die Größe der als sein Operand angegebenen Type bestimmt ist, aus dem *Free Store,* und gibt als Ergebnis einen Zeiger auf diesen Block zurück, dessen Type durch die Type des Operanden festgelegt ist:

- Wenn **new** verwendet wird, um Speicherplatz für eine Nicht-Feld-Type zu erhalten, ist das Ergebnis von **new** ein Zeiger auf diese Type:

```
double *pdVariable = new double;
class A;
```

```
A *pA = new A;
```

➡ Wenn mit **new** Speicherplatz für ein eindimensionales Feld bezogen wird, ist das Ergebnis ein Zeiger auf das erste Element des Feldes:

```
char *sBuffer = new char[80];
```

➡ Wenn **new** verwendet wird, um Platz für ein *mehrdimensionales* Feld (Dimension *n*) zu erhalten, ist das Ergebnis ein Zeiger auf das erste Element des Feldes mit dem Typ "Zeiger auf ein *n–1*-dimensionales Feld, dessen Dimensionen gleich der zweiten bis *n*-ten Felddimension sind":

```
int (*piaA)[6][8] = new int[4][6][8];
    // VORSICHT: "int (*piaA)[6][8]" ist ein Zeiger auf ein Feld von 6*8
    // ints."int *piaA[6][8]" wäre ein Feld von 6*8 Zeigern auf int.
```

Offensichtlich infolge eines Compilerfehlers funktioniert das obige Konstrukt unter GNU C++ *nicht*. Korrekte Funktion konnte unter GNU C++ mit dem folgenden Code erhalten werden:

GNU-C/C++-spezifisch

```
int (*piaA)[6][8];
void *pv;

int main ()
{
    pv = new int[4][6][8];
        // implizite Konversion in Zeiger auf void

    piaA = (int (*)[6][8]) pv;
        // explizite Konversion in Zeiger auf Feld (implizite Konversion
        // ist unzulässig)

    int i = piaA[2][3][4];
}
```

Der Hilfs-Zeiger pv kann auch vermieden werden:

```
int (*piaA)[6][8];

int main ()
{
    piaA = (int(*)[6][8])new int[4][6][8];
        // explizite Konversion in Zeiger auf Feld

    int i = piaA[2][3][4];
}
```

➡ Wenn unzureichender Speicherplatz vorhanden ist, ist das Resultat von **new** Null:

```
#include <iostream.h>

int main ()
{
    int *pi = new int[32000];

    if (! pi)                          // oder: "if (pi == 0)"
    {
        cout << "Zu wenig Speicher\n"

        return 1;              // Programm beenden mit Fehlercode 1
    }

    ...
}
```

Bei komplexeren mit **new** zu allokierenden Datentypen kann es notwendig sein, die Typenbezeichnung hinter "**new**" in Klammern zu setzen:

```
// Feld von 10 Funktionszeigern auf Funktionen vom Typ int anlegen,
// die keine Argumente übernehmen:

int (*pFFeld)() = new int (*[10])();
    // FALSCH - würde interpretiert als "(new int) (*[10])()"
```

GNU-C/C++-spezifisch

```
int (*pFFeld)() = new (int (*[10])());
    // Korrekt (funktioniert in GNU C++ trotzdem nicht)
```

Objekte, die mit **new** erstellt wurden, bleiben bis zum Aufruf von **delete** für das betreffende Objekt bzw. bis zum Programmende bestehen. Es ist für eine korrekte Speicherverwaltung daher notwendig, das Ergebnis von **new** einer Zeigervariablen von ausreichender Lebensdauer zuzuweisen.

Mit **new** erstellte Objekte können gleichzeitig initialisiert werden. Für Klassen-Objekte wird ein für die Initialisierung geeigneter Konstruktor aufgerufen. Für die Initialisierung können beliebige Ausdrücke (nicht nur Konstanten) verwendet werden.

Die Initialisierung von Klassen-Objekten, die einen Konstruktor haben, ist mit **new** nur möglich, wenn eine der beiden folgenden Bedingungen erfüllt ist:

- Die Type und Anzahl der Argumente der Initialisierung stimmen mit denen eines Konstruktors überein; oder
- Die Klasse besitzt einen Default-Konstruktor (ohne Argumente).

Demo-Programm 4_05_01.cc

```
#include <iostream.h>

class POINT
{
public:
    POINT (void)                  // Default-Konstruktor ohne Argumente
        { _x = 0; _y = 0; }

    POINT (short x, short y)      // Overloaded Konstruktor mit Argumenten
        { _x = x; _y = y; }

    Zeige (void)
        { cout << "x = " << _x << ", y = " << _y << "\n"; }

private:
    short _x, _y;
};

int main ()
{
    int *i = new int (13);

    cout << "i = " << *i << "\n";

    POINT *pp1 = new POINT;            // Zeiger auf Objekt, das auf (0, 0)
    pp1->Zeige();                      // initialisiert ist
    POINT p1;                          // Objekt, initialisiert auf (0, 0)
    p1.Zeige();

    POINT *pp2 = new POINT (11, 13);   // Zeiger auf Objekt, das auf  (11, 13)
    pp2->Zeige();                      // initialisiert ist
    POINT p2(17, 19);                  // Objekt, initialisiert auf (17, 19)
    p2.Zeige();
}
```

Das Programm erzeugt die Ausgabe

```
i = 13
x = 0, y = 0
x = 0, y = 0
x = 11, y = 13
x = 17, y = 19
```

Beachten Sie im obigen Beispiel den Unterschied zwischen der Definition von *Objekten* (`p1` und `p2`) und von der Definition von *Zeigern*, die auf Objekte zeigen, die vom *Free Store* allokiert wurden (`pp1` und `pp2`): `p1` und `p2` werden in unserem Fall als automatische Variable mit Block-Lebensdauer auf dem Stack des Programms angelegt; wenn wir für ihre Definition das Schlüsselwort **`static`** verwendet hätten, wäre das gesamte Objekt im Datenbereich des Programms (mit Programm-Lebensdauer) angelegt worden. Im Gegensatz dazu werden mit `pp1` und `pp2` zwei *Zeiger*-Variable angelegt, die auf einen dynamisch verwalteten Speicherbereich zeigen, in dem sich die eigentlichen Objekte befinden (die hier namenlos sind). Die Verwendung von **`new`** ist dann besonders zweckmäßig, wenn Objekte temporär, aber unter Umständen über die aktuelle Funktion hinaus benötigt werden, deren Größe dagegen spricht, dass sie auf dem Stack angelegt werden können, und die als statische Objekte mit Programm-Lebensdauer eine ineffiziente Ausnützung des Arbeitsspeichers bedeuten würden.

Die Initialisierung von Feldern mit **`new`** ist nicht möglich; es wird nur (sofern vorhanden) der Default-Konstruktor aufgerufen.

Wenn eine Zuordnung von Speicher aus dem *Free Store*-Bereich nicht möglich ist, kann auch keine Initialisierung erfolgen. Die Ausdrücke, die für die Initialisierung verwendet wurden, werden in diesem Fall nicht notwendigerweise ausgeführt; ebenso ist die Reihenfolge nicht definiert, in der Initialisierungs-Ausdrücke auch im Normalfall ausgewertet werden.

Die Standard-**`new`**-Funktion kann explizit mit

```
::operator new (sizeof (type))
```

aufgerufen werden. Wenn eine Klassen-Type eine **`operator new`**-Funktion definiert hat (siehe Seite 168), wird für diese Type

```
type::operator new (sizeof (type))
```

aufgerufen.

Optional können benutzerdefinierte Klassentypen eine beliebige Anzahl weiterer Argumente (zusätzlich zu `sizeof(type)`) definieren; der Aufruf einer solchen beispielsweise als

```
type::operator new (sizeof (type), par1, par2, par3)
```

definierten **`operator new`**-Funktion kann dann etwa so erfolgen:

```
type *ptype = new (par1, par2, par3) type;
```

Selbst dann, wenn für eine Type eine **`operator new`**-Funktion definiert wurde, kann unter Verwendung des Operators "**`::`**" die Standard-**`operator new`**-Funktion aufgerufen werden:

```
type *ptype = ::new type;
```

4.5.6.2. Der Operator **`delete`**

Der Speicherbereich, der mit **`new`** für die Erzeugung von Objekten allokiert wurde, sollte dann, wenn er nicht mehr benötigt wird, mit **`delete`** dem Programm wieder zur Verfügung gestellt werden. **`delete`** hat eine Resultat-Type **`void`** (d.h., der Operator *hat* kein Resultat); als Operand wird ein mit **`new`** generierter Zeiger, also ein Zeiger auf ein mit **`new`** allokiertes Objekt, benötigt. Das Ergebnis, wenn **`delete`** *nicht* auf einen derartigen Zeiger angewendet wird, ist unbestimmt. **`delete`** darf jedoch auf einen Zeiger mit dem Wert Null angewendet werden; der von **`new`** zurückgegebene Wert darf also auch dann an **`delete`** gegeben werden, wenn die Speicherzuweisung misslungen ist.

Nachdem ein Zeiger als Operand für **`delete`** verwendet wurde, darf er nicht mehr dereferenziert oder sein Wert anderweitig verwendet werden.

Wenn **`delete`** mit einem Zeiger auf ein Objekt einer Klasse als Operand verwendet wird, für die ein Destruktor definiert wurde, wird dieser Destruktor aufgerufen, bevor der Speicher zurückgegeben wird. Eine eigene Definition von **`operator delete`** in einer Klasse ist möglich.

Für Felder, die mit **`new`** allokiert wurden, (und nur für diese) muss **`delete`** mit dem Operator "**`[]`**" (als "**`delete[]`**") aufgerufen werden:

Demo-Programm 4_05_02.cc

```
#include <iostream.h>

int main ()
{
    int *pi1 = new int (3);                    // einfaches Objekt
    double *pd1 = new double [5];              // Feld von 5 doubles

    *pd1 = 1.1;                                // Initialisierung
    *(pd1 + 1) = 2.2;
    *(pd1 + 2) = 3.3;
    *(pd1 + 3) = 4.4;
    *(pd1 + 4) = 5.5;

    cout << "pi1 = " << *pi1 << "\n";
    for (int i = 0; i < 5; i++)
        cout << "pd1[" << i << "] = " << pd1[i] << "\n";

    delete pi1;                                // pi1 zerstören
    delete [] pd1;                             // Feld zerstören

    // Free Store-Speicher wird "recycled" und neu vergeben:

    int *pi2 = new int (17);
    double *pd2 = new double [5];

    *pd2 = 11.1;
    *(pd2 + 1) = 22.2;
    *(pd2 + 2) = 33.3;
    *(pd2 + 3) = 44.4;
    *(pd2 + 4) = 55.5;

    // das Folgende ist eigentlich unzulässig:

    cout << "pi1 = " << *pi1 << "\n";
    for (i = 0; i < 5; i++)
        cout << "pd1[" << i << "] = " << pd1[i] << "\n";
}
```

Das Programm erzeugt die Ausgabe:

```
pi1 = 3
pd1[0] = 1.1
pd1[1] = 2.2
pd1[2] = 3.3
pd1[3] = 4.4
pd1[4] = 5.5
pi1 = 17
pd1[0] = 11.1
pd1[1] = 22.2
pd1[2] = 33.3
pd1[3] = 44.4
pd1[4] = 55.5
```

In diesem Demo-Programm wird zunächst eine **int**-Variable und ein Feld von **double**s mit **new** allokiert und initialisiert. Der für diese Objekte verwendete Speicher wird anschließend mit **delete** "recycled" und für eine neue **int**-Variable und ein neues Feld von **double**s verwendet. Zu diesem Zeitpunkt sind die Zeigervariablen `pi1` und `pd1` wohl noch vorhanden, aber *semantisch ungültig*, weil die Objekte, auf die sie zeigen, nicht mehr existieren. Im gleichen Speicherbereich existieren nunmehr die Objekte, auf die `pi2` bzw. `pd2` zeigen. Dereferenzieren von `pi1` und `pd1` ergibt für unser Demo-Programm (nicht ganz zufällig) die Werte der Objekte `pi2` und `pd2`; es könnte aber auch einen Programmabsturz bewirken.

4.5.6.3. **new** und **delete** im Vergleich zu malloc() und free()

Wie erwähnt, übernehmen **new** und **delete** in C++ die Aufgaben der C-Funktionen malloc() und free(). Ebenso wie die **operator new**-Funktion benötigt auch malloc() ein Argument vom Typ size_t (der üblicherweise durch **typedef** als Synonym für **unsigned** erhalten wird) und gibt einen Zeiger, in diesem Fall vom Typ **void ***, zurück. free() benötigt einen Zeiger auf einen mit malloc() allokierten Speicherblock (oder einen Null-Zeiger). Die wesentlichen Unterschiede zwischen den C-Funktionen und den C++-Operatoren sind:

- **new** und **delete** sind integraler Bestandteil der Sprache.
- **new** ermöglicht eine Typ-Prüfung seines Ergebnisses, während malloc() immer einen *Type Cast* auf einen entsprechend typisierten Zeiger erfordert.
- **new** und **delete** rufen automatisch den Konstruktor bzw. Destruktor einer Klasse auf.
- Eine Initialisierung des allokierten Speicherblocks ist mit **new** (außer bei Feldern) möglich, während malloc() nur uninitialisierten Speicher verwaltet.

Eine gemischte Verwendung von **new** und malloc() ist möglich.

4.5.7. Präzedenz von Operatoren

Die Präzedenz der C++-Operatoren (*Operator Precedence*) wird nach ihrer Reihung in der Tabelle auf Seite 13 festgelegt, wobei Operatoren, die in einer doppelt umrandeten Gruppe aufscheinen, die gleiche Präzedenz haben. Operatoren, die in einem Ausdruck "gleichberechtigt" nebeneinander stehen, werden in der Reihenfolge ihrer Präzedenzen abgearbeitet. Durch Verwendung von Klammern kann diese Reihenfolge modifiziert werden; Klammerausdrücke werden immer zuerst ausgewertet. Dies gilt sowohl für arithmetische Operationen als auch für Operationen wie Indirektion oder Elementauswahl.

Die folgenden Beispiele illustrieren dies:

```
int i = 2 + 3*4;                // ergibt i == 14;
int j = (2 + 3)*4;              // ergibt i == 20;
```

Im Zweifelsfall ist es günstiger, zu viele Klammern zu verwenden als zu wenige. Auch an sich legale Konstruktionen können ohne Klammern, die die Lesbarkeit unterstützen, ziemlich undurchsichtig wirken:

```
#include <iostream.h>

int main ()
{
    int i = 10 < 20 < 5;

    cout << "i = " << i << "\n";
}
```

erzeugt die Ausgabe

```
i = 1
```

weil der Ausdruck wegen der *Assozivität* der Operatoren "**<**" von links nach rechts interpretiert wird als

```
(10 < 20) < 5
```

und 10 < 20 als wahrer Ausdruck 1 ergibt, was kleiner als 5 ist.

Der Postfix-Inkrement-Operator hat eine *höhere* Präzedenz als der Indirektions-Operator; der Präfix-Inkrement-Operator hat die gleiche Präzedenz wie der Indirektions-Operator. Damit haben die folgenden Ausdrücke — je nach Vorhandensein und Position von Klammern — die folgenden Bedeutungen:

```
char *bufptr;       // Zeiger in einem Puffer mit Text
char ch;
```

Ausdruck:	Bedeutung:
`ch = *bufptr;`	`ch` = Zeichen, auf das `bufptr` zeigt.
`bufptr++;`	Zeige auf das nächste Zeichen im Puffer.
`++bufptr;`	wie oben
`ch = *bufptr++;`	`ch` = Zeichen, auf das `bufptr` zeigt; `bufptr` wird anschließend inkrementiert und zeigt auf das nächste Zeichen.
`ch = *(bufptr + 1);`	`ch` = Zeichen hinter jenem, auf das `bufptr` zeigt.
`ch = *bufptr + 1;`	`ch` = Zeichen, auf das `bufptr` zeigt, um 1 inkrementiert.
`ch = ++(*bufptr);`	`ch` = Zeichen, auf das `bufptr` zeigt, wird im Puffer um 1 inkrementiert und anschließend an `ch` zugewiesen.
`ch = ++*bufptr;`	wie oben
`ch = (*bufptr)++;`	`ch` = Zeichen, auf das `bufptr` zeigt; wird nach der Zuweisung im Puffer um 1 inkrementiert.
`ch = *(++bufptr);`	`bufptr` zeigt auf das nächste Zeichen; dieses wird `ch` zugewiesen.
`ch = *++bufptr;`	wie oben

Insbesondere dann, wenn nicht-arithmetische Operatoren mit arithmetischen gemischt werden, können unerwartete Effekte auftreten:

```
cout << 3 < 2 << "\n";
```

gibt nicht, wie erwartet, den Wert `0` aus, sondern hat eine Compiler-Fehlermeldung zur Folge. Wegen der höheren Präzedenz von "`<<`" gegenüber "`<`" interpretiert der Compiler den obigen Ausdruck als:

```
(cout << 3) < (2 << "\n");
```

Korrekt hätten wir schreiben müssen:

```
cout << (3 < 2) << "\n";
```

4.5.8. Kombinationen von Attributen

Die folgende Tabelle gibt die gültigen Kombinationen der Attribute "**`const`**" und "**`volatile`**" mit der Type ***`type`*** eines Operanden in einem Ausdruck an. Diese Regeln können auch miteinander kombiniert werden; beispielsweise ist ein Zeiger der Type **`const`** ***`type`*`*`** **`volatile`** überall dort zulässig, wo ein Zeiger erwartet wird.

Erwartete Type:	Zulässige Typen:	
type	*type* const *type* volatile *type* volatile const *type*	*type*& const *type*& volatile *type*& volatile const *type*&
*type**	*type** *type** const	*type** volatile *type** volatile const
const *type*	*type* const *type*	const *type*&
volatile *type*	*type* volatile *type*	volatile *type*&

4.6. Befehle (*Statements*)

4.6.1. Befehle mit Ausdrücken

Der einfachste Typ von Befehlen in C/C++ besteht aus einem Ausdruck, der mit einem Strichpunkt ("**;**") abgeschlossen ist. Die Ausdrücke werden der Reihe nach abgearbeitet; mit der Bearbeitung des nächsten Ausdrucks wird erst begonnen, wenn der aktuelle Ausdruck und alle seine Nebenwirkungen zur Gänze abgeschlossen sind:

```
// Zwei Variable miteinander vertauschen

void swap (double& a, double& b)
{
    double temp = a;
    a = b;
    b = temp;
}

// Cartesische in Polarkoordinaten umwandeln

void xy_to_rphi (double x, double y, double& r, double& phi)
{
    r = sqrt (x*x + y*y);
    phi = atan2 (y, x);
}
```

4.6.2. Der leere Befehl (*Null Statement*)

Überall dort, wo C/C++ einen Befehl erwartet, wird auch ein *leerer* Befehl akzeptiert. Ein leerer Befehl besteht nur aus dem (abschließenden) Strichpunkt.

```
// String Source auf Dest kopieren

char *strcpy (char *Dest, const char *Source)
{
    char *DestStart = Dest;      // Hilfs-Zeiger

    // Der Reihe nach alle Zeichen in Source auf Dest kopieren, bis das
    // abschließende Null-Byte gelesen wird. Beachten Sie, dass die Zuweisung
    // erfolgt, bevor die Zeiger auf das nächste Zeichen inkrementiert werden,
    // und bevor der Wert des kopierten Zeichens mit while getestet wird.
    // Daher wird auch das abschließende Null-Byte kopiert.

    while (*Dest++ = *Source++)
        ;    // Leerer Befehl; alles ist schon in der obigen Zeile passiert.

    return DestStart;
             // Die Funktion soll einen Zeiger auf die Kopie zurückgeben.
}
```

4.6.3. Befehlsblöcke

Eine beliebige Anzahl von Befehlen (einschließlich null) kann mit geschwungenen Klammern zu einem (Befehls-) Block (*Compound Statement)* zusammengefasst werden. Überall dort, wo C/C++ einen Einzelbefehl erwartet, wird auch ein Befehlsblock akzeptiert.

```
...

if (a > b)
{
    int temp = a;
    a = b;
    b = temp;
}
...
```

4.6.4. Der Sprungbefehl **goto**

Der Befehl **goto** bewirkt einen unbedingten Sprung auf die mit dem Befehl angegebene Sprungmarke. Die Ziel-Sprungmarke muss innerhalb der gleichen Funktion liegen wie der Befehl **goto**; sie darf aber in einem beliebigen Block liegen. **goto** ist allenfalls sinnvoll, wenn ein vorzeitiger Abbruch mehrerer ineinandergeschachtelter Schleifen erforderlich ist:

```
int main ()
{
    for (int i = 0; i < 10000; i++)
    {
        for (int j = 0; j < 10000; j++)
        {
            // irgendeine Abbruchbedingung

            if (Abbruch ())
                goto ende;              // "ende" ist eine Sprungmarke

            // Code, der in der Schleife bearbeitet werden soll
        }
    }

    // Code, der bei voller Abarbeitung beider Schleifen ausgeführt wird:

ende:                                   // Sprung mit goto landet hier
    // Weiterer Programmcode
}
```

4.6.5. Programm-Rückkehr mit **return**

Der Befehl **return** bewirkt eine sofortige Rückkehr einer Funktion zur aufrufenden Funktion. Innerhalb einer Funktion können beliebig viele **return**-Befehle vorkommen. Wenn **return** in der Funktion `main()` ausgeführt wird, wird das Programm beendet. Bei Funktionen, die *nicht* die Type **void** haben, erwartet C++ einen Ausdruck unmittelbar nach **return**, dessen Ergebnis an die aufrufende Funktion zurückgegeben wird. (**return** *ohne* Ausdruck stellt in diesem Fall einen Fehler dar.) Das "Ergebnis" der Funktion `main()` wird an das Betriebssystem weitergegeben und kann (etwa unter MS-DOS mit "IF ERRORLEVEL") dort abgefragt werden. Der mit dem **return**-Befehl angegebene Ausdruck kann beliebig kompliziert sein; sein Ergebnis wird allenfalls (soweit notwendig und möglich) in den Typ der Funktion konvertiert:

```
#include <math.h>                   // für Deklaration von exp() und log()

double ZehnHochX (double x)
{
    return exp(x*log(10));
}
```

Bei Funktionen der Type **void** darf *kein* Ausdruck bei **return** angegeben werden.

Das Ende des Funktions-Blocks entspricht einem **return** ohne nachfolgenden Ausdruck; es ist nur bei Funktionen vom Typ **void** zulässig, die Funktion am Ende ihres Blocks *ohne* **return** zu verlassen.

4.6.6. Programmverzweigung mit **if…else**

Auf den Befehl **if** muss ein Ausdruck in Klammern ("**( )**") folgen, der von einer arithmetischen oder Zeiger-Type sein muss oder in eine solche umgewandelt werden kann. Nur wenn das Ergebnis dieses Ausdrucks von Null verschieden ist, wird der auf **if** und die Klammern folgende Ausdruck oder Block ausgeführt. Optional kann danach der Befehl **else** mit einem weiteren Ausdruck oder Block folgen, der nur dann ausgeführt wird, wenn der unmittelbar hinter **if** in Klammern stehende Ausdruck den Wert Null ergeben hat:

```
...

if (x > 0)
    cout << "x ist größer als 0\n";

else
    cout << "x ist kleiner oder gleich 0\n";
...
```

Bei ineinandergeschachtelten **if…else**-Befehlen bezieht sich ein **else** immer auf das *letzte* ihm vorausgehende **if**, das noch nicht mit einem **else** gepaart ist. Zweckmäßigerweise sollten geschwungene Klammern ("**{ }**") und Einrückungen verwendet werden, um diesen Zusammenhang zu verdeutlichen:

```
...

if (B1)
    if (B2)
        cout << "B1 == TRUE, B2 == TRUE\n";

    else
        cout << "B1 == TRUE, B2 == FALSE\n";

else
    if (B2)
        cout << "B1 == FALSE, B2 == TRUE\n";

    else
        cout << "B1 == FALSE, B2 == FALSE\n";
```

Dieses Codefragment ist identisch zu:

```
...

if (B1)
{
    if (B2)
        cout << "B1 == TRUE, B2 == TRUE\n";

    else
        cout << "B1 == TRUE, B2 == FALSE\n";
}

else
{
    if (B2)
        cout << "B1 == FALSE, B2 == TRUE\n";

    else
        cout << "B1 == FALSE, B2 == FALSE\n";
}
```

4.6.7. Programmverzweigungen mit **switch**

Während mit **if** oder mit **if**…**else** zwischen *zwei* Wegen durch das Programm gewählt werden kann, erlaubt der Befehl **switch** eine Verzweigung zwischen *beliebig vielen* Wegen. Der in Klammern ("**( )**") auf **switch** folgende Ausdruck, der von einer ganzzahligen Datentype sein oder in eine solche eindeutig konvertiert werden können muss, wird ausgewertet; das Ergebnis dieses Ausdrucks bestimmt die weitere Vorgangsweise. Auf **switch** folgt üblicherweise ein Befehlsblock, innerhalb dessen sich beliebig viele (auch keine) **case**-Sprungmarken und maximal eine **default**-Sprungmarke befinden können. Jede **case**-Sprungmarke muss von einem Ausdruck gefolgt sein, dessen Ergebnis eine konstante ganze Zahl sein muss; diese Werte müssen für alle **case**-Sprungmarken individuell verschieden sein. Wenn das Ergebnis des auf **switch** folgenden Ausdrucks mit dem Wert nach einer **case**-Sprungmarke übereinstimmt, wird (wie bei **goto**) zu dieser Sprungmarke verzweigt. Existiert kein passender Wert einer **case**-Sprungmarke, dann verzweigt das Programm zur **default**-Sprungmarke, sofern eine solche vorhanden ist; existiert keine **default**-Sprungmarke, so verzweigt das Programm hinter das Ende des auf **switch** folgenden Befehlsblocks. Mehrere Sprungmarken dürfen unmittelbar hintereinander stehen. Ein **break**-Befehl bewirkt einen Sprung auf den ersten Befehl nach dem **switch**-Block. Bitte beachten Sie, dass **case** ***x*:** und **default:** *Sprungmarken* sind, die nur innerhalb des auf **switch** folgenden Blocks gültig sind und daher von außen nicht angesprungen werden können. Der auf die einzelnen Sprungmarken folgende Code braucht daher *nicht* als Block in geschwungene Klammern gesetzt zu werden (obwohl dies auch nicht schadet); sofern kein **break**-Befehl vorgesehen wurde, läuft (ebenso wie bei jeder anderen Sprungmarke) die Programmausführung einfach linear über die nächste Sprungmarke hinaus weiter. (Fehlende **break**s in **switch**-Befehlen stellen einen der häufigsten logischen Fehler dar!) Eine typische Anwendungen für **switch** sind Befehlsprozessoren, also Programmteile, die je nach dem Wert eines (Eingabe-)Parameters auf verschiedene Funktionen verzweigen.

Das folgende Demo-Programm fordert in einer Schleife einen numerischen Eingabewert an. Für die Werte 0, 1 und 2 sowie für Primzahlen soll eine entsprechende Meldung ausgegeben werden. Bei Eingabe von 0 wird das Programm beendet.

Demo-Programm 4_06_01.cc

```
#include <iostream.h>

int main ()
{
    int x = 0;

    do
    {
        cout << "Bitte Wert eingeben: ";
        cin >> x;                                    // x einlesen

        switch (x)
        {
            case 0:
                cout << "Null\n";
                break;

            case 1:
                cout << "Eins\n";
                break;

            case 2:
                cout << "Zwei\n";

            case 3:
            case 5:
            case 7:
                cout << "Primzahl!\n";
                break;

            default:
                cout << "Irgendwas\n";
        }

    } while (x != 0);
}
```

Das Demo-Programm fordert in einer **do**…**while**-Schleife (siehe Seite 90) eine numerische Eingabe an. Der Eingabewert wird am Ende der Schleife mit `while (x != 0);` getestet; die Schleife wird so lange durchlaufen, solange `x` einen von `0` verschiedenen Wert hat. Der Wert von `x` wird mit `switch (x)` getestet; für die Werte 0, 1 und 2 von `x` verzweigt das Programm zu den entsprechenden Meldungen. Im Fall von 0 und 1 wird nach der Ausgabe der Meldung aufgrund des **break**-Befehls ans Ende des auf `switch (x)` folgenden Blocks gesprungen; im Fall des Wertes 2 "fällt" das Programm "durch" zur nachfolgenden Code-Sequenz, die auch dann ausgeführt wird, wenn eine der Primzahlen unter 10 eingegeben wurde. Für alle anderen Werte wird der Teil des Programms ausgeführt, der auf die **default**-Sprungmarke folgt. Ein Beispiel für die Ausgabe des Programms ist:

```
Bitte Wert eingeben: 1
Eins
Bitte Wert eingeben: 2
Zwei
Primzahl!
Bitte Wert eingeben: 3
Primzahl!
Bitte Wert eingeben: 4
Irgendwas
Bitte Wert eingeben: 5
Primzahl!
Bitte Wert eingeben: 0
Null
```

4.6.8. **while**-Schleifen

Eine **while**-Schleife besteht aus dem Schlüsselwort **while**, gefolgt von einem Ausdruck in Klammern ("**( )**"), der von einer ganzzahligen Type, einer Zeigertype oder einer Klassentype sein muss, die eindeutig in einen Ausdruck mit einer ganzzahligen Type umgewandelt werden kann. Der darauffolgende Befehl oder Befehlsblock wird wiederholt ausgeführt, solange der Ausdruck in der Klammer hinter **while** ein von Null verschiedenes Ergebnis hat. Der Ausdruck wird jeweils *vor* der Ausführung des Befehls oder Blocks getestet; die Ausführung einer **while**-Schleife kann daher auch völlig unterbleiben. Endlosschleifen können mit einem Befehl der Form `while (1)` realisiert werden. Das folgende Beispiel zeigt ein Programm, das führende Leerzeichen oder Tabulatoren von einem String entfernt:

Demo-Programm 4_06_02.cc

```
#include <stdio.h>

const char *strip (const char *buffer);
void write (const char *buffer);

int main ()
{
    char buffer[256];                          // Eingabe-Puffer

    printf ("Bitte String eingeben: ");
    gets (buffer);
    write (buffer);                            // Original
    write (strip (buffer));                    // gestrippt
}

const char *strip (const char *buffer)
{
    const char *ptr = buffer;

    while (ptr && *ptr && *ptr <= ' ')
        ptr++;
    // Prüfe, ob buffer kein Null-Zeiger ist (ptr != 0), ob das abschließende
    // Null-Byte noch nicht erreicht wurde(*ptr != 0), und ob es sich um ein
    // Zeichen mit einem ASCII-Wert <= 0x20 handelt. Wenn ja, teste das
    // nächste Zeichen (ptr++).

    return ptr;
}
```

```
void write (const char *buffer)
{
    printf ("\n***%s***\n", buffer);    // String mit "***" "eingeklammert"
}
```

Die Ausgabe des Demo-Programms sieht etwa so aus:

```
Bitte String eingeben:     Hello, world!

***    Hello, world!***

***Hello, world!***
```

Hier mussten die "alten" C-Ein-/Ausgabefunktionen aus `stdio.h` verwendet werden, weil `cin` führende Leerzeichen entfernen würde.

Beachten Sie die Bedeutung der Deklaration `const char *ptr`: Diese Deklaration besagt *nicht*, dass der Wert des Zeigers konstant ist (also die Adresse, auf die er zeigt), sondern dass das *Objekt*, auf das er zeigt, nicht verändert werden darf. Das erstere Verhalten hätte mit `char * const ptr` deklariert werden müssen.

4.6.9. **do**...**while**-Schleifen

Dieser Typ von Schleifen ist ähnlich dem der **while**-Schleifen, jedoch mit dem Unterschied, dass die Schleifenbedingung erst geprüft wird, *nachdem* die Schleife durchlaufen wurde. **do**...**while**-Schleifen werden daher mindestens einmal ausgeführt. Sie bestehen aus dem Schlüsselwort **do**, das von einem Befehl oder Befehlsblock gefolgt ist, auf den **while** mit der Schleifenbedingung in Klammern ("**( )**"), gefolgt von einem Strichpunkt, folgt. Das folgende Demo-Programm wartet, bis der Buchstabe "x" auf der Tastatur eingegeben wurde:

Demo-Programm 4_06_03.cc

```
#include <conio.h>
#include <iostream.h>

int main ()
{
    char ch;

    do
        ch = getche ();    // warte auf ein Zeichen von der Tastatur, und gib
                           // sein Echo aus
    while (ch != 'x');

    cout << "\nEndlich ein \"x\"!";
}
```

Die zugehörige Ausgabe ist:

```
qwertzuiopü+#äölkjhgfdsayx
Endlich ein "x"!
```

Beachten Sie, dass eine Eingabe erfolgen muss, *bevor* das eingegebene Zeichen getestet werden kann. Die Syntax im obigen Beispiel ist korrekt; üblicherweise schreibt man aber nicht

```
    do ch = getche ();
    while (ch != 'x');
```

sondern setzt den "Rumpf" der Schleife in geschwungene Klammern:

```
    do
    {
        ch = getche ();
    }
    while (ch != 'x');
```

Prinzipiell hätte das gleiche Resultat, allerdings mit größerem Aufwand, auch mit einer **while**-Schleife erzielt werden können:

Demo-Programm 4_06_04.cc

```
#include <conio.h>
#include <iostream.h>

int main ()
{
    char ch = getche ();

    while (ch != 'x')
        ch = getche ();

    cout << "\nEndlich ein \"x\"!";
}
```

4.6.10. **for**-Schleifen

for-Schleifen sind ähnlich zu **while**-Schleifen, haben jedoch eine erweiterte Funktionalität. Dem **for**-Befehl folgt eine Klammer ("**( )**"), in der, durch Strichpunkte ("**;**") getrennt, drei Befehle bzw. Ausdrücke stehen, die die folgenden Funktionen haben:

- Ein Befehl, der ausgeführt wird, *bevor* das Programm in die Schleife eintritt, und der Ausdrücke und/oder Deklarationen enthalten kann. Dieser Befehl wird üblicherweise zur Initialisierung der in der Schleife verwendeten Variablen eingesetzt.
- Ein Ausdruck, der funktionell und syntaktisch dem auf **while** folgenden Klammerausdruck äquivalent ist. Dieser Ausdruck wird *vor* jedem Schleifendurchlauf ausgewertet; die Schleife wird so lange durchlaufen, solange das Ergebnis des Ausdrucks von Null verschieden ist.
- Ein Befehl, der auf jeden Fall am *Ende* jedes Schleifendurchlaufs ausgeführt wird, unmittelbar, *bevor* der mittlere Ausdruck vor dem nächsten Durchlauf getestet wird. Dieser Befehl wird üblicherweise zum Inkrementieren des Schleifenzählers verwendet, kann jedoch auch anderweitig eingesetzt werden.

Die beiden Befehle können auch aus mehreren Befehlen oder Ausdrücken zusammengesetzt sein, die durch Kommas ("**,**") voneinander getrennt sind. Diese Ausdrücke werden dann in der angegebenen Reihenfolge nacheinander an der korrekten Stelle der Schleife ausgeführt.

Alle drei Ausdrücke in einer **for**-Schleife sind optional; die sie trennenden beiden Strichpunkte müssen jedoch auf jeden Fall vorhanden sein. Mit `for(;;)` kann eine Endlosschleife eingeleitet werden (äquivalent zu `while(1)`). Die Schleife

```
for (Initialisierung; Testen; Weiter)
{
    Schleifenkörper;
}
```

ist äquivalent zu:

```
Initialisierung;

while (Testen)
{
    Schleifenkörper;

    Weiter;
}
```

Das nachfolgende Demo-Programm nimmt bis zu 20 ganze Zahlen über die Konsole entgegen, sortiert sie in aufsteigender Reihenfolge und gibt sie wieder aus. Die Eingabe wird als beendet betrachtet, wenn die Zahl `0` eingegeben wurde.

Demo-Programm 4_06_05.cc

```
#define FELDGROESSE 20                  // maximale Anzahl der Feldelemente

#include <iostream.h>

int HoleZahlen (int *Feld, int Groesse);
void Schreibe (int *Feld, int Aktiv);
void Sortiere (int *Feld, int Aktiv);

int main ()
{
    int IntFeld [FELDGROESSE];

    int AktGroesse = HoleZahlen (IntFeld, sizeof IntFeld/sizeof IntFeld[0]);
    Schreibe (IntFeld, AktGroesse);
    Sortiere (IntFeld, AktGroesse);
    Schreibe (IntFeld, AktGroesse);
}

// maximal <Groesse> ints einlesen und in <Feld> abspeichern; tatsächliche
// Zahl der ints zurückgeben:

int HoleZahlen (int *Feld, int Groesse)
{
    for (int Zaehler = 0, temp = 1; temp && (Zaehler < Groesse);
        Zaehler++)
    {
        cout << "Zahl: ";
        cin >> temp;
        if (temp)                               // != 0
            *(Feld + Zaehler) = temp;
        else
            Zaehler--;                          // zähle 0 nicht
    }

    return Zaehler;
}

// <Aktiv> Elemente von <Feld> in aufsteigender Reihenfolge sortieren:

void Sortiere (int *Feld, int Aktiv)
{
    for (int i = 0; i < Aktiv - 1; i++)
    {
        for (int j = i + 1; j < Aktiv; j++)
        {
            if (Feld[i] > Feld[j])
            {
                int temp = Feld[i];
                Feld[i] = Feld[j];
                Feld[j] = temp;
            }
        }
    }
}

// Inhalt von <Feld>, getrennt durch Kommas, ausschreiben:

void Schreibe (int *Feld, int Aktiv)
{
    for (int i = 0; i < Aktiv; i++)
        cout << Feld[i] << ((i < Aktiv - 1) ? ", " : "\n");
}
```

Ein Beispiel für die Ausgabe des Programms ist:

```
Zahl: 12
Zahl: 5
Zahl: 7
Zahl: 17
Zahl: 3
Zahl: 0
12, 5, 7, 17, 3
3, 5, 7, 12, 17
```

Noch einige Kommentare zu diesem Programm:

- `#define FELDGROESSE 20`: Felder, deren Größe eventuell irgendwann einmal geändert werden soll, sollten über eine mit **`const`** definierte Konstante oder über ein mit **`#define`** definiertes Synonym angegeben werden. Damit sind spätere Änderungen einfach und an einer einzigen Stelle möglich.
- Funktion `main()`: `sizeof IntFeld/sizeof IntFeld[0]`: Dieser Ausdruck, der vom Compiler in eine Konstante umgesetzt wird, ist hier äquivalent zu `FELDGROESSE`. Die Größe des Feldes sollte auf einem dieser beiden Wege wartungsfreundlich angegeben werden.
- Funktion `HoleZahlen()`: Die **`for`**-Schleife zeigt die Verwendung zweier Ausdrücke (hier die Definitionen von `Zaehler` und `temp`), die durch ein Komma getrennt sind. Vor Beginn jedes Schleifendurchlaufs müssen zwei Bedingungen getestet werden (nämlich, dass `temp != 0` und dass das Feld noch nicht voll aufgefüllt ist), die mit einem logischen UND ("**`&&`**") verknüpft sind. Um vor dem ersten Schleifendurchlauf sicherzustellen, dass `temp` nicht zufällig gleich `0` ist und einen Abbruch der Eingabe bewirkt, wird `temp` mit einem Wert `!= 0` initialisiert. Wenn eine Zahl `!= 0` eingegeben wird, wird sie im Feld abgespeichert, anderenfalls wird `Zaehler` dekrementiert, um die nachher auf jeden Fall erfolgende Inkrementierung zu kompensieren. (Die gleiche Funktion hätte sich effizienter durch Verwendung des Befehls **`break`** erzielen lassen, der auf Seite 93 besprochen wird.)
- Funktion `Sortiere()`: Diese Funktion implementiert einen (ineffizienten, aber einfachen) Sortier-Algorithmus: Beginnend mit dem ersten Element des Feldes wird jedes Element mit allen hinter ihm stehenden verglichen; wenn ein hinter ihm stehendes Element kleiner ist als das Vergleichselement, werden beide vertauscht. Die geschwungenen Klammern bei den beiden **`for`**-Befehlen sind eigentlich redundant; sie wurden hier wegen der besseren Lesbarkeit des Programmcodes vorgesehen.
- Funktion `Schreibe()`: Wiederum unter Verwendung einer **`for`**-Schleife schreibt diese Funktion die durch das Argument `Aktiv` vorgegebene Anzahl von Feldelementen aus. Bei allen Elementen außer dem letzten wird ein Komma hinzugefügt; beim letzten Element statt dessen ein Zeilenvorschub. Die Klammern um den Ausdruck

  ```
  ((i < Aktiv - 1) ? ", " : "\n")
  ```

  sind unbedingt erforderlich, um nicht die Ausgabe des Ergebnisses der Vergleichsoperation, sondern — unter korrekter Verwendung des Operators "**`? :`**" — die Ausgabe des korrekten Strings zu erzielen.

4.6.11. **`break`** und **`continue`**

4.6.11.1. Der Befehl **`break`**

Der Befehl **`break`** ist in zwei Umgebungen wirksam; in beiden Fällen bewirkt er ein vorzeitiges Verlassen ebendieser Umgebung und einen Sprung auf den ersten Befehl, der auf diese Umgebung folgt:

- Innerhalb des Befehlsblocks, der auf einen **`switch`**-Befehl folgt; und
- im "Körper" einer **`while`**-, **`do`**...**`while`**- oder **`for`**-Schleife.

Das folgende Beispiel ist eine elegantere Variante der Funktion `HoleZahlen()` aus dem vorigen Abschnitt:

Demo-Programm 4_06_06.cc

```
int HoleZahlen (int *Feld, int Groesse)
{
    for (int Zaehler = 0; Zaehler < Groesse; Zaehler++)
    {
        int temp;

        cout << "Zahl: ";
        cin >> temp;
        if (! temp)                             // == 0
            break;

        *(Feld + Zaehler) = temp;
    }
    return Zaehler;
}
```

Funktionell ist diese Version der Funktion identisch zu der im vorigen Abschnitt präsentierten; die Verwendung von **break** erspart aber einige Kunstgriffe. So braucht nicht mehr am Anfang der Schleife der Wert von `temp` getestet zu werden; die Definition von `temp` kann in die Schleife hineingenommen werden. Falls der Wert `0` eingegeben wurde, wird der **break**-Befehl ausgeführt und die Schleife verlassen. Der Wert `0` wird daher weder im **int**-Feld abgespeichert, noch wird `Zaehler` für eine Null inkrementiert.

Wenn mehrere **switch**-Befehle und/oder Schleifen ineinandergeschachtelt sind, wird mit **break** jeweils der diesen Befehl unmittelbar umschließende **switch**-Befehl bzw. die ihn umschließende Schleife verlassen. Wenn *alle* **switch**-Befehle oder Schleifen verlassen werden sollen, sind daher weitere **break**-Befehle in den äußeren **switch**-Befehlen oder Schleifen, allenfalls mit einer neuen Prüfung der Abbruchbedingung, erforderlich. In diesem Fall kann ein **goto** wesentlich einfacher und auch übersichtlicher sein.

4.6.11.2. Der Befehl **continue**

Dieser Befehl ist in C/C++ nur im Kontext einer Schleife gültig. Er bewirkt einen Sprung ans Ende der ihn unmittelbar umgebenden Schleife. In **while**- und **do**…**while**-Schleifen erfolgt der Sprung unmittelbar vor die Prüfung der Fortsetzungsbedingung der Schleife; in **for**-Schleifen erfolgt er unmittelbar vor den dritten Ausdruck im **for**-Befehl (also im allgemeinen vor die Inkrementierung des Schleifenzählers).

Im folgenden Beispiel wird ein Teil des Körpers der Schleife übersprungen, wenn der eingegebene Wert nicht durch 2 teilbar war:

Demo-Programm 4_06_07.cc

```
#include <iostream.h>

int main ()
{
    for (int i = 0, j = 0; i < 10; i++)
    {
        int x;
        cout << "Wert: ";
        cin >> x;

        if (x%2)                                // durch 2 nicht teilbar
            continue;

        cout << "ist teilbar durch 2\n";
        j++;                                    // zähle Treffer
    }

    cout << j << " Treffer!\n";
}
```

In diesem Beispiel wird die Meldung "`ist teilbar durch 2`" dann mit **continue** übersprungen, wenn die Bedingung `(x%2)` (`x` modulo 2 ungleich 0) erfüllt war; ebenso wird auch die Inkrementierung des "Trefferzählers" `j` in diesem Fall übersprungen. Die Inkrementierung des Schleifenzählers `i` erfolgt aber in jedem Fall; es werden immer genau 10 Werte angefordert.

5. Objektorientierte Programmierung

C++-spezifisch

Dieser Abschnitt ist den in C++ neu eingeführten Sprachfunktionen gewidmet, die eine objektorientierte Programmierung erlauben. Dies sind insbesondere Klassen, Schablonen (*Templates*) und die Neudefinition (*Overloading*) von Funktionen und Operatoren. Die Aspekte der Vererbung von Klasseneigenschaften bei der Definition abgeleiteter Klassen werden ebenso behandelt wie die Zugriffsrechte auf Klassenelemente (Daten und Funktionen). Schließlich kommen die Erzeugung, Zuweisung und Konversion von Klassen-Objekten und spezielle Gesichtspunkte von Operationen mit Klassen-Objekten zur Sprache. Am Ende des Kapitels werden die Kriterien diskutiert, wann die Anwendung welcher Funktionen einer objektorientierten Sprache sinnvoll und empfehlenswert ist.

Der Inhalt dieses gesamten Abschnittes ist im wesentlichen C++-spezifisch. Weitere Hinweise auf diesen Umstand entfallen daher in seinem weiteren Text.

C++-spezifisch

5.1. Klassen-Typen

5.1.1. Allgemeines

Klassen-Typen sind jene, die mit den Schlüsselworten

- **`class`**,
- **`struct`** und
- **`union`**

definiert werden. Klassen-Typen können ineinandergeschachtelt definiert werden; in diesem Fall gilt die Definition einer "inneren" Klasse nur innerhalb der sie umschließenden Klasse.

Generell können alle Klassen-Typen beliebig viele der folgenden Elemente enthalten:

- Datenelemente, die den Zustand und die Eigenschaften eines *Objekts* der Klasse beschreiben;
- *Konstruktor*-Funktionen, die neue Objekte der Klasse initialisieren;
- *Destruktor*-Funktionen, die "aufräumen", wenn ein Objekt nicht mehr benötigt wird; und
- *Klassenelement*-Funktionen, die das Verhalten eines Klassenobjekts bestimmen und für die betreffende Klasse spezifische Operationen vornehmen.

Eigenschaft	**Strukturen**	**Klassen**	***Union*s**
Schlüsselwort	**`struct`**	**`class`**	**`union`**
Standard-Zugriffsrecht	**`public`**	**`private`**	**`public`**
Zugriffsbeschränkungen	keine	keine	nur *ein* Element zu einem gegebenen Zeitpunkt

5.1.2. Anonyme Klassen

Bei der Definition einer Klasse wird üblicherweise ein Name der Klasse angegeben. Diese Angabe kann unter den folgenden Bedingungen auch *entfallen* ("*Anonyme Klassen*"):

- **`typedef`**-Namensdefinitionen:

```
typedef struct
{
    short x;
    short y;
} POINT;
```

- Eingeschlossene Klassen:

```
struct PTWert
{
    POINT pt;
    union
    {
        int iWert;
        long lWert;
    };
};
```

```
PTWert ptw;
int i = ptw.iWert;
POINT p = ptw.pt;
short j = ptw.pt.x;
```

In diesem Fall ist ein Zugriff auf Elemente der *eingeschlossenen* Klasse ebenso möglich, wie wenn diese Elemente der *einschließenden* Klasse wären (nur in C++, nicht in ANSI-C!).

5.1.3. Definition einer Klasse

Eine Klasse gilt mit dem Ende ihrer Definition (also etwa des Blocks

```
class Point
{
public:
    Point ()                            // Konstruktor definiert
        {nx = ny = 0;}

    short& x ();                        // Zugriffsfunktion deklariert
    short& y ();                        // Zugriffsfunktion deklariert

private:
    short nx;
    short ny;
};
```

als definiert. Es ist dabei gleichgültig, ob *Klassenelement-Funktionen* bereits *definiert* wurden (ob ihr "Körper" in geschwungenen Klammern angegeben wurde), oder ob sie nur *deklariert* wurden. Innerhalb der Definition einer Klasse können jedoch *Zeiger* oder *Referenzen* auf diese Klasse vorkommen:

```
struct LinkedList
{
    LinkedList *vor;                    // Link-Zeiger nach vorne
    LinkedList *rueck;                  // und hinten

    char inhalt [80];
};
```

5.2. Klassen-Objekte

5.2.1. Operationen mit Klassen-Objekten

C++ erlaubt die folgenden Operationen mit und an Objekten einer Klasse:

- Zuweisungen: Standardmäßig erfolgen Zuweisungen zwischen *kompatiblen* Klassen-Objekten (Objekten derselben Klasse, oder von einem Objekt einer abgeleiteten Klasse auf ein Objekt ihrer zugehörigen Basisklasse) durch bitweise Kopie. Andere Algorithmen können durch Verwendung eines benutzerdefinierten Zuweisungsoperators (Funktion **`operator=;`** siehe Seite 163) erhalten werden.
- Initialisierungen unter Verwendung eines Kopier-Konstruktors (siehe Seite 62, 144 und 148).

5.2.2. Leere Klassen

Die Deklaration einer *leeren Klasse* ist zulässig; aber auch Objekte einer leeren Klasse haben eine von Null verschiedene Größe. Dies ist notwendig, um sicherzustellen, dass Zeiger auf unterschiedliche, auch leere, Objekte immer auf unterschiedliche Adressen weisen. Im folgenden Beispiel werden zwei Objekte der Klasse `Leer` erzeugt und ihre Adressen ermittelt:

Demo-Programm 5_02_01.cc

```
#include <iostream.h>

class Leer
{
};

int main ()
{
    Leer leera;
    Leer leerb;

    unsigned long la, lb;
    la = (unsigned long) &leera;
    lb = (unsigned long) &leerb;

    cout << "Adresse von leera = " << hex << la << "\n";
    cout << "Adresse von leerb = " << hex << lb << "\n";
    cout << "Größe von Leer = " << sizeof (Leer);
}
```

Ausführen dieses Programms mit dem GNU C++-Compiler ergibt beispielsweise:

GNU-C/C++-spezifisch

```
Adresse von leera = 51a90
Adresse von leerb = 51a8c
Größe von Leer = 1
```

Beachten Sie bitte die folgenden Punkte bei dem obenstehenden Demo-Programm:

- Die Adressen von `leera` und `leerb` wurden hier durch eine explizite Typenumwandlung (hier mit *Type Casts*) in normale ganzzahlige Werte umgewandelt. (Das ist eigentlich nicht nötig, weil `cout` auch die Zeiger korrekt gehandhabt hätte — `cout << hex << &leera << "\n";`)
- In diesem Fall war nur die Verwendung von *Type Casts* möglich, weil die Konversion im Funktions-Stil bei einem zweiteiligen Typennamen wie "**`unsigned long`**" einen Syntaxfehler zur Folge hätte:

```
la = (unsigned long) &leera;              // Ok
la = unsigned long (&leera);              // Fehler
la = long (&leera);                       // Ok
```

➡ Bei der Ausgabe von `la` und `lb` wurde der *Manipulator* `hex` verwendet, der bewirkt, dass alle nachfolgenden Werte, auch bei weiteren Aufrufen von `cout`, als hexadezimale Zahl ausgegeben werden. (Es gibt auch Manipulatoren `dec` und `oct`, die dezimale bzw. oktale Ausgabe erzwingen; siehe Seite 192.)

Das folgende Beispiel zeigt die Wirkung dieser *Manipulatoren*:

Demo-Programm 5_02_02.cc

```
#include <iostream.h>

int main ()
{
    const int i = 123;
    const int j = 65535;

    cout << dec << "i = " << i << ", j = " << j << "\n";
    cout << oct << "i = " << i << ", j = " << j << "\n";
    cout << hex << "i = " << i << ", j = " << j << "\n";
    cout << "i = " << i << ", j = " << j << "\n";
}
```

Das Programm erzeugt die Ausgabe:

```
i = 123, j = 65535
i = 173, j = 177777
i = 7b, j = ffff
i = 7b, j = ffff
```

Die letzte Zeile beweist, dass der Manipulator `hex` über den Aufruf von `cout` hinaus wirksam ist, bei dem er angegeben wurde.

5.3. Definition von Klassen

Die *Deklaration* einer Klasse, also die Auflistung der *Elemente* der Klasse, muss *genau einmal* erfolgen. Änderungen der Zahl oder Type der Klassenelemente sind nach der Deklaration nicht mehr zulässig.

Die Initialisierung von *Objekten* einer Klasse im Zuge ihrer *Definition* kann erfolgen durch

- eine Liste der Anfangswerte des Objekts in "**`{ }`**"; dies ist nur zulässig, wenn die Klasse
  - keinen Konstruktor;
  - keine Elemente, die *nicht* **`public`** sind;
  - keine Basisklassen und virtuellen Funktionen

  hat (siehe Seite 61).
- einen geeigneten Konstruktor der Klasse (siehe Seite 62).

Eine Initialisierung innerhalb der *Deklaration* der Klasse ist unzulässig.

Statische Datenobjekte müssen außerhalb der Deklaration der Klasse explizit initialisiert werden.

5.4. Klassenelemente

5.4.1. Funktionen

5.4.1.1. Nicht-statische Funktionen

Nicht-statische Funktionen, die innerhalb einer Klasse deklariert oder definiert wurden, müssen grundsätzlich mit dem Elementauswahl-Operator ("`.`" bzw. "`->`") aufgerufen werden. Sie beziehen sich immer auf das spezifische Objekt der Klasse, für das sie unter Verwendung des Elementauswahl-Operators aufgerufen wurden. Eine Ausnahme besteht beim Aufruf dieser Funktionen innerhalb *anderer* Klassenelement-Funktionen derselben Klasse. (In diesem Fall ist es klar, auf *welches* Objekt der Klasse sich der Aufruf beziehen soll.)

Demo-Programm 5_04_01.cc

```
#include <iostream.h>

class Point
{
public:
    short& x()
        { return _x; }
    short& y()
        { return _y; }
    void Show ()
        { cout << "x = " << x() << ", y = " << y() << "\n"; }
private:
    short _x, _y;
};

int main ()
{
    Point pt1;
    Point pt2;

    pt1.x() = 3;
    pt1.y() = 5;

    pt2.x() = 7;
    pt2.y() = 9;

    pt1.Show();
    pt2.Show();
}
```

In der Funktion `Point::Show()` kann auf die Funktionen `Point::x()` und `Point::y()` *ohne* Angabe der Klassenbezeichnung ("`Point::`") oder eines Objekts der Klasse zugegriffen werden. Die Ausgabe des Programms ist:

```
x = 3, y = 5
x = 7, y = 9
```

5.4.1.2. Der **`this`**-Zeiger

Der Zeiger **`this`** ist für alle nicht-statischen Klassenelement-Funktionen definiert. Er zeigt auf das Objekt, für das die Klassenelement-Funktion aufgerufen wurde. Die Type von **`this`** ist *`type`* **`* const`**; für die Klasse `Point` also beispielsweise `Point * const`. Alle Zugriffe einer Klassenelement-Funktion auf (Daten- oder Funktions-) Elemente der Klasse erfolgen unter impliziter

Verwendung von **this**; eine explizite Verwendung von **this** ist im allgemeinen nicht erforderlich. Die Klasse Point des Beispiels im vorigen Abschnitt hätte also auch folgendermaßen definiert werden können:

Demo-Programm 5_04_02.cc

```
class Point
{
public:
    short& x()
        { return this->_x; }
    short& y()
        { return this->_y; }
    void Show ()
        { cout << "x = " << this->x() << ", y = " << this->y() << "\n"; }

private:
    short _x, _y;
};
```

Beachten Sie, dass die Type von **this** die eines *konstanten Zeigers* ist. **this** kann also nicht auf ein anderes Objekt "umgebogen" werden.

Die Type von **this** kann in der Funktionsdeklaration innerhalb der Definition der Klasse durch die Schlüsselwörter **const** und **volatile** (und durch implementierungsspezifische Schlüsselwörter wie **__near** und **__far**) verändert werden. Beispielsweise hätte eine Deklaration

```
class Point
{
public:
    short& x() const
        { return _x; }
    short& y()
        { return _y; }
    void Show ()
        { cout << "x = " << this->x() << ", y = " << this->y() << "\n"; }
private:
    short _x, _y;
};
```

bewirkt, dass in der Funktion Point::x() (*nicht* aber in Point::y()) die Type von **this const** Point * **const** gewesen wäre. Damit wäre es zwar weiterhin möglich, den Wert von _x auszulesen, nicht aber, ihn zu verändern. Der GNU C++-Compiler würde unter Verwendung der obigen Definition von Point die Warnung ausgeben:

GNU-C/C++-spezifisch

```
test3.cc: In method 'short int & Point::x() const':
test3.cc:6: warning: conversion to reference of read-only member 'short int
Point::_x'
```

5.4.1.3. Statische Funktionen

Statische Klassenelement-Funktionen haben im Gegensatz zu nicht-statischen *keinen* **this**-Zeiger; sie können daher nicht direkt auf Objekte der Klasse, sondern nur auf folgende Elemente der Klasse zugreifen:

- Statische Datenelemente;
- Enumeratoren;
- Eingeschlossene Typen.

Sie können ohne Verwendung eines Objekts ihrer Klasse aufgerufen werden (siehe das Beispiel auf Seite 46). Ein Aufruf unter Verwendung eines Objekts der Klasse ist jedoch zulässig, wenn ein solches existiert. Das oben erwähnte Demo-Programm könnte daher auch so geschrieben werden:

Demo-Programm 5_04_03.cc

```
#include <iostream.h>

class Punkt
{
public:
    Punkt () { PunkteZahl++; }                // Konstruktor
    ~Punkt () { PunkteZahl--; }               // Destruktor

    unsigned& x() { return xKoord; }          // Zugriffsfunktion
    unsigned& y() { return yKoord; }          // Zugriffsfunktion

    static int Anzahl()                       // statische Zugriffsfunktion
        { return PunkteZahl; }

    static int PunkteZahl;

private:
    unsigned xKoord;
    unsigned yKoord;
};

int Punkt::PunkteZahl = 0;       // Definition von Punkt::PunkteZahl

int main ()
{
    cout << "Anzahl der Punkte: " << Punkt::Anzahl () << "\n";
                                 // "Punkt::Anzahl", weil statische Funktion
    Punkt p1;                    // neues Objekt

    cout << "Anzahl der Punkte: " << p1.Anzahl () << "\n";
                                 // p1 ist ein gültiges Objekt

    // weiterer Code wie auf Seite 46
}
```

Klassenobjekte, auf die sich statische Funktionen beziehen, werden nicht ausgewertet (wichtig, wenn das Objekt eine Funktion ist!).

5.4.2. Datenelemente

5.4.2.1. Statische Daten

Die statischen Datenelemente einer Klasse (also solche, die mit dem Schlüsselwort **`static`** versehen sind) werden in der Definition der Klasse wohl *deklariert*, müssen aber gesondert *definiert* werden. Sie sind auch eigenständige Objekte, deren Type im allgemeinen *nicht* die Type der Klasse ist. Sie werden zwar ebenso mit Klassengültigkeit (*Class Scope*) deklariert wie die nichtstatischen Klassenelemente, werden aber mit Dateigültigkeit und externer *Linkage* definiert. Auf sie kann — ebenso wie auf statische Klassenelement-Funktionen — entweder über ein Objekt der Klasse mit dem Elementauswahl-Operator ("`.`" oder "`->`") zugegriffen werden, oder unter Angabe des Klassennamens, gefolgt vom Operator "`::`". Das Beispiel von Seite 46 bzw. 104 kann folgendermaßen modifiziert werden:

Demo-Programm 5_04_04.cc

```
#include <iostream.h>

class Punkt
{
public:
    Punkt () { PunkteZahl++; }   // Konstruktor
    ~Punkt () { PunkteZahl--; }  // Destruktor

    static int PunkteZahl;
};
```

```
int Punkt::PunkteZahl = 0;              // Definition von Punkt::PunkteZahl

int main ()
{
    cout << "Anzahl der Punkte: " << Punkt::PunkteZahl << "\n";
        // "Punkt::PunkteZahl", weil statisches Objekt, und noch kein Objekt
        // der Klasse Punkt vorhanden
    Punkt p1;                           // neues Objekt
    cout << "Anzahl der Punkte: " << p1.PunkteZahl << "\n";
                                        // Alternativ: "p1." statt "Punkt::"
    Punkt *p2 = new Punkt;              // p2 ist Zeiger auf ein Objekt

    // die folgenden drei Ausgaben sind identisch:
    cout << "Anzahl der Punkte: " << p2->PunkteZahl << "\n";
    cout << "Anzahl der Punkte: " << p1.PunkteZahl << "\n";
    cout << "Anzahl der Punkte: " << Punkt::PunkteZahl << "\n";

    delete p2;                          // ruft Destruktor auf
    cout << "Anzahl der Punkte: " << Punkt::PunkteZahl << "\n";
}
```

Das Programm erzeugt die Ausgabe:

```
Anzahl der Punkte: 0
Anzahl der Punkte: 1
Anzahl der Punkte: 2
Anzahl der Punkte: 2
Anzahl der Punkte: 2
Anzahl der Punkte: 1
```

Statische Datenelemente, die nicht in der Definition der Klasse als **`public`** deklariert wurden, sind nur für Klassenelement-Funktionen und solche, die mit **`friend`** (siehe Seite 130) deklariert wurden, zugänglich. Nichtsdestoweniger können und müssen auch *nicht* als **`public`** deklarierte statische Klassenelemente explizit außerhalb der Klassendefinition definiert und eventuell initialisiert werden. Im Beispiel von Seite 46 bzw. von Seite 104, in dem nur Klassenelement-Funktionen auf das statische Datenelement `Punkt::PunkteZahl` zugreifen, hätte die Definition der Klasse auch lauten können:

Demo-Programm 5_04_05.cc

```
class Punkt
{
public:
    Punkt () { PunkteZahl++; }
    ~Punkt () { PunkteZahl--; }

    // weitere Funktionen wie oben

    static int Anzahl() { return PunkteZahl; }

private:
    static int PunkteZahl;

    // weitere Datenelemente wie oben
};

int Punkt::PunkteZahl = 0;              // Definition von Punkt::PunkteZahl
```

5.4.2.2. Unions

C++-**`union`**s können ebenso wie alle anderen Klassentypen auch Klassenelement-Funktionen enthalten. Sie dürfen auch Konstruktoren und Destruktoren, jedoch keine *virtuellen Funktionen* (siehe Seite 117) enthalten. Sie dürfen weder *Basisklassen* (siehe Seite 112ff) sein noch haben. Weiters dürfen sie *nicht* die folgenden Datentypen als Elemente beinhalten:

- Klassen-Typen mit Konstruktoren und/oder Destruktoren;
- Klassen-Typen mit einem benutzerdefinierten Zuweisungs-Operator (siehe Seite 163);
- Statische Datenelemente.

unions dürfen auch *anonym* sein (d.h., keinen Namen haben); in diesem Fall deklarieren sie ein *Objekt* und keine *Type*. Das folgende Beispiel verwendet eine **union**, um wahlweise ein Zeichen, einen Zeiger auf einen String oder eine ganzzahlige Variable zu speichern und auszugeben.

Für anonyme **unions** gelten die folgenden Einschränkungen:

- Sie sind immer **public**;
- Sie dürfen keine Funktions-Elemente haben.

Demo-Programm 5_04_06.cc

```
#include <iostream.h>

class Data
{
    enum DataType { Char, Int, String };
    DataType type;                              // ist private!

    union
    {
        char CharData;
        int IntData;
        char *StringData;
    };

    _data (int i)                               // Hilfsfunktion
    {
        IntData = i;
        type = Int;
    }

public:
    Data () { _data (0); }                      // Default-Konstruktor

    Data (int i) { _data (i); }                 // Konstruktor für ints

    Data (char ch)                              // Konstruktor für chars
    {
        CharData = ch;
        type = Char;
    }

    Data (char *psz)                            // Konstruktor für Strings
    {
        StringData = psz;
        type = String;
    }

    void Print ();              // Ausgabe-Funktion: hier nur deklariert
};

void Data::Print (void)         // Ausgabe-Funktion: Definition
{
    switch (type)
    {
        case Char:
            cout << "char:   " << CharData << "\n";
            break;

        case Int:
            cout << "int:    " << IntData << "\n";
            break;
```

```
        case String:
            cout << "string: " << StringData << "\n";
            break;
    }
}

int main ()
{
    Data d1;                    // Default: int = 0
    Data d2 = 'K';              // type = Char
    Data d3 = 17;               // type = Int
    Data d4 = "Hello, world!";  // String

    d1.Print();
    d2.Print();
    d3.Print();
    d4.Print();

    d2.CharData = 'L';
    d2.Print();
}
```

Das Programm erzeugt die Ausgabe:

```
int:    0
char:   K
int:    17
string: Hello, world!
char:   L
```

In diesem Demo-Programm wird ein Klassenelement mit einer Aufzählungs-Type (`DataType type`) verwendet, um zwischen den verschiedenen in einem Objekt der Klasse möglichen Datentypen zu unterscheiden. Beachten Sie, dass auf die Aufzählungs-Type `DataType` nur innerhalb der Klasse `Data` und in der als Teil der Klasse deklarierten Funktion `Data::Print` zugegriffen werden kann; in `main()` wäre ein Zugriff nicht mehr möglich.

Eigentlich sollte aufgrund der Definition der Klasse auch die **`union`** in `Data` **`private`**, also von außen nicht zugänglich sein. Da anonyme **`union`**s aber immer **`public`** sind, ist der Zugriff auf ihr Element `CharData` in `main()` zulässig.

Dank der Definition der **`union`** in `Data` als *anonyme* **`union`** ist ein Zugriff auf ihre Elemente einfacher möglich. Hätte die **`union`** einen Namen bekommen (im folgenden Codefragment "`DataUnion`"), so wäre ein Zugriff auf ihre Elemente nur unter Verwendung des Elementauswahl-Operators "**`.`**" möglich gewesen. In diesem Fall wäre die **`union`** auch, wie erwartet, **`private`** für die Klasse `Data` gewesen, also von `main()` aus nicht mehr zugänglich:

Demo-Programm 5_04_07.cc

```
class Data
{
    enum DataType { Char, Int, String };
    DataType type;                          // ist private!

    union
    {
        char CharData;
        int IntData;
        char *StringData;
    } DataUnion;

    _data (int i)                           // Hilfsfunktion
    {
        DataUnion.IntData = i;
        type = Int;
    }

public:
    Data () { _data (0); }                  // Default-Konstruktor

    Data (int i) { _data (i); }             // Konstruktor für ints
```

```
    Data (char ch)                              // Konstruktor für chars
    {
        DataUnion.CharData = ch;
        type = Char;
    }

    Data (char *psz)                    // Konstruktor für Strings
    {
        DataUnion.StringData = psz;
        type = String;
    }

    void Print ();        // Ausgabe-Funktion: hier nur deklariert
};

// und so weiter
```

Das Demo-Programm verwendet *Function Overloading*, um unterschiedliche Konstruktoren für die verschiedenen Datentypen (**char, int, char** *) zu realisieren. Aus Gründen der Code-Effizienz wurde eine gemeinsame Funktion für den Default-Konstruktor und den Konstruktor für ein **int**-Argument vorgesehen (`_data()`). Es wäre naheliegend, jedoch falsch, gewesen, den folgenden Code zu schreiben:

```
    Data () { Data (0); }

    Data (int i)
    {
        IntData = i;
        type = Int;
    }
```

In diesem Fall wäre mit dem Befehl "`Data d1;`" in `main()` zunächst der Default-Konstruktor `Data()` aufgerufen worden. Dieser hätte seinerseits den *Konstruktor* `Data(int)` aufgerufen, der ein *neues* Datenobjekt initialisiert hätte (nämlich ein temporäres Objekt in `Data()`, auf das keinerlei Zugriff möglich gewesen wäre, und das mit dem Ende von `Data()` wieder verschwunden wäre). Das Objekt `d1` wäre daher nicht initialisiert worden; insbesondere wäre in seinem Aufzählungs-Element `type` nur zufällig einer der zulässigen Werte (`0`, `1` oder `2`) gestanden. Damit ist die Wahrscheinlichkeit gering, dass für `d1` eine der drei **case**-Sprungmarken in `Print()` angesprungen worden wäre; das Programm hätte mit großer Wahrscheinlichkeit daher keine Ausgabe für `d1` produziert (und schon gar keine korrekte).

5.4.2.3. Bitfelder

Objekte, die mit **struct** oder **class** deklariert wurden, können Elemente (Bitfelder, *Bit Fields)* enthalten, die kleiner als eine ganzzahlige Datentype sind (siehe Seite 48). Solche Elemente werden mit einem *Deklarator* (ganzzahlige Type und, optional, Name), gefolgt von einem Doppelpunkt ("**:**") und einem ganzzahligen konstanten Ausdruck deklariert, der die Anzahl der Bits in dem Bitfeld angibt. Anonyme (also namenlose) Bitfelder können als Platzhalter verwendet werden. Bitfelder werden häufig in hardwarenahen Problemstellungen verwendet, wenn es darum geht, die einzelnen Bits eines Bytes, Wortes oder Doppelwortes unabhängig voneinander zu interpretieren.

Das folgende Demo-Programm zeigt unter Verwendung von Bitfeldern die *Equipment List* des PC-BIOS an:

Demo-Programm 5_04_08.cc

```
#include <iostream.h>

extern "C" int biosequip(void);  // Bibliotheksfunktion für Equipment List

typedef unsigned short WORD;

struct EquipList
{
    WORD FloppyPresent   : 1;
    WORD NPUPresent      : 1;
    WORD                 : 2;    // nicht benützt (XT-RAM-Größe)
```

```
    WORD ActiveVideoMode : 2;
    WORD FloppyCount     : 2;
    WORD DMAPresent      : 1;
    WORD COMPortCount    : 3;
    WORD GamePortPresent : 1;
    WORD                 : 1;    // nicht benützt (PCJr-spezifisch)
    WORD LPTPortCount    : 2;
};

int main ()
{
    int i = biosequip ();
    EquipList Equip = *(EquipList *) &i;

    if (Equip.FloppyPresent)
        cout << (Equip.FloppyCount + 1);
    else
        cout << "Keine";

    cout << " Diskettenlaufwerk(e)\n";

    if (Equip.NPUPresent)
        cout << "Numerik-Prozessor vorhanden\n";

    cout << Equip.COMPortCount << " Serielle Schnittstelle(n)\n";
    cout << Equip.LPTPortCount << " Druckerschnittstelle(n)\n";
    cout << "Video-Mode: ";

    switch (Equip.ActiveVideoMode)
    {
        case 1:
            cout << "40 Spalten CGA\n";
            break;

        case 2:
            cout << "80 Spalten CGA\n";
            break;

        case 3:
            cout << "80 Spalten Monochrom\n";
            break;

        default:
            cout << "???";
    }
};
```

Auf einem der Rechner des Autors ermittelte das Demo-Programm wahrheitsgemäß:

```
2 Diskettenlaufwerk(e)
Numerik-Prozessor vorhanden
4 Serielle Schnittstelle(n)
3 Druckerschnittstelle(n)
Video-Mode: 80 Spalten CGA
```

GNU-C/C++-spezifisch

Das Demo-Programm verwendet eine Bibliotheksfunktion (`biosequip()`), um den Wert der zwei Bytes der *Equipment List* an der physikalischen Adresse `0x40:0x10` zu ermitteln. Ein direkter Zugriff auf diese Adresse (etwa unter Verwendung eines Zeigers) ist bei Verwendung des GNU-C++-Compilers nicht ohne weiteres möglich, weil der Compiler und die von ihm erzeugten Programme im *Protected Mode* des 386 laufen und einen *virtuellen Adressraum* verwenden, der vom *physikalischen Adressraum* des von DOS und dem PC-BIOS verwendeten *Real Mode* entkoppelt ist.

Da das Resultat der Funktion `biosequip()` vom Typ **`int`** ist, kann es einem Objekt vom Typ `EquipList` nicht unmittelbar zugewiesen werden. Das verwendete Konstrukt

```
    int i = biosequip ();
    EquipList Equip = *(EquipList *) &i;
```

weist das Ergebnis von `biosequip()` einer Hilfs-Variablen `i` zu, ermittelt deren Adresse ("`&i`") und konvertiert den Zeiger auf **`int`** in einen Zeiger auf `EquipList` ("`(EquipList *)`"), der anschließend dereferenziert wird ("`*(EquipList *) &i`"). Das Ergebnis dieser Aktion hat die Type `EquipList` und kann daher einem Objekt dieser Type ("`Equip`") zugewiesen werden. Auf die einzelnen Elemente des Bitfeldes kann ebenso zugegriffen werden wie auf die Elemente einer "gewöhnlichen" Struktur oder Klasse.

Für die Deklaration von `EquipList` wurde als "Basistype" die Type **`unsigned short`** gewählt, für die mittels einer **`typedef`**-Anweisung das Synonym `WORD` definiert wurde. In diesem konkreten Fall wäre es grundsätzlich gleichgültig gewesen, ob als "Basistype" **`unsigned char`**, **`unsigned short`** oder **`unsigned long`** gewählt worden wäre, weil die Elemente des Bitfeldes in keinem Fall die Grenzen einer dieser Typen überspannt hätten.

5.4.3. Verschachtelte Klassen

Eine Klasse kann innerhalb (und mit der Gültigkeit innerhalb) einer anderen Klasse deklariert werden; solche Klassen werden als *verschachtelte Klassen* (*Nested Classes*) bezeichnet. Eine verschachtelte Klasse ist unmittelbar nur von der sie umgebenden Klasse aus zugänglich; für andere Zugriffe muss ein voll spezifizierter Name (*Fully Qualified Name — `umschließende Klasse :: eingeschlossene Klasse`*) angegeben werden.

```
class EinAusgabe
{
public:
    enum EinAusFehler {Ok, Zugriff, Allgemein };

    // verschachtelte Klassen:

    class Eingabe
    {
    public:
        int read ();
        int read_ok () { return Eingabefehler == Ok; };
    private:
        EinAusFehler Eingabefehler;
    };

    class Ausgabe
    { // Deklarationen };
};

int EinAusgabe::Eingabe::read ()
{
    // Definition der Klassenelement-Funktion
}
```

Es gelten die folgenden Regeln:

- Verschachtelte Klassen gelten als innerhalb der sie einschließenden Klasse deklariert; sie sind von außerhalb nicht sichtbar. Es können daher außerhalb der einschließenden Klasse Objekte der verschachtelten Klasse nur unter Verwendung eines voll spezifizierten Namens deklariert werden:

  ```
  EinAusgabe ea;                // Objekt der einschließenden Klasse
  EinAusgabe::Eingabe ein;      // Objekt der verschachtelten Klasse
  ```

- Klassenelement-Funktionen der verschachtelten Klasse können nur über Objekte der verschachtelten Klasse aufgerufen werden:

  ```
  int i = ein.read ();
  ```

- Die Deklaration einer verschachtelten Klasse innerhalb der sie umgebenden Klasse erzeugt von vornherein kein *Objekt* dieser Klasse. (In der einschließenden Klasse kann aber ein Objekt

der verschachtelten Klasse definiert werden, über das dann der Aufruf von Klassenelement-Funktionen der verschachtelten Klasse erfolgen kann:)

```
class EinAusgabe
{
public:                                 // verschachtelte Klassen:
    class Eingabe
    {
    public:
        int read ();
    private:
        ...
    };

    class Ausgabe
    { // Deklarationen };

    Eingabe eing;                       // ist noch immer public!
};

EinAusgabe ea;
...
int j = ea.eing.read_ok ();
```

- Voll spezifizierte Namen in der Form

  ```
  int EinAusgabe::Eingabe::read ();
  ```

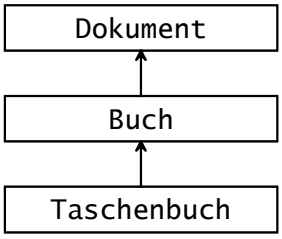
  können durch Verwendung einer **typedef**-Definition vereinfacht werden:

  ```
  typedef EinAusgabe::Eingabe EAE;
  int EAE::read();
  ```

5.5. Abgeleitete Klassen

5.5.1. Einfache Vererbung (*Inheritance*)

Eine Klasse, die eine Spezialisierung einer anderen Klasse darstellt, kann durch einen als *Vererbung* (*Inheritance*) bezeichneten Mechanismus aus dieser abgeleitet werden. Die einfachste Form ist die *einfache Vererbung*, bei der jede abgeleitete Klasse genau eine (direkte) *Basisklasse* hat:



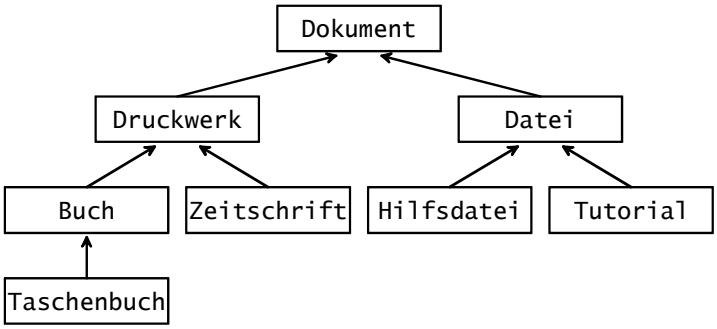

```
class Dokument
{
    // Liste der Elemente der Klasse
};

class Buch : public Dokument           // Buch ist von Dokument abgeleitet
{
    // Liste der Elemente der Klasse
};

class Taschenbuch : public Buch        // Taschenbuch ist von Buch abgeleitet
{
    // Liste der Elemente der Klasse
};
```

In diesem Beispiel ist `Dokument` eine *direkte Basisklasse* von `Buch` und eine *indirekte Basisklasse* von `Taschenbuch`. Eine Basisklasse muss immer vollständig definiert sein, bevor eine von ihr abgeleitete Klasse definiert werden kann; eine Vorwärtsdeklaration in der Form

```
class Dokument;
```

ist nicht ausreichend. Die Bedeutung der Zugriffsbezeichnung **`public`** in der Definition der abgeleiteten Klassen wird später behandelt (siehe Seite 127).

Eine Klasse kann als Basisklasse beliebig vieler abgeleiteter Klassen dienen:



Eine abgeleitete Klasse enthält alle Elemente der Basisklasse plus zusätzlicher Elemente, die für die abgeleitete Klasse deklariert wurden. Eine abgeleitete Klasse kann sich daher auf alle Elemente ihrer Basisklasse (sowie allfälliger indirekter Basisklassen) beziehen.

Wenn ein Element in der abgeleiteten Klasse neu definiert wurde, kann der Operator "**`::`**" verwendet werden, um zwischen dem Objekt der Basisklasse und dem der abgeleiteten Klasse zu unterscheiden.

Demo-Programm 5_05_01.cc

```
#include <iostream.h>
#include <string.h>

class Dokument                                  // Basisklasse
{
public:
    char *Name;                                 // Name des Dokuments
    void PrintName () { cout << Name << "\n"; }
};

class Buch : public Dokument                    // abgeleitet von Dokument
{
public:
    Buch (char *name, int seiten);              // Konstruktor
private:
    int Seitenzahl;
};

Buch::Buch (char *name, int seiten)             // Konstruktor
{
    Name = new char [strlen (name) + 1];
        // strlen: deklariert in string.h: String Length
    strcpy (Name, name);                        // kopieren
    Seitenzahl = seiten;
}

int main ()
{
    Buch Stroustrup ("The C++ Programming Language, 2nd Ed", 669);
    Stroustrup.PrintName ();
}
```

In diesem Demo-Programm greift der Konstruktor der Klasse `Buch` auf ein Element der Basisklasse `Dokument`, nämlich den Zeiger `Name`, zu. (Der Konstruktor verwendet den Operator **new**, um ein Feld der Type **char** zu allokieren, das groß genug für den als Argument übergebenen String `name` ist. Die Funktion `strlen` gibt die Länge eines Strings *ohne* das abschließende Null-Byte zurück, sodass zu ihrem Ergebnis 1 addiert werden muss, um die korrekte Feldgröße zu erhalten. Der als Argument übergebene String wird anschließend mit der Bibliotheksfunktion `strcpy` in den neu allokierten Speicherbereich kopiert, dessen Adresse im Element `Name` des Objekts "verankert" wird. Durch das Allokieren eines Speicherblocks und das Kopieren des Strings wird sichergestellt, dass das Objekt nicht unbeabsichtigt verändert werden kann, wenn der beim Aufruf des Konstruktors verwendete String später verändert wird. Die Funktionen `strlen` und `strcpy` werden in der *Include*-Datei `string.h` deklariert.) In `main()` wird ein Objekt der Klasse `Buch` namens `Stroustrup` definiert; die Ausgabe mit der in der Basisklasse `Dokument` definierten Funktion `PrintName()` ergibt:

```
The C++ Programming Language, 2nd Ed
```

Elemente der Basisklasse und der abgeleiteten Klasse können auf gleiche Weise verwendet werden. In der abgeleiteten Klasse können auch Elemente der Basisklasse neu definiert werden:

Demo-Programm 5_05_02.cc

```
#include <iostream.h>
#include <string.h>

class Dokument                                  // Basisklasse
{
public:
    char *Name;                                 // Name des Dokuments
    void PrintName () { cout << Name << "\n"; }
};

class Buch : public Dokument                    // Abgeleitet von Dokument
{
public:
    Buch (char *name, int seiten);              // Konstruktor
    void PrintName ();
```

```
private:
    int Seitenzahl;
};

Buch::Buch (char *name, int seiten)           // Konstruktor
{
    Name = new char [strlen (name) + 1];
    strcpy (Name, name);                      // kopieren
    Seitenzahl = seiten;
}

void Buch::PrintName ()
{
    cout << "Titel:  " << Name << "\nSeiten: " << Seitenzahl << "\n";
}

int main ()
{
    Buch Stroustrup ("The C++ Programming Language, 2nd Ed", 669);

    Stroustrup.PrintName ();
    Stroustrup.Dokument::PrintName ();
}
```

Hier wird in `main()` einmal die mit der Klasse `Buch` und dann die mit der Klasse `Dokument` definierte Funktion `PrintName()` aufgerufen. Die Ausgabe ist daher:

```
Titel:  The C++ Programming Language, 2nd Ed
Seiten: 669
The C++ Programming Language, 2nd Ed
```

Zeiger und Referenzen auf abgeleitete Klassen können implizit in Zeiger bzw. Referenzen auf die Basisklasse umgewandelt werden, sofern es eine zugängliche und eindeutige Basisklasse gibt. Das folgende Beispiel zeigt den Aufbau einer "Bibliothek" aus einer Reihe von Zeigern auf verschiedene abgeleitete Klassen entsprechend dem Beispiel von Seite 112, die in einer *heterogenen Liste* (bestehend aus Zeigern auf Objekte von unterschiedlichen Typen) zusammengefasst sind:

Demo-Programm 5_05_03.cc

```
#include <iostream.h>
#include <string.h>
#include <ctype.h>

class Dokument                          // Basisklasse für alle anderen Klassen
{
public:
    char *Name;                         // Name des Dokuments

    Dokument () { }                     // Default-Konstruktor
    Dokument (char *name)               // Konstruktor für char*
        { Speichern (name); }

    void PrintName ()                   // Default-Ausgabe
        { cout << Name << "\n"; }

    void Speichern (char *name)         // Hilfsfunktion
    {
        Name = new char [strlen (name) + 1];
        strcpy (Name, name);
    }
};

class Druckwerk : public Dokument
{
public:
    Druckwerk () { }
    Druckwerk (char *name) { Speichern (name); }
};
```

```
class Datei : public Dokument
{
public:
    Datei () { }
    Datei (char *name) { Speichern (name); }
    void PrintName () { cout << "Datei: " << Name << "\n"; }
};

class Buch : public Druckwerk
{
public:
    Buch () { }
    Buch (char *name) { Speichern (name); }
    void PrintName () { cout << "Buchtitel: "<< Name << "\n"; }
};

class Zeitschrift : public Druckwerk
{
public:
    Zeitschrift () { }
    Zeitschrift (char *name) { Speichern (name); }
    void PrintName () { cout << "Zeitschrift: " << Name << "\n"; }
};

class Taschenbuch : public Buch
{
public:
    Taschenbuch () { }
    Taschenbuch (char *name) { Speichern (name); }
};

class Hilfsdatei : public Datei
{
public:
    Hilfsdatei () { }
    Hilfsdatei (char *name) { Speichern (name); }
};

class Tutorial : public Datei
{
public:
    Tutorial () { }
    Tutorial (char *name) { Speichern (name); }
};

const int BibGroesse = 10;                    // Anzahl der Einträge

int main ()
{
    Dokument *Bibliothek [BibGroesse];        // "Datenbank"

    cout << "Typ der Eintragung:\n"
        "(D)okument, D(r)uckwerk, D(a)tei, (B)uch, (Z)eitschrift,\n"
        "(T)aschenbuch, (H)ilfsdatei, T(u)torial:\n\n";

    // lies BibGroesse Eintragungen für die "Bibliothek"

    for (int i = 0; i < BibGroesse; i++)
    {
        char cDtyp;                           // Schalt-Zeichen
        char buffer [128];

        cout << "Typ:   ";
        cin.getline (buffer, 128, '\n');
        cDtyp = *buffer;

        cout << "Titel: ";
```

```
        cin.getline (buffer, 128, '\n');

        // konvertiere cDtyp in Kleinbuchstaben

        switch (tolower (cDtyp))
        {
            // trage jede Eintragung mit dem korrekten Konstruktor ein

            case 'd':
                Bibliothek[i] = new Dokument (buffer);
                break;

            case 'r':
                Bibliothek[i] = new Druckwerk (buffer);
                break;

            case 'a':
                Bibliothek[i] = new Datei (buffer);
                break;

            case 'b':
                Bibliothek[i] = new Buch (buffer);
                break;

            case 'z':
                Bibliothek[i] = new Zeitschrift (buffer);
                break;

            case 't':
                Bibliothek[i] = new Taschenbuch (buffer);
                break;

            case 'h':
                Bibliothek[i] = new Hilfsdatei (buffer);
                break;

            case 'u':
                Bibliothek[i] = new Tutorial (buffer);
                break;

            default:
                i--;                            // Kompensation von "i++"
        }
    }

    for (i = 0; i < BibGroesse; i++)            // gib alle Eintragungen aus
        Bibliothek[i]->PrintName();
}
```

In diesem Demo-Programm ist die vollständige Klassen-Hierarchie der Abbildung auf Seite 112 definiert. Einzelne Klassen — `Dokument`, `Datei`, `Buch` und `Zeitschrift` — verwenden eigene `PrintName()`-Funktionen. *Alle* Klassen haben sowohl einen Default-Konstruktor, der keine Argumente benötigt, als auch einen Konstruktor mit einem Argument vom Typ **`char *`**, der — ähnlich wie bei den vorangegangenen Beispielen — den als Argument übergebenen String in neu allokierten Speicher kopiert. Die Angabe von Default-Konstruktoren ist bei Übersetzung des Programms mit dem GNU-C++-Compiler unbedingt erforderlich. Der Compiler generiert beim Aufruf eines Konstruktors einer abgeleiteten Klasse Aufrufe aller Konstruktoren der direkten und indirekten Basisklassen. Beim Aufruf von

GNU-C/C++-spezifisch

```
new Taschenbuch (buffer)
```

sind dies beispielsweise in der angegebenen Reihenfolge:

```
Taschenbuch (char *)
Buch ()
Druckwerk ()
Dokument ()
```

Erst danach wird der eigentliche Konstruktor (in diesem Fall die Funktion `Speichern()`) abgearbeitet. Wenn für eine Basisklasse ein Konstruktor mit *irgendeinem* Argument-Typ definiert wurde, aber kein Default-Konstruktor existiert, erfolgt beim GNU-C++-Compiler eine Fehlermeldung, und die Übersetzung des Moduls wird abgebrochen. (Da eine Basisklasse eine Teilmenge einer von ihr abgeleiteten Klasse ist, muss sichergestellt werden, dass die der Basisklasse angehörigen Elemente korrekt initialisiert werden.)

GNU-C/C++-spezifisch

Bei der Ausgabe der "Bibliotheksliste" mit diesem Demo-Programm wird allerdings nur die `PrintName()`-Funktion der Klasse `Dokument` verwendet, weil das Feld `Bibliothek` eben vom Typ `Dokument*` ist. Die Charakteristiken der abgeleiteten Klassen gehen also verloren. Diese Einschränkung kann durch Verwendung *virtueller Funktionen* (siehe weiter unten) behoben werden.

Das Demo-Programm verwendet für die Eingabe die Funktion

`cin.getline (char *buffer, int bufsize, char terminator)`.

Im Gegensatz zu `cin >> buffer` liest diese Funktion immer eine komplette Zeile einschließlich des Abschlußzeichens (hier `'\n'`) ein, während `cin >> buffer` die Eingabe beim ersten Trennzeichen (das heißt, beim ersten Leerschritt) abbricht. (Beachten Sie bitte, dass `cin.getline()` als drittes Argument einen Wert vom Typ **`char`** und nicht **`char*`** erwartet, dass also `"'\n'"` und nicht `""\n""` geschrieben werden muss.) Das "Schaltzeichen" `cDtyp` wird als erstes Zeichen in `buffer` nach der ersten Eingabe gewonnen und in einem **`switch`**-Befehl ausgewertet.

5.5.2. Virtuelle Funktionen

Die gemeinsamen Eigenschaften abgeleiteter Klassen werden im allgemeinen durch ihre Basisklasse repräsentiert; spezifische Eigenschaften einer Klasse werden in dieser selbst festgelegt. Dies trifft insbesondere auf gewisse Klassenelement-Funktionen zu, beispielsweise auf die Funktion `PrintName()` im Beispiel des vorangegangenen Abschnitts. Wenn Klassenelement-Funktionen als *virtuell* deklariert werden (mit dem Schlüsselwort **`virtual`**), dann wird vom Compiler automatisch bei der Erstellung eines Objekts einer (abgeleiteten) Klasse die erforderliche Information mit abgelegt, die es ermöglicht, auch dann die dieser Klasse entsprechende Klassenelement-Funktion aufzurufen, wenn nur ein *Zeiger* auf die *Basisklasse* übergeben wird. Die Deklaration der Klassen des Beispiels von Seite 114 unter Verwendung virtueller Funktionen sieht folgendermaßen aus:

Demo-Programm 5_05_04.cc

```
class Dokument                          // Basisklasse für alle anderen Klassen
{
public:
    char *Name;                                         // Name des Dokuments

    Dokument () { }                                     // Default-Konstruktor
    Dokument (char *name) { Speichern (name); }         // Konstruktor für char*

    virtual void PrintName () { cout << Name << "\n"; }    // Default-Ausgabe

    void Speichern (char *name)                         // Hilfsfunktion
        { Name = new char [strlen (name) + 1]; strcpy (Name, name); }
};

class Druckwerk : public Dokument
{
public:
    Druckwerk () { }
    Druckwerk (char *name) { Speichern (name); }
};

class Datei : public Dokument
{
public:
    Datei () { }
    Datei (char *name) { Speichern (name); }
    virtual void PrintName () { cout << "Datei: " << Name << "\n"; }
};
```

```
class Buch : public Druckwerk
{
public:
    Buch () { }
    Buch (char *name) { Speichern (name); }
    virtual void PrintName () { cout << "Buchtitel: " << Name << "\n"; }
};

class Zeitschrift : public Druckwerk
{
public:
    Zeitschrift () { }
    Zeitschrift (char *name) { Speichern (name); }
    virtual void PrintName () { cout << "Zeitschrift: " << Name << "\n"; }
};

class Taschenbuch : public Buch
{
public:
    Taschenbuch () { }
    Taschenbuch (char *name) { Speichern (name); }
};

class Hilfsdatei : public Datei
{
public:
    Hilfsdatei () { }
    Hilfsdatei (char *name) { Speichern (name); }
};

class Tutorial : public Datei
{
public:
    Tutorial () { }
    Tutorial (char *name) { Speichern (name); }
};
```

Unter Verwendung der Funktion `main()` des letzten Beispiels vom vorigen Abschnitt (Seite 115) kann mit dieser Klassen-Definition ein Programm erstellt werden, das — im Gegensatz zu der Variante von Seite 114 — alle Titel mit der passenden `PrintName()`-Funktion ausgibt. Für zehn Dokumente im weitesten Sinn mit den Typen

```
Dokument
Druckwerk
Datei
Buch
Zeitschrift
Taschenbuch
Hilfsdatei
Tutorial
Buch
Zeitschrift
```

erhält man mit dem Demo-Programm unter Verwendung virtueller Funktionen beispielsweise als Ausgabe:

```
Technisches Programmieren in C++
Microsoft Windows Software Development Kit
Datei: 5_05_03.cc
Buchtitel: Stroustrup: The C++ Programming Language
Zeitschrift: PC Magazine
Buchtitel: Hütte
Datei: GCC.INF
Datei: Windows - Getting Started
Buchtitel: Petzold: Programming Windows 3.1
Zeitschrift: Byte
```

Beachten Sie bitte, dass Versionen der Funktion `PrintName()` nur für vier Klassen (`Dokument`, `Datei`, `Buch` und `Zeitschrift`) angegeben wurden. Sofern keine neue Implementierung von `PrintName()` vorhanden ist, wird die Implementierung der jeweils nächsten Basisklasse verwendet (z.B. die von `Buch` für `Taschenbuch`).

Dieselbe Eingabe (mit gleichen Typen und Titeln) hätte mit dem Demo-Programm vom vorigen Abschnitt (Seite 114) ergeben:

```
Technisches Programmieren in C++
Microsoft Windows Software Development Kit
5_05_03.cc
Stroustrup: The C++ Programming Language
PC Magazine
Hütte
GCC.INF
Windows - Getting Started
Petzold: Programming Windows 3.1
Byte
```

Die wesentlichen Unterschiede zwischen den Versionen mit und ohne virtuelle Funktionen sind:

- Der Compiler reserviert für jedes Objekt jeder Klasse und jede Familie virtueller Funktionen zusätzlichen Speicherplatz, in dem er bei der Ausführung des Konstruktors einen Zeiger auf eine Tabelle ablegt, die (unter anderem) einen Funktionszeiger auf die korrekte virtuelle Funktion enthält.
- Bei der Ausführung einer virtuellen Funktion unter Übergabe eines Zeigers auf eine der Basisklassen wird aufgrund der im Objekt abgespeicherten Informationen der korrekte Funktionszeiger ausgewählt und die passende virtuelle Funktion ausgeführt.

Für virtuelle Funktionen gelten die folgenden Regeln:

- Alle Implementierungen einer virtuellen Funktion müssen die gleiche Ergebnistype haben.
- In einer Basisklasse kann eine virtuelle Funktion als *reine virtuelle Funktion* (*Pure Virtual Function*) in der Form

  ```
  virtual void PrintName () = 0;
  ```

  deklariert werden; in diesem Fall *muss* in *jeder* von der Basisklasse abgeleiteten Klasse eine geeignete Implementierung dieser Funktion vorgesehen werden.

Für den Aufruf einer Klassenelement-Funktion gilt grundsätzlich:

- Aufruf über ein Objekt (`Objekt.Funktion()`):

  Es wird immer die für die Klasse des Objekts gültige Implementierung der Funktion ausgeführt (Implementierung aus der eigenen Klasse oder der nächstgelegenen Basisklasse), gleichgültig, ob die Funktion virtuell ist oder nicht.
- Aufruf über Zeiger oder Referenz auf ein Objekt:
  - Virtuelle Funktion: Aufruf der Implementierung der Klassenelement-Funktion, die dem zugrundeliegenden Objekt entspricht.
  - Nicht-virtuelle Funktion: Aufruf der Implementierung der Klassenelement-Funktion, die der Klasse des zugrundeliegenden Zeigers oder der Referenz entspricht.

Das folgende Beispiel soll dieses Verhalten illustrieren:

Demo-Programm 5_05_05.cc

```
#include <iostream.h>

class Basis                                     // Basisklasse
{
public:
    virtual void Name ()                        // virtuell
        { cout << "Basis::Name()\n"; }

    void Aufruf ()                              // nicht-virtuell
        { cout << "Basis::Aufruf()\n"; }
};
```

```
class Abgeleitet : public Basis
{
public:
    void Name ()                                    // virtuell
        { cout << "Abgeleitet::Name()\n"; }

    void Aufruf ()                                  // nicht-virtuell
        { cout << "Abgeleitet::Aufruf()\n"; }
};

int main ()
{
    Basis oBasis;                       // Objekte von Basis und Abgeleitet
    Abgeleitet oAbgeleitet;

    Basis *poBasis1 = & oBasis;         // Zeiger
    Basis *poBasis2 = & oAbgeleitet;
    Abgeleitet *poAbgeleitet = & oAbgeleitet;

    // Funktionsaufrufe:
    cout << "Objekte:\n";
    oBasis.Name();
    oBasis.Aufruf();
    oAbgeleitet.Name();
    oAbgeleitet.Aufruf();

    cout << "Zeiger vom Typ Basis:\n";
    poBasis1->Name();
    poBasis1->Aufruf();
    poBasis2->Name();
    poBasis2->Aufruf();

    cout << "Zeiger vom Typ Abgeleitet:\n";
    poAbgeleitet->Name();
    poAbgeleitet->Aufruf();
};
```

Das Programm produziert die Ausgabe:

```
Objekte:
Basis::Name()
Basis::Aufruf()
Abgeleitet::Name()
Abgeleitet::Aufruf()
Zeiger vom Typ Basis:
Basis::Name()
Basis::Aufruf()
Abgeleitet::Name()
Basis::Aufruf()
Zeiger vom Typ Abgeleitet:
Abgeleitet::Name()
Abgeleitet::Aufruf()
```

Die virtuelle Funktion `Name()` wird auch bei Verwendung eines Zeigers vom Typ `Basis` auf ein Objekt der Klasse `Abgeleitet` mit der korrekten Implementierung ausgeführt, die nicht-virtuelle Funktion `Aufruf()` hingegen nicht.

Die Verwendung des Schlüsselwortes **`virtual`** ist nur bei Klassenelement-Funktionen zulässig. Seine Verwendung bei der Deklaration einer virtuellen Funktion in einer *abgeleiteten* Klasse ist optional; alle Funktionen in einer abgeleiteten Klasse, die eine virtuelle Funktion in einer Basisklasse neu definieren, sind *a priori* virtuell. Virtuelle Funktionen in einer Basisklasse *müssen definiert* werden, außer, sie wurden als *reine virtuelle Funktionen* deklariert.

Der Aufrufmechanismus virtueller Funktionen kann unter Verwendung des Operators "**`::`**" und der Angabe der Ziel-Klasse ausgeschaltet werden.

Mit den vorangehenden Definitionen der Klassen `Basis` und `Abgeleitet` kann man dann schreiben:

Demo-Programm 5_05_06.cc

```
int main ()
{
    Abgeleitet oAbgeleitet;
    Abgeleitet *poAbgeleitet = & oAbgeleitet;

    cout << "Objekte:\n";
    oAbgeleitet.Name();
    oAbgeleitet.Aufruf();
    oAbgeleitet.Basis::Name();
    oAbgeleitet.Basis::Aufruf();
    cout << "Zeiger vom Typ Abgeleitet:\n";
    poAbgeleitet->Name();
    poAbgeleitet->Aufruf();
    poAbgeleitet->Basis::Name();
    poAbgeleitet->Basis::Aufruf();
};
```

Eine Klassenauswahl in umgekehrter Richtung (`oBasis.Abgeleitet::Name()`) ist unzulässig. Das Programm erstellt die Ausgabe:

```
Objekte:
Abgeleitet::Name()
Abgeleitet::Aufruf()
Basis::Name()
Basis::Aufruf()
Zeiger vom Typ Abgeleitet:
Abgeleitet::Name()
Abgeleitet::Aufruf()
Basis::Name()
Basis::Aufruf()
```

5.5.3. Abstrakte Klassen

Abstrakte Klassen sind solche, die mindestens eine *reine virtuelle Funktion* enthalten oder von einer abstrakten Basisklasse abgeleitet wurden, ohne eine Implementierung für deren reine virtuelle Funktionen zu beinhalten. Sie können verwendet werden, um ein Protokoll zu erzwingen (beispielsweise einen Satz von spezifischen Funktionen für jede von ihnen abgeleitete Klasse). Abstrakte Klassen existieren ausschließlich als Basisklassen für abgeleitete Klassen; es ist nicht zulässig, ein Objekt einer abstrakten Klasse zu definieren.

Wenn in unseren "Bibliotheks"-Demo-Programmen von Seite 114 und 117 die Klasse `Dokument` in der folgenden Form deklariert worden wäre (bei identischer Deklaration aller abgeleiteten Klassen), dann hätte dies die folgenden Konsequenzen:

Demo-Programm 5_05_07.cc

```
class Dokument
{
public:
    char *Name;                          // Name des Dokuments

    Dokument () { }                                  // Default-Konstruktor
    Dokument (char *name) { Speichern (name); }      // Konstruktor für char*

    virtual void PrintName () = 0;       // reine virtuelle Ausgabe-Funktion

    void Speichern (char *name)          // Hilfsfunktion
    {
        Name = new char [strlen (name) + 1];
        strcpy (Name, name);
    }
};
```

➡ Die Klassen `Dokument` und `Druckwerk` wären *abstrakte* Klassen; es dürften keine Objekte dieser Klassen (beispielsweise im **`switch`**-Befehl in `main()`) generiert werden.

- Für alle anderen abgeleiteten Klassen unseres Beispiels würde sich nichts ändern; es wäre aber nicht möglich, eine Klasse zu definieren, die direkt von `Dokument` oder `Druckwerk` abgeleitet wäre, aber keine `PrintName()`-Funktion definiert hätte.
- Obwohl es unzulässig ist, *Objekte* einer abstrakten Klasse zu definieren, können *Zeiger* und *Referenzen* auf eine abstrakte Klasse verwendet werden. Es ist daher nach wie vor möglich, in unserem Demo-Programm die *heterogene Liste* der Bibliothekseintragungen mit

  ```
  Dokument *Bibliothek [BibGroesse];
  ```

  zu definieren und auf ihre Elemente zuzugreifen.

Weiters ist die Verwendung abstrakter Klassen unzulässig für:

- Funktionsargumente und Funktionsergebnisse;
- Explizite Typkonversion.
- Bei direktem oder indirektem Aufruf einer reinen virtuellen Funktion durch den Konstruktor einer Klasse ist das Ergebnis undefiniert.

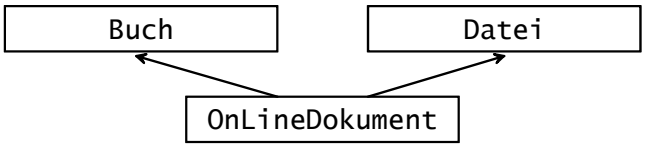
Reine virtuelle Funktionen können in einer abstrakten Klasse auch *definiert* werden; sie können jedoch nur unter Verwendung der Syntax

<Name der abstrakten Klasse>`::`*<Funktion>*`()`

aufgerufen werden. Damit ist es beispielsweise möglich, reine virtuelle Destruktoren für eine abstrakte Klasse zu definieren. (Ähnlich wie bei Konstruktoren sieht C++ auch automatische Aufrufe von Destruktoren vor, sofern solche vorhanden sind. Eine Definition eines reinen virtuellen Destruktors, allenfalls auch als leere Funktion, stellt sicher, dass jede abgeleitete Klasse einen Destruktor spezifizieren muss, dass aber auch ein Aufruf des Destruktors der abstrakten Basisklasse möglich ist.)

5.5.4. Mehrfache Vererbung

Eine abgeleitete Klasse in C++ kann auch *mehrere* direkte Basisklassen haben. Im folgenden Beispiel ist die Klasse `OnLineDokument` aus den Klassen `Buch` und `Datei` abgeleitet:

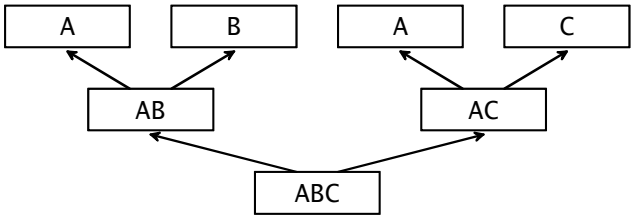



```
class Buch
{
    // Klassenelemente
};

class Datei
{
    // Klassenelemente
}

class OnLineDokument : public Buch, public Datei
{
    // neue Klassenelemente
}
```

Die Klasse `OnLineDokument` vereinigt die Eigenschaften ihrer beiden Basisklassen. Die Reihenfolge, in der die Basisklassen deklariert werden, bestimmt die Reihenfolge, in der die Konstruktoren der Basisklassen bei der Definition eines Objekts der abgeleiteten Klasse aufgerufen werden, und die umgekehrte Reihenfolge des Aufrufs der Destruktoren.

5.5.5. Mehrfache Basisklassen

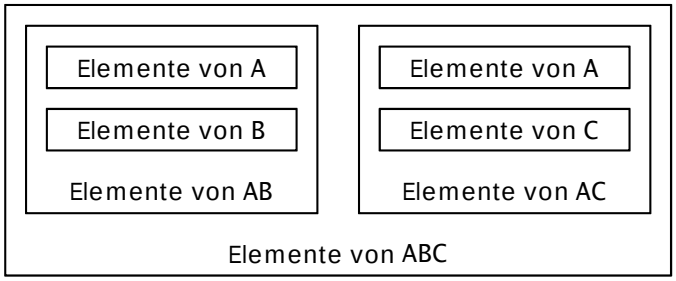
Bei Verwendung mehrfacher Vererbung zur Erstellung abgeleiteter Klassen kann dieselbe Klasse mehrfach als indirekte (aber nicht als direkte) Basisklasse aufscheinen:



Diese Klassenstruktur kann folgendermaßen deklariert werden:

```
class A { /* Klassenelemente von A */ };
class B { /* Klassenelemente von B */ };
class C { /* Klassenelemente von C */ };

class AB : public A, public B
{
    // Klassenelemente von AB
}

class AC : public A, public C
{
    // Klassenelemente von AC
}

class ABC : public AB, public AC
{
    // Klassenelemente von ABC
}
```

Die logische Struktur der abgeleiteten Klasse ABC ist dann:

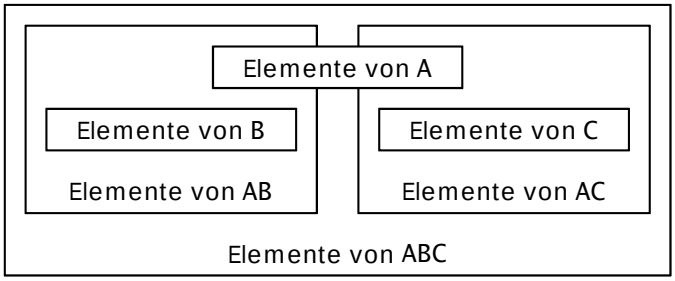


5.5.6. Virtuelle Basisklassen

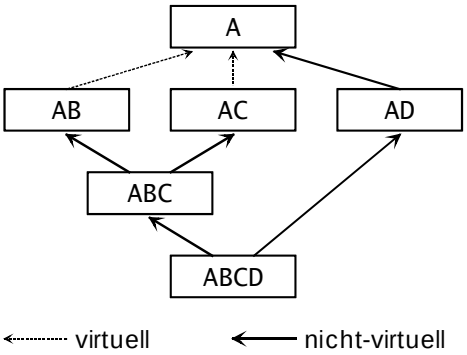
Im obigen Beispiel enthält ein Objekt der Klasse ABC *zweimal* die Elemente der Klasse A. Dies kann dann zu Mehrdeutigkeiten führen, wenn ein Element der Klasse A angesprochen werden soll. Sowohl die Effizienz der Datenspeicherung als auch die Eindeutigkeit der Zuordnung kann verbessert werden, wenn die Klasse A als *virtuelle* Basisklasse der Klassen AB und AC deklariert wird. In diesem Fall wird für ein Objekt der Klasse ABC nur *ein* Satz der Elemente der Klasse A angelegt:

```
class AB : virtual public A, public B
{
    // Klassenelemente von AB
}
```

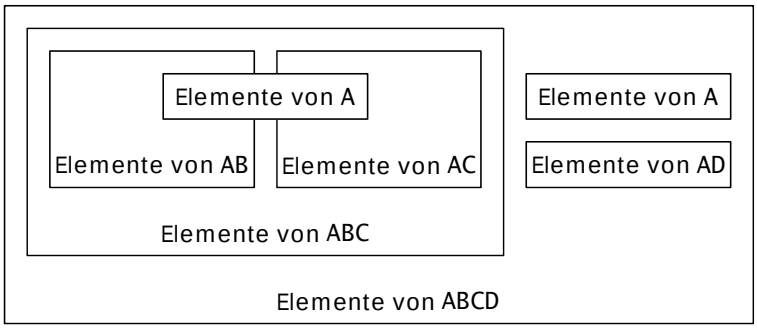
```
class AC : virtual public A, public C
{
    // Klassenelemente von AC
}
```



Eine Klasse kann auch eine virtuelle und eine nicht-virtuelle Komponente derselben Basistype haben:



Dieser Vererbungsstruktur entspricht die folgende Struktur eines Objektes der Klasse ABCD:



Unter bestimmten Voraussetzungen kann die Verwendung virtueller Basisklassen jedoch einen zusätzlichen Overhead an Programmcode und Datengröße bedeuten.

5.5.7. Mehrdeutigkeiten

Im Zuge einer mehrfachen Vererbung kann eine Klasse den selben Namen auf mehr als einem Pfad erben, wobei die Implementierungen des Namens nicht notwendigerweise die gleichen zu sein brauchen — es entsteht so *Mehrdeutigkeit* (*Ambiguity*).

Im folgenden Beispiel werden die Namen `a` und `b` in den beiden Klassen `A` und `B` mit unterschiedlichen Bedeutungen definiert:

```
class A                                 // erste Basisklasse
{
    unsigned a;
    unsigned b();
};
```

```
class B                                 // zweite Basisklasse
{
    unsigned a();                       // anderes "a"!
    int b();                            // anderes "b"!
};

class C : public A, public B            // abgeleitet
{
};

C *pC = new C;
...
pC->b();                                // WELCHES b()?
```

Das obige Beispiel führt zu einer Fehlermeldung des Compilers, weil nicht klar ist, welche Implementierung von `b()` aufzurufen ist. Der Compiler testet auf Mehrdeutigkeiten in der folgenden Reihenfolge:

- Wenn sich ein Name auf mehr als eine Funktion, eine Type, ein Objekt oder einen Enumerator beziehen kann, erfolgt eine Fehlermeldung.
- Bezieht sich der Name auf *overloaded* Funktionen und kann die passende Funktion eindeutig aufgrund der Argumentliste gewählt werden, wählt der Compiler diese Funktion.
- Werden Zugriffsrechte auf ein Klassenelement verletzt, erfolgt wiederum eine Fehlermeldung.

Die Mehrdeutigkeit im vorigen Beispiel kann durch vollständige Spezifikation des Klassennamens beseitigt werden; die folgende Zeile wird fehlerfrei übersetzt:

```
pC->B::b();
```

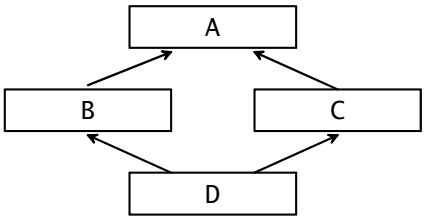
Bei Verwendung virtueller Basisklassen existiert nur *ein* Satz der Elemente der virtuellen Basisklasse in der abgeleiteten Klasse. Jeder Zugriff auf ein Element der virtuellen Basisklasse ist daher eindeutig. Im Gegensatz dazu ist bei nicht-virtueller Deklaration der Basisklasse jeder Zugriff auf ein Element der Basisklasse mehrdeutig.

Wenn ein Name in einer Basisklasse und in einer abgeleiteten Klasse definiert wurde, *dominiert* die Implementierung des Namens in der abgeleiteten Klasse. Im Falle einer potentiellen Mehrdeutigkeit wird immer die Implementierung der abgeleiteten Klasse verwendet:

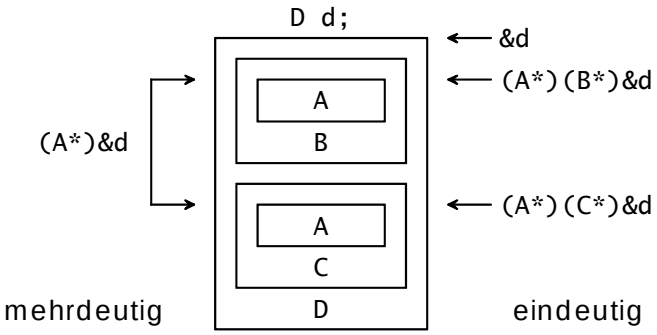
```
class A
{
public:
    int a;
};

class B : public virtual A
{
public:
    int a();
};

class C : public virtual A
{
    // Klassenelemente
};

class D: public B, public C
{
public:
    D() { a(); }                // eindeutig: B::a() dominiert über A::a
};
```

Im obigen Beispiel wäre jedoch die Definition des Konstruktors von Klasse `D` mehrdeutig gewesen, wenn Klasse `B` und/oder Klasse `C` *nicht* als virtuell deklariert worden wäre.

Auch die explizite Konversion von Zeigern oder Referenzen auf Klassen kann Mehrdeutigkeiten verursachen, wenn nicht-virtuelle Basisklassen verwendet werden:



Für ein Objekt der Klasse D gelten dann die folgenden Zeiger:



Nur durch eine explizite Angabe des Subobjektes, auf das sich der Zeiger auf A beziehen soll (wie in der rechten Hälfte der obigen Darstellung), kann eine eindeutige Zuordnung getroffen werden.

5.6. Zugriff auf Klassenelemente

5.6.1. Zugriffskontrolle

In C++ existieren drei durch Schlüsselworte steuerbare Typen des Zugriffs auf die Elemente einer Klasse:

- **`private`**:

  Klassenelemente, die als **`private`** deklariert wurden, können nur von Klassenelement-Funktionen der Klasse und von Klassen und Funktionen, die als **`friend`** der Klasse deklariert wurden, benützt werden.

- **`protected`**:

  Klassenelemente, die als **`protected`** deklariert wurden, können von Klassenelement-Funktionen der Klasse und von Klassen und Funktionen, die als **`friend`** der Klasse deklariert wurden, sowie von Klassen, die von der Klasse abgeleitet wurden, benützt werden.

- **`public`**:

  Klassenelemente, die als **`public`** deklariert wurden, können von beliebigen Funktionen benützt werden.

Diese Zugriffskontrolle kann durch explizite Typenumwandlung umgangen werden. Die Zugriffskontrolle betrifft alle Namen (Funktions- und Datenelemente, eingeschlossene Klassen und Enumeratoren) in gleicher Weise.

Ohne Verwendung eines der drei Zugriffs-Schlüsselworte ist der Zugriff auf Klassentypen, die mit **`struct`** oder **`union`** deklariert wurden, **`public`**, und auf solche, die mit **`class`** deklariert wurden, **`private`**. Alle Klassenelemente, die in der Deklaration der Klasse einem der Zugriffs-Schlüsselworte folgen, unterliegen dem entsprechenden Zugriffstyp. Diese Spezifikation endet mit dem nächsten Zugriffs-Schlüsselwort oder dem Ende der Deklaration. Es können beliebig viele Zugriffs-Schlüsselworte in der Deklaration einer Klasse verwendet werden. Die folgenden beiden Deklarationen sind äquivalent:

```
class Punkt
{
public:
    int& x(int);
    int& y(int);
private:
    int _x, _y;
};
```

```
class Punkt
{
    int _x;
public:
    int& x(int);
private:
    int _y;
public:
    int& y(int);
};
```

Für abgeleitete Klassen hängt der Zugriff auf die Elemente einer Basisklasse von den Zugriffsrechten innerhalb der Basisklasse und von dem bei der Ableitung spezifizierten Zugriffsrecht ab:

	Zugriffsrecht bei der Ableitung:		
in Basisklasse:	**`private`**	**`protected`**	**`public`**
`private`	immer unzugänglich		
`protected`	**`private`**	**`protected`**	**`protected`**
`public`	**`private`**	**`protected`**	**`public`**

Das folgende Beispiel illustriert die Zugriffsrechte bei abgeleiteten Klassen:

Demo-Programm 5_06_01.cc

```
class Basis
{
public:
    int Publ_Element;
protected:
    int Prot_Element;
private:
    int Priv_Element;
};

class Publ_Abl : public Basis
{
    int i, j, k;
public:
    Publ_Abl_Func()
    {
        i = Publ_Element;
        j = Prot_Element;
        k = Priv_Element;        // unzulässig
    }
};

class Prot_Abl : protected Basis
{
    int i, j, k;
public:
    Prot_Abl_Func()
    {
        i = Publ_Element;
        j = Prot_Element;
        k = Priv_Element;        // unzulässig
    }
};

class Priv_Abl : private Basis
{
    int i, j, k;
public:
    Priv_Abl_Func()
    {
        i = Publ_Element;
        j = Prot_Element;
        k = Priv_Element;        // unzulässig
    }
};

class Publ_Abl_Abl : public Publ_Abl
{
    int i, j, k;
public:
    Publ_Abl_Abl_Func()
    {
        Publ_Abl_Func();
        i = Publ_Element;
        j = Prot_Element;
        k = Priv_Element;        // unzulässig
    }
};

class Prot_Abl_Abl : protected Prot_Abl
{
    int i, j, k;
public:
    Prot_Abl_Abl_Func ()
    {
        Prot_Abl_Func();
        i = Publ_Element;
```

```
        j = Prot_Element;
        k = Priv_Element;        // unzulässig
    }
};

class Priv_Abl_Abl : private Priv_Abl
{
    int i, j, k;
public:
    Priv_Abl_Abl_Func ()
    {
        Priv_Abl_Func();
        i = Publ_Element;        // unzulässig
        j = Prot_Element;        // unzulässig
        k = Priv_Element;        // unzulässig
    }
};

int main ()
{
    // Objekte abgeleiteter Klassen:
    Publ_Abl Publ;
    Prot_Abl Prot;
    Priv_Abl Priv;
    Publ_Abl_Abl PublA;
    Prot_Abl_Abl ProtA;
    Priv_Abl_Abl PrivA;

    // Funktionsaufrufe:
    Publ.Publ_Abl_Func();
    Prot.Prot_Abl_Func();
    Priv.Priv_Abl_Func();
    PublA.Publ_Abl_Func();
    ProtA.Prot_Abl_Func();
    PrivA.Priv_Abl_Func();
    PublA.Publ_Abl_Abl_Func();
    ProtA.Prot_Abl_Abl_Func();
    PrivA.Priv_Abl_Abl_Func();


    // Zugriffe auf Elemente der Basisklasse
    int i1 = Publ.Publ_Element;
    int j1 = Publ.Prot_Element;  // unzulässig
    int k1 = Publ.Priv_Element;  // unzulässig

    int i2 = Prot.Publ_Element;
    int j2 = Prot.Prot_Element;  // unzulässig
    int k2 = Prot.Priv_Element;  // unzulässig

    int i3 = Priv.Publ_Element;
    int j3 = Priv.Prot_Element;  // unzulässig
    int k3 = Priv.Priv_Element;  // unzulässig

    int i4 = PublA.Publ_Element;
    int j4 = PublA.Prot_Element; // unzulässig
    int k4 = PublA.Priv_Element; // unzulässig

    int i5 = ProtA.Publ_Element; // unzulässig
    int j5 = ProtA.Prot_Element; // unzulässig
    int k5 = ProtA.Priv_Element; // unzulässig

    int i6 = PrivA.Publ_Element; // unzulässig
    int j6 = PrivA.Prot_Element; // unzulässig
    int k6 = PrivA.Priv_Element; // unzulässig
}
```

Demo-Programm 5_06_02.cc

GNU-C/C++-spezifisch

Das vorangegangene Beispiel zeigt, welche Zugriffe auf Elemente der Basisklasse und von davon abgeleiteten Klassen zulässig sind, je nachdem, welcher Zugriffs-Typ für die Elemente und für die Ableitung der abgeleiteten Klassen verwendet wurde. Exakt das gleiche Verhalten ergibt sich, wenn statt der Datenelemente der Klasse `Basis` Funktionen verwendet worden wären. (Die Angaben über unzulässige Zugriffe beziehen sich auf den GNU C++-Compiler; sie entsprechen nicht vollständig der "offiziellen" C++-Definition. Bei anderen Compilern kann daher ein geringfügig abweichendes Verhalten erwartet werden.)

Bei der Definition einer abgeleiteten Klasse darf das Schlüsselwort für den Zugriffs-Typ auf die Basisklasse weggelassen werden. In diesem Fall wird der gleiche Zugriffs-Typ verwendet, der standardmäßig für den definierten Klassentyp gilt:

```
class Abgeleitet : Basis
```

ist äquivalent zu:

```
class Abgeleitet : private Basis
```

und

```
struct Abgeleitet : Basis
```

ist äquivalent zu:

```
struct Abgeleitet : public Basis
```

5.6.2. Das Schlüsselwort **friend**

Das Schlüsselwort **friend** erlaubt einen Zugriff externer Funktionen oder Klassen auf Elemente einer Klasse, die als **protected** oder **private** deklariert wurden.

5.6.2.1. **friend**-Funktionen

Funktionen, die mit **friend** in der Definition einer Klasse deklariert wurden, gelten nicht als Elemente der Klasse. Sie sind vielmehr "gewöhnliche" Funktionen, die auch nicht mit dem Namen eines Klassen-Objekts und dem Operator "`.`" oder "`->`" aufgerufen zu werden brauchen. Das folgende Beispiel illustriert eine Klasse `Punkt` und einen *overloaded* Operator **operator+** (siehe Seite 163):

Demo-Programm 5_06_03.cc

```
#include <iostream.h>

class Punkt
{
public:
    Punkt (short x, short y)
        { _x = x; _y = y; }
    void Print()
        { cout << "Punkt (" << _x << ", " << _y << ")\n"; }
private:
    short _x, _y;

    friend Punkt operator+(Punkt&, int);            // friend-Deklaration
    friend Punkt operator+(int, Punkt&);            // friend-Deklaration
};

// Definition der friend-Funktionen: Operation "Punkt + Offset"

Punkt operator+ (Punkt& pt, int offset)
{
    Punkt temp = pt;

    temp._x += offset;          // direkter Zugriff auf private Elemente
    temp._y += offset;

    return temp;
}
```

```
// Definition der friend-Funktionen: Operation "Offset + Punkt"

Punkt operator+(int offset, Punkt& pt)
{
    Punkt temp = pt;

    temp._x += offset;          // direkter Zugriff auf private Elemente
    temp._y += offset;

    return temp;
}

int main ()
{
    Punkt p(10, 20);
    p.Print();

    p = p + 5;                  // ruft 1. operator+() auf
    p.Print();

    p = 3 + p;                  // ruft 2. operator+() auf
    p.Print();
}
```

Das Programm erzeugt die Ausgabe:

```
Punkt (10, 20)
Punkt (15, 25)
Punkt (18, 28)
```

Es ist gleichgültig, *wo* innerhalb der Definition einer Klasse die Deklaration von **friend**s erfolgt. Es müssen aber immer *alle* **friend**-Funktionen, unabhängig davon, ob sie gleiche oder unterschiedliche Namen haben, explizit als **friend** deklariert werden (in unserem Fall die beiden *overloaded Operator*-Funktionen).

In unserem Beispiel war es zweckmäßig, die Benutzer-Definition des Operators "**+**" nicht in einer Klassenelement-Funktion, sondern in einer externen Funktion vorzunehmen: Um die Kommutativität der Addition zu erhalten, wurden hier zwei **operator+**-Funktionen definiert, bei denen das `Punkt`- und das **int**-Argument einmal an erster und einmal an zweiter Stelle stehen. *Overloaded Operator*-Funktionen, die als Klassenelement-Funktionen definiert wurden, können nur *ein* explizites Argument haben (in diesem Fall den **int**-Offset, der zum `Punkt`-Objekt addiert werden soll; siehe Seite 163). Die Kommutativität der Additions-Operation wäre damit nicht mehr gegeben.

5.6.2.2. Klassen und Klassenelement-Funktionen als **friend**s

Entweder einzelne Klassenelement-Funktionen oder ganze Klassen können als **friend** einer Klasse deklariert werden; sie haben dann die gleichen Zugriffsrechte auf Elemente dieser Klasse wie deren eigene Funktionen:

Demo-Programm 5_06_04.cc

```
class A;                        // Vorwärts-Deklaration

class B                         // Deklaration ohne Definition der Funktionen
{
public:
    int Func1(A);
    int Func2(A);
};
```

```
class A
{
private:
    int _a;
    friend int B::Func1(A);
    friend class C;
};

// Definition der Funktionen in Klasse B:

int B::Func1(A a) { return a._a; }      // Ok: B::Func1 ist friend von A

int B::Func2(A a) { return a._a; }      // unzulässig: Func2 ist nicht friend

class C
{
public:
    int Func3(A a) { return a._a; }    // Ok: Klasse C ist friend von A
};

class D : public C
{
public:
    int Func4(A a) { return a._a; }
                    // unzulässig: Freundschaften können nicht vererbt werden
};
```

In diesem Beispiel wird eine Klassenelement-Funktion (`B::Func1(A)`) und eine Klasse (`C`) als **`friend`** der Klasse `A` deklariert. Sowohl diese Funktion in Klasse `B` als auch *alle* Element-Funktionen von Klasse `C` haben Zugriff auf das als **`private`** deklarierte Element `_a` von `A` (genau so, als ob sie Elemente von Klasse `A` wären). Diese Rechte gehen jedoch nicht auf abgeleitete Klassen (z.B. Klasse `D`) über. Beachten Sie bitte die Reihenfolge, in der die Klassen definiert wurden: Klasse `A` wurde voraus als Klasse *deklariert*, aber nicht *definiert*, um die Definition von Klasse `B` zu ermöglichen. Klasse `B` musste mit allen ihren Elementen *definiert* werden, damit die Spezifikation von `B::Func1(A)` als **`friend`** in der nachfolgenden Definition von Klasse `A` zulässig war. (Klassen, aus denen einzelne Element-Funktionen als **`friend`** deklariert werden sollen, müssen zum Zeitpunkt der Deklaration als **`friend`** bereits definiert sein.) Die Definition der Funktionen `B::Func1(A)` und `B::Func2(A)` war wiederum erst möglich, nachdem Klasse `A` (und ihr Element `_a`) definiert worden war. Unproblematisch ist die Definition ganzer Klassen als **`friend`**: Wenn die als **`friend`** zu deklarierende Klasse zum Zeitpunkt dieser Deklaration bereits definiert war, erübrigt sich die Verwendung des Schlüsselwortes "**`class`**" in der Deklaration als **`friend`**; im übrigen kann eine ganze als **`friend`** zu deklarierende Klasse vor oder nach dieser Deklaration definiert werden.

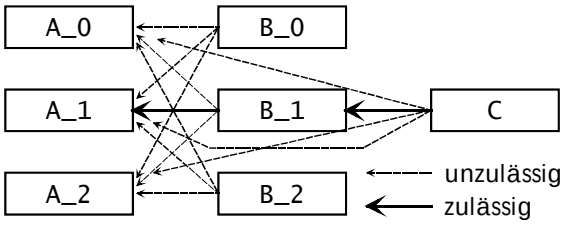
Das folgende Beispiel illustriert die Wirkung von **`friend`** bei abgeleiteten Klassen und "Freunden von Freunden":

Demo-Programm 5_06_05.cc

```
class A_0                                   // Basisklasse, Familie "A"
{
protected:
    int a_0;
};

class A_1 : public A_0                      // abgeleitet
{
protected:
    int a_1;

friend class B_1;
};

class A_2 : public A_1                      // abgeleitet
{
protected:
    int a_2;
};
```

```
class B_0                                       // Basisklasse, Familie "B"
{
public:
    int b_00 (A_0 x) { return x.a_0; }          // unzulässig
    int b_01 (A_1 x) { return x.a_1; }          // unzulässig
    int b_02 (A_2 x) { return x.a_2; }          // unzulässig
};

class B_1 : public B_0                          // abgeleitet
{
public:
    int b_10 (A_0 x) { return x.a_0; }          // unzulässig
    int b_11 (A_1 x) { return x.a_1; }          // einzig zulässiger Zugriff
    int b_12 (A_2 x) { return x.a_2; }          // unzulässig
friend class C;
};

class B_2 : public B_1                          // abgeleitet
{
public:
    int b_20 (A_0 x) { return x.a_0; }          // unzulässig
    int b_21 (A_1 x) { return x.a_1; }          // unzulässig
    int b_22 (A_2 x) { return x.a_2; }          // unzulässig
};

class C                                         // "Freund des Freundes" von A
{
public:
    int c_00 (A_0 x) { return x.a_0; }          // unzulässig
    int c_01 (A_1 x) { return x.a_1; }          // unzulässig
    int c_02 (A_2 x) { return x.a_2; }          // unzulässig
};
```

Die einzige in den Klassen `B_0`, `B_1`, `B_2` und `C` definierte Funktion, für die in diesem Demo-Programm *keine* Fehlermeldung auftritt, ist die Funktion `B_1::b11(A_1)`: Die Deklaration von Klasse `B_1` als **`friend`** von Klasse `A_1` sichert einzig und allein den Zugriff von Funktionen von Klasse `B_1` auf Elemente von Klasse `A_1`; sie schließt weder Zugriffsrechte auf Basisklassen oder abgeleitete Klassen (also `A_0` bzw. `A_2`) ein, noch sind die Zugriffsrechte von Klasse `B_1` auf die von ihr abgeleiteten Klassen vererbbar. Ebenso wenig existiert ein Zugriffsrecht für "Freunde von Freunden": Klasse `C` hat zwar Zugriff auf alle Elemente von Klasse `B_1`; die Zugriffsrechte von Klasse `B_1` auf Klasse `A_1` gehen jedoch nicht auf Klasse `C` über.



5.6.3. Zugriff auf virtuelle Funktionen

Die Zugriffsrechte auf virtuelle Funktionen hängen von der Type des Objekts, Zeigers oder der Referenz ab, mit dem bzw. der der Funktionsaufruf vorgenommen wird:

Demo-Programm 5_06_06.cc

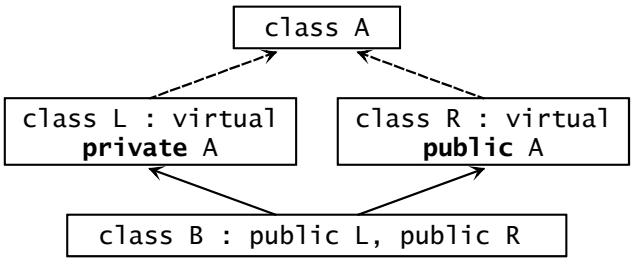
```
class Basis
{
public:
    virtual int Status() { return stat; }
protected:
    int stat;
};
```

```
class Abgeleitet : public Basis
{
private:
    int Status() { return stat << 1; }
};

int main ()
{
    Abgeleitet abg;
    Basis *pbabg = &abg;
    Abgeleitet *paabg = &abg;

    int i = abg.Status();        // Fehler: Abgeleitet::Status() private
    i = pbabg->Status();         // Ok: Basis::Status() ist public
    i = paabg->Status();         // Fehler: Abgeleitet::Status() private
};
```

Nichtsdestoweniger wird beim (legalen) Aufruf von `pbabg->Status()` nicht die Funktion `Status()` in Klasse `Basis`, sondern die Funktion `Status()` in Klasse `Abgeleitet` aufgerufen. Die *Auswahl* der passenden virtuellen Funktion erfolgt also aufgrund des zugrundeliegenden *Objekts*, während die Zugriffsrechte aufgrund der für den Aufruf verwendeten *Type* ausgewertet werden.

5.6.4. Unterschiedliche Zugriffs-Pfade

Bei mehrfacher Vererbung mit virtuellen Basisklassen können mehrere Pfade zu einem Namen der Basisklasse führen. In einem derartigen Fall wählt der Compiler jenen Pfad, der die höchsten Zugriffsrechte ergibt:



Im obigen Beispiel wird ein Element in Klasse `A` immer über den rechten Pfad (also via Klasse `R`) erreicht, weil in Klasse `R` die Basisklasse `A` als **`public`** deklariert wurde, während sie in Klasse `L` als **`private`** deklariert wurde.

5.7. Schablonen (*Templates*)

Schablonen (*Templates*) stellen eine Verallgemeinerung einer Klasse oder Funktion dar, wobei die Type einzelner Klassenelemente oder Funktionsargumente offen bleibt. Damit kann eine ganze Familie von Klassen oder *overloaded* Funktionen definiert werden, deren spezielle typmäßige Implementierung erst bei der Definition von Objekten oder beim Aufruf der Funktion mit Argumenten eines bestimmten Typs festgelegt wird. Damit verhalten sich Schablonen ähnlich wie Makros, erlauben jedoch im Gegensatz zu diesen die volle Typprüfung von C++. Schablonen sind nicht in allen Implementierungen von C++ verfügbar (sie sind es in GNU C++, waren es aber z.B. nicht in Microsoft Visual C++ Version 1.x).

GNU-C/C++-spezifisch

Um die Wirkungsweise einer Schablone zu veranschaulichen, soll ein Makro (zur Berechnung des Maximalwertes zweier Argumente) herangezogen werden:

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

Beim Aufruf dieses Makros setzt der Präprozessor die korrespondierenden aktuellen Argumente anstelle der formalen Argumente `a` und `b` in das Quell-Programm ein; die aktuellen Argumente können von beliebigem Typ sein, solange ihre Typen zueinander kompatibel sind. Schablonen haben etwa die gleiche Funktionalität; es erfolgt aber auf jeden Fall eine rigorose Prüfung der Datentypen.

5.7.1. Schablonen für Klassen

Schablonen für Klassen werden ähnlich definiert wie eine Klasse; der Klassen-Definition wird jedoch

```
template <class Type>
```

vorangestellt. In der eigentlichen Definition der Klasse kann beliebig oft der bei der Definition der Schablone verwendete Typenname, in unserem Fall *Type*, als Typenbezeichnung vorkommen. Bei der Definition eines *Objekts* der Klasse muss dann, ebenfalls in "`< >`", die tatsächliche Type angegeben werden, für die das Objekt angelegt werden soll. Dies kann eine beliebige fundamentale oder abgeleitete Type sein, also beispielsweise **<char>** oder <Point>.

Das folgende Beispiel zeigt die Verwendung einer Schablone für eine allgemeine Stack-Klasse: Elemente beliebigen Typs können in einen Stapelspeicher geschrieben und in der umgekehrten Eingabereihenfolge von dort wieder entnommen werden. Als allgemeiner Typenname wurde hier "T" verwendet.

Demo-Programm 5_07_01.cc

```
#include <iostream.h>

template <class T> class stack                  // allgemeine Klasse "stack"
{
    T *basis;
    T *zeiger;
    int gr;

public:
    stack (int groesse)                         // Konstruktor
        { basis = zeiger = new T [gr = groesse]; }

    ~stack ()                                   // Destruktor
        { delete [] basis; }

    void push (T t)                             // Objekt auf den Stack schieben
    {
        if (zeiger - basis < gr)                // Überlaufschutz
            *zeiger++ = t;
    }
```

```
    T pop (void)                                // Objekt vom Stack holen
    {
        if (zeiger > basis)                     // Überlaufschutz
            return *--zeiger;
        else
            return 0;
    }

    int frei (void)                             // freie Stack-Eintragungen
        { return gr - (zeiger - basis); }
};


int main ()
{
    int stackgr;

    cout << "Stack-Größe:  ";
    cin >> stackgr;

    stack <int> si (stackgr);           // Implementierung: Stack für ints
    stack <char> sc (stackgr);          // Implementierung: Stack für chars

    do                                  // int-Stack auffüllen
    {
        int i;

        cout << "Zahl eingeben: ";
        cin >> i;

        si.push (i);      // Elementfunktion mit int-Argument aufgerufen

        cout << "Noch frei: " << si.frei() << " int-Plätze\n";
    }
    while (si.frei () > 0);

    cout << "\n";

    do                                  // char-Stack auffüllen
    {
        char ch;

        cout << "Zeichen eingeben: ";
        cin >> ch;
        sc.push (ch);     // Elementfunktion mit char-Argument aufgerufen

        cout << "Noch frei: " << sc.frei() << " char-Plätze\n";
    }
    while (sc.frei () > 0);

    // Stacks ausräumen

    cout << "Stacks ausleeren:\n";

    for (int i = 0; i < stackgr; i++)
        cout << "int = " << si.pop() << "; char = " << sc.pop() << "\n";
}
```

Das Programm ermöglicht beispielsweise den folgenden Dialog:

```
Stack-Größe:  5
Zahl eingeben: 3
Noch frei: 4 int-Plätze
Zahl eingeben: 5
Noch frei: 3 int-Plätze
Zahl eingeben: 7
Noch frei: 2 int-Plätze
```

```
Zahl eingeben: 11
Noch frei: 1 int-Plätze
Zahl eingeben: 13
Noch frei: 0 int-Plätze

Zeichen eingeben: a
Noch frei: 4 char-Plätze
Zeichen eingeben: b
Noch frei: 3 char-Plätze
Zeichen eingeben: c
Noch frei: 2 char-Plätze
Zeichen eingeben: d
Noch frei: 1 char-Plätze
Zeichen eingeben: e
Noch frei: 0 char-Plätze
Stacks ausleeren:
int = 13; char = e
int = 11; char = d
int = 7; char = c
int = 5; char = b
int = 3; char = a
```

Bei der Definition der Klasse `stack` wird also festgelegt, dass überall dort, wo die Type `T` in der Definition der Klasse vorkommt, eine *beliebige* Type stehen kann (in unserem Beispiel dann **`int`** oder **`char`**). Die drei Datenelemente der Klasse (die **`private`** und daher von außen nicht zugänglich sind), sind je ein Zeiger auf die Basis des Stacks (`basis`) und auf das letzte benutzte Element (`zeiger`) sowie die maximale Größe des Stacks in Elementen (`gr`). Der Typ der Zeiger hängt davon ab, für welche Datentype der Stack angelegt wurde: für Stack-Objekte vom Typ **`int`** wäre es beispielsweise **`int *`**.

Der Konstruktor der Klasse `stack` allokiert Speicher, der für die als Argument übergebene Größe des Stacks ausreicht. Der vom Operator **`new`** zurückgegebene Zeiger wird sowohl dem Basiszeiger als auch dem Zeiger auf das oberste Element zugewiesen; ein neuer Stack ist leer. Dem Stack werden Objekte über die Funktion `push()` zugeführt; `push()` wird mit einem Argument aufgerufen, dessen Typ der gleiche sein muss wie die Klasse (oder der Typ), für den der Stack definiert wurde. (Hier ist also im Gegensatz zu einem Makro eine strikte Typenprüfung möglich.) Dann und nur dann, wenn auf dem Stack noch Platz ist, wird das Objekt, auf das `zeiger` zeigt, gleich dem Argument von `push()` gesetzt und `zeiger` anschließend inkrementiert. Umgekehrt wird das jeweils oberste Objekt mit `pop()` vom Stack geholt; das Ergebnis von `pop()` hat wieder die dem spezifischen Stack entsprechende Type. Sofern noch Daten auf dem Stack stehen (`zeiger > basis`), wird `zeiger` dekrementiert und dereferenziert; anderenfalls wird (in Ermangelung eines sinnvolleren Wertes) der Wert Null zurückgegeben. Die Funktion `frei()` ermittelt mit einer einfachen Zeigerarithmetik die Anzahl der Objekte auf dem Stack und subtrahiert sie von der Größe des gesamten Stacks. Beachten Sie bitte, dass auch in diesem Fall die Größe der Objekte implizit eingeht (weil die Differenz zweier Zeiger immer die Differenz ihrer Adressen, dividiert durch die Größe des Objekts, ist). Der Destruktor der Klasse verwendet den Operator **`delete[]`** (in der Form für allokierte Felder), um den allokierten Speicher wieder zurückzugeben.

In der Funktion `main()` wird zunächst die maximale Größe des Stacks interaktiv vorgegeben; anschließend werden zwei Stack-Objekte für **`int`**s und **`char`**s definiert:

```
    stack <int> si (stackgr);
    stack <char> sc (stackgr);
```

Diese beiden Stacks werden nacheinander interaktiv mit Zahlenwerten bzw. Zeichen aufgefüllt; wenn sie voll sind, werden sie in einer Schleife simultan entleert.

Die gleiche Funktionalität hätte auch *ohne* Verwendung von Schablonen erzielt werden können. In diesem Fall hätte jedoch für jede auf einem Stack aufzubewahrende Datentype eine *eigene* Klasse definiert werden müssen, also beispielsweise:

```
class int_stack
{
    int *basis;
    int *zeiger;
    int gr;
```

```
public:
    stack (int groesse) { basis = zeiger = new int [gr = groesse]; }
        // Konstruktor

    ~stack () { delete [] basis; }     // Destruktor

    void push (int t)
    {
        if (zeiger - basis < gr)
            *zeiger++ = t;
    }

    int pop (void)
    {
        if (zeiger > basis)
            return *--zeiger;
        else
            return 0;
    }

    int frei (void)
    {
        return gr - (zeiger - basis);
    }
};

class char_stack
{
...
};
```

Der Vorteil einer generalisierten Definition der Klasse liegt auf der Hand.

Die Definition der Klassenelement-Funktionen einer Schablone kann auch außerhalb der Definition der Schablone erfolgen:

Demo-Programm 5_07_02.cc

```
template <class T> class stack
{
    T *basis;
    T *zeiger;
    int gr;
public:
    stack (int);                        // Konstruktor
    ~stack ();                          // Destruktor
    void push (T);
    T pop (void);
    int frei (void);
};

template <class T> stack <T>::stack (int groesse)
    { basis = zeiger = new T [gr = groesse]; }

template <class T> stack <T>::~stack ()
    { delete [] basis; }

template <class T> void stack <T>::push (T t)
    { if (zeiger - basis < gr) *zeiger++ = t; }

template <class T> T stack <T>::pop (void)
{
    if (zeiger > basis)
        return *--zeiger;
    else
        return 0;
}

template <class T> int stack <T>::int frei (void)
    { return gr - (zeiger - basis); }
```

5.7.2. Funktions-Schablonen (*Function Templates*)

Auch außerhalb der Definition von Klassen kann der Mechanismus der Schablonen für die generalisierte Definition von Funktionen verwendet werden. Das folgende Beispiel zeigt die Definition einer Schablone für ein eindimensionales Feld beliebiger Größe und Type und eine Funktion, mit der die Elemente dieses Feldes (unter Verwendung des Sortier-Algorithmus von Seite 92) sortiert werden können. Im Gegensatz zu dem Programm von Seite 92 ist der Sortieralgorithmus in der hier präsentierten Form jedoch für (fast) alle Datentypen einsetzbar:

Demo-Programm 5_07_03.cc

```
#include <iostream.h>

template <class Type> class Feld
{
public:
    Feld (int groesse = 1) { f = new Type [gr = groesse]; }   // Konstruktor
    ~Feld () { delete[] f; }                                  // Destruktor

    int gr;                                   // Datenelemente: Größe und
    Type *f;                                  // Zeiger auf Feldelemente
};

// generalisierte Sortierfunktion:

template <class T> void sort (Feld<T>& F)
{
    unsigned n = F.gr;                        // Feldgröße

    for (int i = 0; i < n - 1; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            if (F.f[i] > F.f[j])
            {
                T temp = F.f[i];
                F.f[i] = F.f[j];
                F.f[j] = temp;
            }
        }
    }
}

// spezielle Ausgabefunktion:

void show (int zahl, Feld<char>& Fc, Feld<int>& Fi, Feld<double>& Fd)
{
    for (int i = 0; i < zahl; i++)
        cout << Fc.f[i] <<"\t" << Fi.f[i] << "\t" << Fd.f[i] << "\n";
}

int main ()
{
    int zahl;

    cout << "Feldgröße:       ";
    cin >> zahl;

    // "Felder" definieren:
    Feld <char> Fc (zahl);             // chars
    Feld <int> Fi (zahl);              // ints
    Feld <double> Fd (zahl);           // doubles
```

```
    for (int i = 0; i < zahl; i++)
    {
        cout << "Zeichen:         ";
        cin >> Fc.f[i];
        cout << "Ganze Zahl:      ";
        cin >> Fi.f[i];
        cout << "Gleitkommazahl:  ";
        cin >> Fd.f[i];
    }

    cout << "\nUnsortiert:\n";
    show (zahl, Fc, Fi, Fd);

    sort (Fc);                                  // sortieren
    sort (Fi);
    sort (Fd);

    cout << "\nSortiert:\n";
    show (zahl, Fc, Fi, Fd);
}
```

In diesem Demo-Programm wird zunächst eine Schablone `Feld` definiert, die als Datenelemente die Größe des Feldes (`gr`) sowie einen Zeiger auf den Anfang des Feldes (`f`) enthält. Der Konstruktor der Schablone allokiert Speicher für `groesse` Feldelemente und weist den von **`new`** übergebenen Zeiger dem Element `f` zu. Um den Konstruktor auch für Einzelelemente der Klasse des Feldes verwenden zu können, wird das Argument `groesse` auf `1` voreingestellt. (Diese Eigenschaft wird später in `sort()` benötigt, um ein temporäres Objekt (`temp`) zum Vertauschen der Feldelemente generieren zu können.) Der Destruktor der Klasse `Feld` ruft einfach den Operator **`delete[]`** auf.

Die Funktion `sort()` wird mit einer Referenz auf ein Objekt der Type `Feld` aufgerufen. Die dem Feld zugrundeliegende eigentliche Datentype wird durch die Type des Arguments festgelegt. `sort()` extrahiert die Feldgröße aus dem übergebenen Objekt und wendet auf die Elemente des Feldes `F.f` den Sortier-Algorithmus an. Ein Prototyp für `sort()` könnte lauten:

```
template <class T> void sort (Feld<T>& F);
```

oder

```
template <class T> void sort (Feld<T>&);
```

GNU-C/C++-spezifisch

(Wenn der Funktions-Prototyp *vor* der Definition der Schablone angegeben wird, erwartet der Compiler eine *externe* Funktions-Definition. Da diese nicht existiert, wird beim Zusammenbinden des Programms eine Fehlermeldung generiert.)

Die Funktion `show()` ist *keine* Schablone; im Gegensatz zu `sort()`, für das vom Compiler für jede Datentype, mit der `sort()` aufgerufen wird, separater Code angelegt wird, existiert `show()` genau einmal im übersetzten Programm. Die drei Implementierungen von `Feld`, mit dem `show()` aufgerufen wird, stellen vollständig definierte Klassen-Typen und *keine* Schablonen dar. Der Prototyp für `sort()` sieht folgendermaßen aus:

```
void show (int zahl, Feld<char>& Fc, Feld<int>& Fi, Feld<double>& Fd);
```

oder

```
void show (int, Feld<char>&, Feld<int>&, Feld<double>&);
```

In `main()` wird — ähnlich wie im Beispiel des vorigen Abschnitts — eine Feldgröße interaktiv eingegeben; entsprechend dieser Größe werden drei Objekte der Klasse `Feld` für **`char`**s, **`int`**s und **`double`**s erzeugt. Den Elementen dieser Felder werden anschließend Werte mit geeignetem Typ interaktiv zugewiesen; vor und nach dem (unabhängigen) Sortieren aller Felder wird ihr Inhalt mit `show()` ausgeschrieben. Diese beiden Ausgaben können beispielsweise so aussehen:

```
Unsortiert:
s       17      123.456
h       11      3.14159
i       23      321.098
f       13      1.4142
t       19      0.987654
```

```
Sortiert:
f       11      0.987654
h       13      1.4142
i       17      3.14159
s       19      123.456
t       23      321.098
```

Diese Definition von `sort()` ist grundsätzlich für alle Datentypen brauchbar, für die ein sinnvoller Vergleich

```
if (F.f[i] > F.f[j])
```

existiert. Für manche wichtige Datentypen (z.B. **char***) ist dies jedoch nicht der Fall; für den Vergleich der *Inhalte* (nicht der *Adressen*) von Strings ist die Bibliotheksfunktion `strcmp()` zu verwenden. In diesem Fall kann eine Vergleichsfunktion als Schablone definiert werden, die für alle Datentypen außer einigen speziell zu behandelnden verwendet wird; für spezielle Typen (z.B. **char***) wird nach den Regeln des *Function Overloading* eine spezielle Funktion mit gleichem Basis-Namen definiert:

Demo-Programm 5_07_04.cc

```
#include <iostream.h>
#include <string.h>

template <class T> class Vergleich      // generalisierte Klasse Vergleich
{
public:
    static int groesser (T& a, T& b)
        { return a > b; }
};

class Vergleich <char*>                 // spezielle Klasse für char*
{
public:
    static int groesser (char *a, char *b)
        { return strcmp (a, b) > 0; }
};

template <class Type> class Feld        // Definition der Klasse Feld
{
public:
    Feld (int groesse = 1) { f = new Type [gr = groesse]; }  // Konstruktor
    ~Feld (){ delete[] f; }                                  // Destruktor

    int gr;                             // Datenelement: Größe
    Type *f;                            // Zeiger auf Feldelemente
};

template <class T> void sort (Feld<T>& F)           // Sortierfunktion
{
    unsigned n = F.gr;                  // Feldgröße

    for (int i = 0; i < n - 1; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            if (Vergleich<T>::groesser(F.f[i], F.f[j]))
            {
                T temp = F.f[i];
                F.f[i] = F.f[j];
                F.f[j] = temp;
            }
        }
    }
}
```

```
// spezielle Ausgabefunktion

void show (int zahl, Feld<char*>& Fpc, Feld<int>& Fi, Feld<double>& Fd)
{
    for (int i = 0; i < zahl; i++)
        cout << Fi.f[i] << "\t" << Fd.f[i] << "\t" << Fpc.f[i] << "\n";
}

int main ()
{
    int zahl;
    cout << "Feldgröße:       ";
    cin >> zahl;
                                                // "Felder" definieren:
    Feld <char*> Fpc (zahl);                    // Feld von Strings
    Feld <int> Fi (zahl);                       // Feld von ints
    Feld <double> Fd (zahl);                    // Feld von doubles

    for (int i = 0; i < zahl; i++)
    {
        Fpc.f[i] = new char[80];                // Speicherplatz für String
        cout << "String:          ";
        cin >> Fpc.f[i];
        cout << "Ganze Zahl:      ";
        cin >> Fi.f[i];
        cout << "Gleitkommazahl:  ";
        cin >> Fd.f[i];
    }

    cout << "\nUnsortiert:\n";
    show (zahl, Fpc, Fi, Fd);

    sort (Fpc);                                 // sortieren
    sort (Fi);
    sort (Fd);

    cout << "\nSortiert:\n";
    show (zahl, Fpc, Fi, Fd);
}
```

Mit Ausnahme der Definition der Klasse `Vergleich` in der allgemeinen (Schablonen-) Form sowie in der speziellen Form für **`char*`** und der fett markierten Änderung in `sort()` ist dieses Programm sehr ähnlich zu dem vorangehenden Demo-Programm. Die Definitionen von `Vergleich`, `Feld` und `sort` könnten beispielsweise in eine Programmbibliothek aufgenommen werden. Es ist hier auch möglich, Strings zu sortieren; dementsprechend wurde in `main()` ein Objekt der Type `Feld<char*>` definiert und auch in `show()` eine entsprechende Änderung vorgenommen. Da ein `Feld<char*>` auf einen String *zeigt*, diesen aber nicht *enthält*, musste mit

```
Fpc.f[i] = new char[80];
```

explizit Speicherplatz für den String allokiert werden. (Dies wäre allenfalls auch durch Verwendung eines entsprechenden Konstruktors in `Feld` möglich gewesen, wurde einfachheitshalber aber hier unterlassen.) Die Ausgabe des Programms kann etwa so aussehen:

```
Unsortiert:
53      3.14159 the
47      1.4142  quick
61      23.45   brown
59      1.4141  fox
43      12.34   jumps

Sortiert:
43      1.4141  brown
47      1.4142  fox
53      3.14159 jumps
59      12.34   quick
61      23.45   the
```

5.8. Spezielle Funktionen in Klassen

5.8.1. Überblick

Verschiedene spezielle Funktionen in C++ können ausschließlich als Teil der Definition einer Klasse definiert werden. Sie bestimmen, wie Objekte der Klasse erzeugt, vernichtet, kopiert und in Elemente einer anderen Klasse konvertiert werden. Einige dieser Funktionen können auch implizit vom Compiler aufgerufen werden. Die speziellen Funktionen unterliegen ebenso wie gewöhnliche Klassenelement-Funktionen den Zugriffsregeln (siehe Seite 127).

Die folgende Tabelle fasst das Verhalten dieser Funktionen zusammen:

Funktion	vererbt von Basisklasse	virtuelle Funktion	Ergebnis mit `return`	Element oder `friend`	Default vom Compiler
Konstruktor	nein	nein	nein	Element	ja
Kopier-Konstruktor	nein	nein	nein	Element	ja
Destruktor	nein	ja	nein	Element	ja
Konversion	ja	ja	nein	Element	nein
Zuweisung	nein	ja	ja	Element	ja
`new`	ja	nein	**`void*`**	statisches Element	nein
`delete`	ja	nein	**`void`**	statisches Element	nein
Element-funktionen	ja	ja	ja	Element	nein
`friend`-Funktionen	nein	nein	ja	**`friend`**	nein

5.8.2. Konstruktoren

Klassenelement-Funktionen mit dem Namen der Klasse sind Konstruktoren. Konstruktoren haben keine Ergebnis-Type (nicht einmal **`void`**!) und dürfen auch keinen Wert als Resultat zurückgeben. Es ist auch unzulässig, einen Funktionszeiger auf einen Konstruktor zu definieren. Wenn eine Klasse einen Konstruktor besitzt, wird jedes ihrer Objekte zum Zeitpunkt seiner Erstellung unter Verwendung des Konstruktors initialisiert. Solche Objekte können sein:

- Globale Objekte (mit externer oder Datei-Gültigkeit);
- Lokale Objekte mit Gültigkeit innerhalb einer Funktion oder des sie umschließenden Blocks;
- Dynamische Objekte, die mit **`new`** vom *Free Store* allokiert wurden;
- Temporäre Objekte, die implizit vom Compiler generiert werden;
- Objekte, die Elemente einer anderen Klasse sind;
- Jener Teil des Objektes einer abgeleiteten Klasse, der zur Basisklasse gehört und bei der Erstellung eines Objektes der abgeleiteten Klasse mit erstellt wird.

Entsprechend den Regeln für *Function Overloading* können beliebig viele Konstruktoren in einer Klasse definiert werden.

In C++ existieren zwei spezielle Typen von Konstruktoren:

- Default-Konstruktor: Er hat kein Argument oder kann ohne Argument aufgerufen werden (bei Angabe von voreingestellten Argumenten). Er wird implizit vom Compiler verwendet, um ein Standard-Objekt der Klasse zu erzeugen.
- Kopier-Konstruktor: Er akzeptiert ein einziges Argument, dessen Type eine Referenz auf die Klasse sein muss. Er wird vom Compiler implizit verwendet, um bei der Erstellung eines neuen Objektes den Inhalt eines bestehenden Objektes zu duplizieren (siehe Seite 62 und 148).

Der Compiler kann einen Default- und einen Kopier-Konstruktor generieren, wenn keiner spezifiziert wurde. Solche Konstruktoren initialisieren die intern zur Behandlung virtueller Basisklassen und Funktionen benötigten Tabellen und rufen die Konstruktoren von Basisklassen und/oder Element-Klassen auf, wenn solche existieren. Der compilergenerierte Kopier-Konstruktor kopiert zusätzlich die Elemente des Objekts; wenn keine Konstruktoren von Basis- oder Elementklassen existieren, werden deren Elemente bitweise kopiert.

Konstruktoren können verwendet werden, um *Objekte* zu initialisieren, die als **`const`** oder **`volatile`** deklariert wurden. Die *Konstruktoren* selbst können jedoch *nicht* als **`const`**, **`volatile`**, **`virtual`** oder **`static`** deklariert werden.

Konstruktoren können explizit aufgerufen werden, beispielsweise in Funktionsaufrufen:

```
DrawLine (Point(17, 23), Point(35, 46));
```

oder bei Initialisierungen:

```
Point pt = Point(39, 46);
```

In einem solchen Fall existiert das Objekt, das durch den Konstruktor erstellt wird, nur für die Dauer der Auswertung des jeweiligen Ausdrucks.

Konstruktoren können beliebige Klassenelement-Funktionen aufrufen; potentielle Probleme können jedoch beim Aufruf einer virtuellen Klassenelement-Funktion einer abstrakten Basisklasse entstehen.

Felder werden elementweise in aufsteigender Reihenfolge unter Verwendung des Default-Konstruktors erstellt.

Objekte abgeleiteter Klassen werden durch Aufruf der Konstruktoren, beginnend mit der untersten Basisklasse, erstellt: Vom Compiler wird unmittelbar am Anfang des Konstruktors einer abgeleiteten Klasse ein Aufruf des/der Konstruktor(en) seiner direkten Basisklasse(n) eingefügt, sodass *de facto* der Konstruktor der untersten Basisklasse *vor* den Konstruktoren der abgeleiteten Klassen ausgeführt wird. Damit ist sichergestellt, dass beim Aufruf jedes Konstruktors alle seine Basisklassen vollständig aufgebaut sind. Analog werden die Konstruktoren der Klassen der *Elemente* einer Klasse aufgerufen, bevor der Konstruktor jener Klasse ausgeführt wird, in der sie enthalten sind.

5.8.3. Destruktoren

Destruktoren zerstören Klassenobjekte, die nicht mehr benötigt werden. Der Name des Destruktors ist immer der der Klasse mit einer vorangestellten Tilde ("**~**"). Für Destruktoren gelten die folgenden Regeln:

- Sie haben keine Funktionsargumente.
- Sie haben kein Ergebnis und keine Ergebnistype (nicht einmal **`void`**!).
- Sie dürfen nicht als **`const`**, **`volatile`** oder **`static`** deklariert werden (aber sehr wohl auf Objekte angewendet werden, die als **`const`**, **`volatile`** oder **`static`** deklariert wurden).
- Sie können als virtuelle (und sogar als reine virtuelle Funktionen in einer abstrakten Klasse) deklariert werden. In diesem Fall können Objekte zerstört werden, deren Type nicht bekannt ist; der korrekte Destruktor wird aufgrund des virtuellen Aufrufmechanismus automatisch gewählt.
- Es ist unzulässig, einen Funktionszeiger auf einen Destruktor zu definieren.
- Abgeleitete Klassen erben nie den Destruktor ihrer Basisklasse. Wenn für eine abgeleitete Klasse kein Destruktor existiert, generiert der Compiler einen solchen.

Destruktoren werden in den folgenden Fällen ausgeführt:

- Wenn ein mit **new** allokiertes Objekt mit **delete** zerstört wird. In diesem Fall wird immer das "am meisten abgeleitete" Objekt zerstört, wofür allerdings nur dann garantiert werden kann, wenn virtuelle Destruktoren verwendet werden.
- Wenn ein lokales Objekt mit Blockgültigkeit seine Gültigkeit am Ende des Blocks verliert.
- Wenn die Lebensdauer eines temporären Objekts endet.
- Im Falle globaler oder statischer Objekte, wenn das Programm beendet wird.
- Wenn der Destruktor explizit aufgerufen wird.

Wenn eine Basisklasse oder eine Element-Klasse einen zugänglichen Destruktor hat, generiert der Compiler für alle abgeleiteten Klassen einen Default-Destruktor, sofern für diese Klassen kein Destruktor definiert wurde. Dieser compilergenerierte Destruktor ist **public**; er ruft die Destruktoren der Basisklassen und der Element-Klassen auf.

Destruktoren können beliebige Klassenelement-Funktionen aufrufen.

Bei der Ausführung eines Destruktors werden — soweit zutreffend — die folgenden Schritte durchlaufen:

- Der Destruktor der Klasse wird aufgerufen und komplett ausgeführt.
- Wenn die Klasse Elemente enthält, die Objekte anderer Klassen sind, werden für diese Objekte die Destruktoren ihrer Klassen in der umgekehrten Reihenfolge ihrer Deklaration aufgerufen.
- Die Destruktoren nicht-virtueller Basisklassen werden in der umgekehrten Reihenfolge ihrer Deklaration aufgerufen.
- Die Destruktoren virtueller Basisklassen werden in der umgekehrten Reihenfolge ihrer Deklaration aufgerufen.

Ein expliziter Aufruf eines Destruktors kann wie folgt erfolgen:

```
class Point;

Point pt;
Point *ppt = &pt;
...

// nicht-virtueller Aufruf:
pt.Point::~Point();              // oder
ppt->Point::~Point();

// virtueller Aufruf:
pt.~Point();                     // oder
ppt->~Point();
```

5.8.4. Temporäre Objekte

Temporäre Objekte werden automatisch vom Compiler in den folgenden Fällen erstellt:

- Initialisierung einer (als **const** betrachteten) Referenz mit einer Type, die sich von der des zugrundeliegenden Objekts unterscheidet (siehe Seite 65).
- Zur Aufnahme des mit **return** zurückgegebenen Ergebnisses einer Funktion, wenn die beiden folgenden Bedingungen erfüllt sind:
  - Die Funktion hat als Resultat ein Objekt einer benutzerdefinierten Type (typisch **class** oder **struct**), und
  - das Ergebnis der Funktion wird *nicht* auf ein anderes Objekt kopiert.

  Diese Bedingungen sind dann erfüllt, wenn entweder eine Funktion mit einem Ergebnis-Typ ungleich **void** aufgerufen wird, ohne dass ihr Ergebnis zugewiesen wird, oder wenn Funktionen Teile von Ausdrücken sind und ihr Ergebnis (mit einer benutzerdefinierten Implementierung eines Operators) als Operand in dem Ausdruck weiterverwendet werden soll. In jedem dieser Fälle muss ein temporäres Objekt erstellt werden, auf das die Funktion ihr Ergebnis abspeichern kann.

- Für das Ergebnis einer Typenumwandlung (*Cast*) in eine benutzerdefinierte Type.

Diese temporären Objekte werden automatisch wieder zerstört, sobald sie nicht mehr benötigt werden. Dies ist bei Objekten, die das Zwischenergebnis eines Ausdruckes aufnehmen sollen, am Ende des Ausdrucks (d.h., bei Erreichen des Strichpunkts oder der schließenden Klammer jenes Ausdrucks, der bei bedingter Programmausführung die Bedingung darstellt). Temporäre Objekte, die als einer der Operanden einer logischen Verknüpfung (mit "**||**" oder "**&&**") erstellt wurden, werden zerstört, sobald der rechte Operand dieser Verknüpfung ausgewertet wurde. Temporäre Objekte, die für die Definition einer Referenz benötigt wurden, werden unmittelbar nach der Referenz zerstört. Wenn in einem Ausdruck mehrere temporäre Objekte erzeugt wurden, werden sie in der umgekehrten Reihenfolge ihrer Erstellung wieder zerstört.

5.8.5. Konversionen

Die Konversion eines Klassen-Objekts in ein Objekt einer anderen Klasse kann erfolgen durch:

- Erstellen eines neuen Objekts der Ziel-Klasse und Kopieren des Inhalts des ursprünglichen Objekts auf das Zielobjekt — Konversion mittels Konstruktor.
- Umsetzung mittels einer benutzerdefinierten Konversionsfunktion.

Konversionen erfolgen in den folgenden Fällen:

- Explizite Konversion (*Type Cast* oder Funktions-Stil);
- Initialisierung eines Objekts mit einem Ausdruck einer anderen als seiner eigenen Type;
- Aufruf einer Funktion mit einem aktuellen Argument, dessen Type von der des formalen Arguments in der Deklaration der Funktion abweicht;
- Zuweisung des Ergebnisses einer Funktion auf ein Objekt einer anderen Type;
- Ausdruck mit Objekten unterschiedlicher Typen;
- Die Type eines Ausdrucks ist von der erwarteten verschieden (z.B. **switch**-Befehle).

Benutzerdefinierte Konversionen werden nur dann verwendet, wenn sie eindeutig sind; ansonsten erfolgt eine Fehlermeldung.

5.8.5.1. Konversions-Konstruktoren

Konstruktoren mit einem einzigen Argument können für die Umwandlung in eine benutzerdefinierte Klassen-Type verwendet werden, sofern ein Konstruktor mit einer korrekten Type seines Arguments existiert. (Standardumwandlungen, z.B. zwischen **int** und **double**, werden jedenfalls automatisch durchgeführt.) Das folgende Beispiel zeigt eine (zulässige) Konversion von **int** in `class A` und von dort in `class B`:

Demo-Programm 5_08_01.cc

```
#include <iostream.h>

class A
{
public:
    A (int i = 0) {_n = i; }           // Konstruktor mit int-Argument
    operator int() { return _n; }      // Konversions-Funktion A -> int
private:
    int _n;
};

class B
{
public:
    B (A a) { _a = a; }          // Konstruktor mit Argument der Klasse A
    show () { cout << int (_a) << "\n"; } // Ausgabe mit Konversion A -> int
private:
    A _a;
};
```

```
int main ()
{
    B b = 17;                    // Initialisierung von B mit int-Wert
    b.show();
}
```

In diesem Demo-Programm laufen die folgenden Vorgänge ab:

- Der Konstruktor für A wird mit dem Wert 17 als Argument aufgerufen (Konversion von **int** in die Type des Arguments des Konstruktors von B).
- Der Konstruktor für B wird mit dem zuvor erstellten temporären Objekt der Klasse A als Argument aufgerufen.
- Der Konstruktor von B ruft den Konstruktor von A als Default-Konstruktor (mit dem voreingestellten Argument 0), um das Objekt _a der Klasse A in Klasse B zu initialisieren.
- Der Inhalt des Arguments für Konstruktor B (17) wird in das Objekt _a kopiert.
- Beim Aufruf von show() wird die Konversions-Funktion **operator int()** mit der Adresse von b als (implizites) Argument aufgerufen; diese Funktion gibt den Wert b._a als (**int**-) Ergebnis zurück.
- Dieser Wert wird an die Ausgabefunktion der Klasse ostream übergeben und ausgeschrieben.

Das Demo-Programm funktioniert jedoch nur dann korrekt, wenn alle folgenden Bedingungen erfüllt sind:

- Es muss ein Default-Konstruktor für die Klasse A existieren, damit das Objekt _a in Klasse B initialisiert werden kann. Der einfachste Weg, einen Default-Konstruktor für A zu erstellen, ist die Verwendung eines voreingestellten Wertes für das Argument des Konstruktors A.
- Für die Ausgabe wird eine Konversionsfunktion **operator int()** benötigt, die das Objekt b._a mit der Type A (für die keine Standard-Ausgabefunktion existiert) in die Type **int** umwandelt. Eine "gewöhnliche" Typenumwandlung mit (int)_a oder int(_a) *ohne* Definition eines *overloaded* Konversions-Operators funktioniert nicht! (In B::show() ist es übrigens gleichgültig, ob die Typkonversion von A in **int** mittels eines funktionsartigen Aufrufs (int(_a), wie im Demo-Programm) erfolgt, oder ob ein *Type Cast* ((int)_a) verwendet wird.)

5.8.5.2. Konversionsfunktionen

Konversionsfunktionen erlauben die explizite Typenumwandlung durch *Type Casts* oder funktionsartigen Aufruf. Sie werden als Klassenelement-Funktionen mit dem Namen "**operator** *type***()**" definiert; sie haben keine Argumente und ein Resultat mit der durch *type* festgelegten Type. Ein Beispiel für eine derartige Konversionsfunktion war die Funktion A::operator int() im Beispiel des vorigen Abschnitts.

Für Konversionsfunktionen gelten die folgenden Regeln:

- Klassen, Aufzählungen und **typedef**s dürfen nicht innerhalb der Definition einer Konversionsfunktion definiert werden; ihre Definition muss separat zuvor erfolgen.
- Konversionsfunktionen haben keine Argumente.
- Die Type des Ergebnisses einer Konversionsfunktion ist durch ihren Namen festgelegt. Die Angabe einer Ergebnistype ("double operator int()" oder selbst "int operator int()") ist jedenfalls unzulässig.
- Konversionsfunktionen dürfen als virtuell deklariert werden.

Konversionsfunktionen (und *Operator Overloading*; siehe Seite 158) können zu Mehrdeutigkeiten führen, wie im folgenden Beispiel illustriert wird:

Demo-Programm 5_08_02.cc

```
#include <string.h>

class String
{
public:
    String (char *s) { strcpy (str, s); }
                                 // Konversions-Konstruktor char* -> String
```

```
    operator char*() { return str; }
                                  // Konversions-Funktion String -> char *
    int operator == (const String s) { return ! strcmp (str, s.str); }
                                  // benutzerdefinierter Gleichheits-Operator
private:
    char str[80];
};

int main ()
{
    String s ("abcde");
    char *pc = "efgh";
    int i;

    i = (s == pc);          // mehrdeutig (aber in GNU C++ zulässig)
    i = ((char*) s == pc);  // String -> char*, ADRESSEN werden verglichen
    i = (s == String (pc)); // char* -> String, benutzerdefinierter Vergleich
}
```

GNU-C/C++-spezifisch

Die Mehrdeutigkeit in diesem Beispiel kommt durch den Ausdruck `i = (s == pc);` zustande. Dieser kann folgendermaßen interpretiert werden:

- Verwende den benutzerdefinierten Operator **`operator char*()`**, um das Objekt der Type `String` in ein Objekt der Type **`char*`** umzuwandeln, und vergleiche die beiden Zeiger mit der Standard-Vergleichsoperation von C/C++. Diese Vorgangsweise entspricht:

  ```
  i = ((char*) s == pc);
  ```

  Diese Operation vergleicht die *Adressen*, nicht die *Inhalte* der beiden Strings!

- Verwende den Konstruktor der Klasse `String`, um ein Objekt der Type **`char*`** in ein Objekt der Klasse `String` umzuwandeln, und vergleiche die beiden Strings unter Verwendung des benutzerdefinierten Vergleichs-Operators **`operator==()`**. Diese Vorgangsweise entspricht:

  ```
  i = (s == String (pc));
  ```

  In diesem Fall werden die *Inhalte* der beiden Strings (mittels der Bibliotheksfunktion `strcmp()`) verglichen.

GNU-C/C++-spezifisch

Eigentlich sollte aufgrund dieser Mehrdeutigkeit der Compiler eine Fehlermeldung abgeben und die Übersetzung abbrechen; der GNU C++-Compiler wählt aber (ohne Warnung) die letztere (und wahrscheinlich sinnvollere) der beiden Möglichkeiten.

5.8.6. Kopierfunktionen

Kopiervorgänge erfolgen in zwei Fällen: Initialisierung und Zuweisung. Die Semantik eines Kopiervorganges hängt von der Type der beteiligten Objekte ab; es ist daher notwendig, die Möglichkeit vorzusehen, die Kopierfunktion entsprechend zu definieren. Für eine C++-Klasse können beide Kopierfunktionen unabhängig festgelegt werden:

- Initialisierungen durch einen benutzerdefinierten Kopier-Konstruktor;
- Zuweisungen durch eine benutzerdefinierte **`operator=()`**-Funktion.

Kopier-Konstruktoren und Zuweisungs-Funktionen haben ein einziges Argument der Type "Referenz auf die Klasse"; sie sollte zweckmäßigerweise als

`const` *`Klassenname`***`&`**

definiert werden, um auch Initialisierungen unter Verwendung von konstanten Objekten zu erlauben. Wenn kein Kopier-Konstruktor bzw. keine Zuweisungs-Funktion definiert wurde, generiert der Compiler eine solche. Diese automatisch erstellten Funktionen haben ein Argument der Type "*`Klassenname`***`&`**" (ohne "**`const`**"), außer, alle Basis- und Elementklassen haben Kopier-Konstruktoren bzw. Zuweisungs-Funktionen mit einem Argument der Type "**`const`** *`Klassenname`***`&`**". In diesem Fall ist auch das Argument der compilergenerierten Funktion vom Typ "**`const`** *`Klassenname`***`&`**".

Zuweisungsfunktionen haben die Deklaration

*`type`***`&`** *`type`***`::operator=(const`** *`type`***`&)`**.

Die Deklaration der Resultattype kann auch unterbleiben.

Das folgende Beispiel illustriert die Eigenschaften von Kopier-Konstruktor und benutzerdefinierter Zuweisungsfunktion:

Demo-Programm 5_08_03.cc

```
#include <iostream.h>

class Point
{
public:
    Point (int x = 0, int y = 0) { _x = x; _y = y; }
                                        // Standard- und Default-Konstruktor
    Point (const Point& pt) { _x = 3*pt._x; _y = 3*pt._y; }
                                        // Kopier-Konstruktor
    Point& operator=(const Point& pt)   // Zuweisungs-Operator
    {
        _x = 5*pt._x;
        _y = 5*pt._y;
        return *this;
    }

    show (){ cout << "x = " << _x << ", y = " << _y << "\n"; }

private:
    short _x, _y;
};

int main ()
{
    Point pt1 (10, 20);                 // Standard-Konstruktor
    Point pt2 = pt1;                    // Kopier-Konstruktor
    Point pt3;                          // Default-Konstruktor

    pt1.show();
    pt2.show();
    pt3.show();

    pt3 = pt1;                          // benutzerdefinierte Zuweisung
    pt3.show();
}
```

Zur Verdeutlichung der Mechanismen von Kopier-Konstruktor und benutzerdefinierter Zuweisungsfunktion wurden die Punkt-Koordinaten mit 3 bzw. 5 multipliziert. Das Programm hat die Ausgabe:

```
x = 10, y = 20
x = 30, y = 60
x = 0, y = 0
x = 50, y = 100
```

Wenn der Compiler einen Kopier-Konstruktor bzw. eine Zuweisungsfunktion generiert, erfolgt die Initialisierung bzw. Zuweisung durch elementweises Kopieren des Quellobjekts auf das Zielobjekt.

Das vorige Programm wäre auch ohne die Definitionen von

Demo-Programm 5_08_04.cc

`Point (const Point& pt)` und
`Point& operator=(const Point& pt)`

fehlerfrei übersetzbar und lauffähig; seine Ausgabe wäre in diesem Fall (wie üblicherweise erwartet):

```
x = 10, y = 20
x = 10, y = 20
x = 0, y = 0
x = 10, y = 20
```

Das folgende Programm demonstriert die Initialisierung und Zuweisung zwischen Objekten einer Basisklasse und einer abgeleiteten Klasse unter Verwendung des compilergenerierten Kopier-Konstruktors und der Standard-Zuweisungsfunktion:

Demo-Programm 5_08_05.cc

```
#include <iostream.h>

class Basis
{
public:
    Basis (int i = 0, int j = 0) { _i = i; _j = j; }
    Show () { cout << "Basis: i = " << _i << ", j = " << _j << "\n"; }
protected:
    int _i, _j;
};

class Abgeleitet : public Basis
{
public:
    Abgeleitet (int i=0, int j=0, int k=0) { _i = i; _j = j; _k = k; }
    Show () { cout << "Abgeleitet: i = " << _i << ", j = " << _j
        << ", k = " << _k << "\n"; }
protected:
    int _k;
};

int main ()
{
    Basis b1 (3, 5);
    Abgeleitet a1 (7, 11, 13);
    Basis b2 = a1;                      // Initialisierung
    Abgeleitet a2 = b1;                 // unzulässig
    Basis b3;                           // (0, 0)
    Abgeleitet a3;                      // (0, 0, 0)

    b1.Show();
    a1.Show();
    b2.Show();
    a2.Show();
    b3.Show();
    a3.Show();

    b3 = a1;                            // Zuweisung
    a3 = b1;                            // unzulässig

    b3.Show();
    a3.Show();
}
```

Für Initialisierungen und Zuweisungen zwischen Objekten einer Basisklasse und einer abgeleiteten Klasse gelten die folgenden Regeln:

- Objekte einer Basisklasse können mit einem Objekt einer von ihr abgeleiteten Klasse initialisiert werden bzw. ihnen kann ein Objekt einer abgeleiteten Klasse zugewiesen werden.
- Initialisierungen und Zuweisungen in umgekehrter Richtung sind nicht zulässig: Ein Objekt einer abgeleiteten Klasse kann weder mit einem Objekt seiner Basisklasse initialisiert werden, noch kann ihm dessen Wert zugewiesen werden.

Das Demo-Programm erzeugt daher die Ausgabe:

```
Basis: i = 3, j = 5
Abgeleitet: i = 7, j = 11, k = 13
Basis: i = 7, j = 11
Basis: i = 0, j = 0
Abgeleitet: i = 0, j = 0, k = 0
Basis: i = 7, j = 11
Abgeleitet: i = 0, j = 0, k = 0
```

5.8.7. Spezielle Initialisierungen

5.8.7.1. Felder

Felder von Klassen-Objekten werden — soweit vorhanden — unter Verwendung des Konstruktors der Klasse initialisiert. Wenn eine Liste von Initialisierungswerten gegeben wird, die weniger Elemente enthält, als zur Initialisierung des gesamten Feldes notwendig wären, werden die ersten Feldelemente mit den Werten aus der Liste initialisiert, alle weiteren unter Verwendung des Default-Konstruktors:

Demo-Programm 5_08_06.cc

```
#include <iostream.h>

class Punkt
{
public:
    Punkt (short x = 3, short y = 5) { _x = x; _y = y; }
    Zeige () { cout << "x = " << _x << ", y = " << _y << "\n"; }
private:
    short _x, _y;
};

int main ()
{
    Punkt pt[5] = {
        Punkt (11, 13),                 // pt[0]
        Punkt (17, 19),                 // pt[1]
        Punkt (23, 29) };               // pt[2]

    for (int i = 0; i < 5; i++)
        pt[i].Zeige();
}
```

In diesem Beispiel werden die ersten drei Elemente des Feldes `pt` explizit aus der Liste von Initialisierungswerten initialisiert; die letzten beiden werden unter Verwendung des Default-Konstruktors "gebaut", der durch Voreinstellung der Argumente des Standard-Konstruktors erhalten wird. (Zur besseren Illustration dieses Verhaltens wurden die Default-Initialisierungswerte auf `(3, 5)` gesetzt.) Die Ausgabe des Programms ist daher:

```
x = 11, y = 13
x = 17, y = 19
x = 23, y = 29
x = 3, y = 5
x = 3, y = 5
```

Statische Klassenelement-Felder können wie "gewöhnliche" Felder bei ihrer Definition außerhalb der Klassen-Definition initialisiert werden.

5.8.7.2. Klassen-Objekte als Klassenelemente

Klassen können Elemente enthalten, die ihrerseits Objekte einer anderen Klasse sind. Diese Objekte können aber nur unter einer der folgenden Voraussetzungen initialisiert werden:

- Die Klasse des eingeschlossenen Objekts benötigt keinen Konstruktor.
- Die Klasse des eingeschlossenen Objekts hat einen zugänglichen Default-Konstruktor.
- Alle Konstruktoren der umschließenden Klasse initialisieren das eingeschlossene Objekt explizit.

Im folgenden Beispiel enthält die Klasse `Rect` zwei Objekte der Klasse `Point`:

Demo-Programm 5_08_07.cc

```
#include <iostream.h>

class Point
{
public:
    Point (short x = 0, short y = 0) { _x = x; _y = y; }
        // Standard- und Default-Konstruktor
    Show () { cout << "x = " << _x << ", y = " << _y << "\n"; }
private:
    short _x, _y;
};

class Rect
{
public:
    Rect (short x1, short y1, short x2, short y2);   // Deklaration
    Show ()
    {
        cout << "Punkt 1: ";
        _pt1.Show();
        cout << "Punkt 2: ";
        _pt2.Show();
    }

private:
    Point _pt1, _pt2;                   // Elemente von Type Point
};

// Definition des Konstruktors von Rect:
Rect::Rect (short x1, short y1, short x2, short y2) :
    _pt1 (x1, y1), _pt2 (x2, y2) { }     // LEERE Funktionsdefinition!

int main ()
{
    Rect r (1, 2, 3, 4);
    r.Show();
}
```

Beachten Sie bitte die Syntax der Definition des Konstruktors von `Rect` (insbesondere den Doppelpunkt!) Der Konstruktor von `Rect` ruft zweimal den Konstruktor von `Point` auf, um die Objekte `_pt1` bzw. `_pt2` zu initialisieren. Das Programm hat die Ausgabe:

```
Punkt 1: x = 1, y = 2
Punkt 2: x = 3, y = 4
```

5.8.7.3. Initialisierung von Basisklassen

Basisklassen werden im wesentlichen ebenso initialisiert wie Objekte von Klassen als Klassenelemente:

```
class Window                            // Basisklasse
{
public:
    Window (Rect rSize);                // Konstruktor
    ...
};

class DialogBox : public Window         // abgeleitete Klasse
{
public:
    DialogBox (Rect rSize);             // Konstruktor
    ...
};
```

```
// Definition des Konstruktors von DialogBox

DialogBox::DialogBox (Rect rSize) : Window (rSize) { }
```

Bei der Erstellung eines Objekts der Type `DialogBox` wird hier der Konstruktor der Klasse `Window` mit dem Argument `rSize` aufgerufen. Im obigen Beispiel erfolgt keine Initialisierung der Klassenelemente von `DialogBox` (wohl aber eine der Elemente von `Window`).

Die Elemente einer virtuellen Basisklasse müssen entweder explizit vom Konstruktor der "am meisten abgeleiteten" Klasse initialisiert werden, oder die virtuelle Basisklasse muss einen Default-Konstruktor haben. Die Initialisierung der der virtuellen Basisklasse zugehörigen Objekte innerhalb einer anderen abgeleiteten Klasse als der "am meisten abgeleiteten" wird ignoriert.

Das folgende Demo-Programm basiert in wesentlichen Teilen auf dem "Bibliotheksverwaltungsprogramm" von Seite 114 bzw. 117; es verwaltet ebenso wie dieses eine *heterogene Liste* von Klassen-Objekten, also ein Feld von Zeigern auf Klassen-Objekte. Das Programm definiert eine Basisklasse `Dokument` und drei davon abgeleitete Klassen `Buch`, `Manual` und `Datei`, die nur deswegen benötigt werden, weil für Objekte der abgeleiteten Klassen unterschiedliche (Ausgabe-) Klassenelement-Funktionen gewünscht werden. Datenelemente werden jedoch auch bei Objekten einer der abgeleiteten Klassen ausschließlich in dem der Basisklasse entsprechenden Teil des Objekts — in zwei **`char`**-Feldern und einer **`int`**-Variable — abgespeichert. Damit ist es auch zweckmäßig, die Argumente des Konstruktors einer abgeleiteten Klasse an den Konstruktor der Basisklasse "weiterzureichen":

Demo-Programm 5_08_08.cc

```
#include <iostream.h>
#include <string.h>                             // für strcpy() und strncpy()

class Dokument
{
public:
    Dokument() { anzahl++; }                    // Default-Konstruktor

    Dokument(const char *s1, const char *s2, int i)
    {                                           // Standard-Konstruktor
        strncpy (_s1, s1, sizeof (_s1) - 1);
        _s1 [sizeof (_s1) - 1] = 0;
        strncpy (_s2, s2, sizeof (_s2) - 1);
        _s2 [sizeof (_s2) - 1] = 0;
        _i = i;
        anzahl++;
    }

    virtual ~Dokument() { anzahl--; }           // Destruktor

    virtual void zeige() { }

    static int anzahl;                  // Anzahl der definierten Objekte

protected:
    char _s1[50], _s2[30];
    int _i;
};

int Dokument::anzahl = 0;               // Definiere anzahl

class Buch : public Dokument
{
public:
    Buch (const char *Titel, const char *Autor, int Seiten) :
        Dokument (Titel, Autor, Seiten) { }

    void zeige ()
    {
        cout << "Autor:  " << _s2 << ", Titel: " << _s1 << "\n\t"
            << _i << " Seiten\n";
    }
};
```

```
class Manual : public Dokument
{
public:
    Manual (const char *Titel, int Seiten) : Dokument (Titel, "", Seiten) { }

    void zeige ()
        { cout << "Manual: " << _s1 << " - " << _i << " Seiten\n"; }
};

class Datei : public Dokument
{
public:
    Datei (const char *Inhalt, const char *Pfad, int Bytes) :
        Dokument (Inhalt, Pfad, Bytes) { }

    void zeige ()
    {
        cout << "Datei:  " << _s1 << ", Pfad: " << _s2 << "\n\t"
            << _i << " Bytes\n";
    }
};

const int max_Eintr = 6;

Dokument *doc[max_Eintr];              // Feld von Zeigern

int main ()
{
    doc[0] = new Buch ("The C++ Programming Language", "Bjarne Stroustrup",
        669);
    doc[1] = new Buch ("C - The Complete Reference", "Herbert Schildt", 773);
    doc[2] = new Manual ("Microsoft Visual C++ - C++ Language Reference",
        468);
    doc[3] = new Manual ("Microsoft Quick C Run Time Library Reference", 687);
    doc[4] = new Datei ("GNU C++ Compiler-Dokumentation",
        "c:\\gnu\\docs\\gcc\\gcc.tex", 173664);
            // ACHTUNG: zwei "\" sind notwendig, um einen "\" darzustellen!
    doc[5] = new Datei ("GNU C Run Time Library Reference",
        "c:\\gnu\\docs\\djgpp\\libcref.i", 226882);

    cout << Dokument::anzahl << " Eintragungen:\n";

    for (int i = 0; i < Dokument::anzahl; i++)
        doc[i]->zeige();                    // Liste ausschreiben
}
```

Beachten Sie bitte, dass die statische Klassenelement-Variable `Dokument::anzahl` bei jedem Aufruf des Konstruktors von `Dokument` inkrementiert wird, sodass sie die korrekte aktuelle Anzahl von Datenelementen enthält. Das Demo-Programm gibt eine (sehr unvollständige) Liste der diesen Unterlagen zugrundeliegenden Quellen unter Verwendung der jeweils klassenspezifischen Ausgabefunktion aus:

```
6 Eintragungen:
Autor:  Bjarne Stroustrup, Titel: The C++ Programming Language
        669 Seiten
Autor:  Herbert Schildt, Titel: C - The Complete Reference
        773 Seiten
Manual: Microsoft Visual C++ - C++ Language Reference - 468 Seiten
Manual: Microsoft Quick C Run Time Library Reference - 687 Seiten
Datei:  GNU C++ Compiler-Dokumentation, Pfad: c:\gnu\docs\gcc\gcc.tex
        173664 Bytes
Datei:  GNU C Run Time Library Reference, Pfad: c:\gnu\docs\djgpp\libcref.i
        226882 Bytes
```

5.9. *Overloading*

5.9.1. *Function Overloading*

Unter dem Begriff "*Function Overloading*" versteht man die Möglichkeit in C++, Funktionen mit identischen Namen, aber mit unterschiedlicher Zahl und Type der formalen Argumente zu definieren (siehe Seite 73). Der Compiler wählt zum Zeitpunkt der Übersetzung die passende Funktion aufgrund der aktuellen Argumente nach folgenden Regeln (mit abnehmender Präzedenz):

- Eine exakte Übereinstimmung besteht.
- Eine triviale Konversion ist erforderlich:
  - Objekt in Referenz auf Objekt oder umgekehrt;
  - eindimensionales Feld in Zeiger;
  - Funktionsaufruf in Aufruf eines Funktionszeigers;
  - Objekt oder Zeiger in **`const`** oder **`volatile`** Objekt oder Zeiger.
- Eine Konversion zwischen ganzzahligen Typen (*Integral Promotion*) ist erforderlich.
- Eine Standard-Konversion existiert.
- Eine benutzerdefinierte Konversion existiert.
- Eine variable Anzahl von Funktionsargumenten ("**`...`**") war deklariert.

Funktionen mit *n* voreingestellten Argumenten werden als Satz von *n*+1 individuell verschiedenen Funktionen betrachtet. Nicht-statische Klassenelement-Funktionen haben einen impliziten **`this`**-Zeiger; die Type dieses **`this`**-Zeigers muss exakt mit der des entsprechenden Arguments übereinstimmen.

Der Compiler prüft für jeden Funktionsaufruf alle vorhandenen *overloaded* Funktionen und wählt zunächst einmal alle jene aus, für die für *jedes* Argument *irgendeine* der beschriebenen Übereinstimmungsregeln erfüllt ist. Danach werden die erforderlichen Konversionen nach ihrer Präzedenz gewichtet und jene Funktion gewählt, für die der Konversionsaufwand am geringsten ist. Existieren mehrere Funktionen mit gleichem Konversionsaufwand, so hat eine Fehlermeldung zu erfolgen, wie beim folgenden Code:

Demo-Programm 5_09_01.cc

```
#include <iostream.h>

int func (int i, int j) { return i == j; }

int func (double x, double y) { return x == y; }

int main () { cout << func (3.141592, 3) << "\n"; }
```

In diesem Fall liegt eine Mehrdeutigkeit darin, dass ein Aufruf `func(double, int)` erfolgt, also auf jeden Fall eines der beiden Argumente in die Type des anderen konvertiert werden muss. Wenn beide Argumente in **`int`** konvertiert werden, wird `func` mit zwei identischen Werten aufgerufen; das Ergebnis ist dann 1. Bei Konversion in **`double`** sind beide Argumente von `func` verschieden; das Ergebnis wäre dann 0. Der GNU C++-Compiler gibt beim Übersetzen dieses Programms zwar eine Warnung

GNU-C/C++-spezifisch

```
5_09_01.cc: In function 'int main()':
5_09_01.cc:17: warning: float or double used for argument 1 of 'int func(int, int)'
```

aus (unabhängig vom gewählten *Warning Level*), erstellt aber trotzdem ein lauffähiges Programm unter Verwendung von `func(int, int)`, das dann auch als Ergebnis "1" ausgibt.

Vertauscht man hingegen die Reihenfolge der Definitionen von `func(int, int)` und `func(double, double)`, so erfolgt bei der Übersetzung keine Fehlermeldung; der Compiler verwendet `func(double, double)` und gibt dementsprechend als Ergebnis "0" aus.

Demo-Programm 5_09_02.cc

Die Auswahlregeln für *overloaded* Funktionen implizieren die Regeln, wodurch sich solche Funktionen voneinander unterscheiden *müssen*, um als unterschiedliche Funktionen zu gelten:

- Anzahl und/oder Typen der Argumente müssen unterschiedlich sein.
- Unterschiede im Typ des Resultats allein sind *nicht* ausreichend.
- In der Argumentliste gelten Objekte und Referenzen auf ein Objekt gleichen Typs als identisch:

  ```
  int func (int&, double&);
  ```

  ist identisch zu

  ```
  int func (int, double);
  ```

- Eindimensionale Felder und Zeiger der gleichen Type sind identisch. Mehrdimensionale Felder sind nur dann unterschiedlich, wenn sich ihre zweiten und folgenden Felddimensionen voneinander unterscheiden.
- Klassenelement-Funktionen dürfen sich nicht nur darin unterscheiden, dass eine **static** ist und die andere nicht.
- **typedef**-Definitionen werden in die ihnen zugrundeliegende Type umgesetzt. Eine Type und ein ihr mit **typedef** zugewiesenes Synonym sind daher identisch.
- Aufzählungstypen können zur Unterscheidung herangezogen werden.
- Wenn Referenzen oder Zeiger mit **const** oder **volatile** als Attribut versehen sind, werden sie als unterschiedlich von der Basistype betrachtet. Das folgende Konstrukt ist daher zulässig:

Demo-Programm 5_09_03.cc

```
#include <iostream.h>

class Ol
{
public:
    Ol() { cout << "Default-Konstruktor fuer Ol\n";}
        // Default-Konstruktor (#1)
    Ol(Ol& o) { cout << "Ol&\n"; }
        // Kopier-Konstruktor mit Referenz-Argument (#2)
    Ol(const Ol& o) {cout <<"const Ol&\n";}
        // Kopier-Konstruktor mit const Referenz-Argument (#3)
    Ol(volatile Ol& o) { cout << "volatile Ol&\n"; }
        // Kopier-Konstruktor mit volatile Referenz-Argument (#4)
};

int main ()
{
    Ol o1;                      // Default-Konstruktor (#1)
    Ol o2(o1);                  // Ol(Ol&) (Konstruktor #2)
    const Ol o3;                // Default-Konstruktor (#1)
    Ol o4(o3);                  // Ol(const Ol&) (Konstruktor #3)
    volatile Ol o5;             // Default-Konstruktor (#1)
    Ol o6(o5);                  // Ol(volatile Ol&) (Konstruktor #4)
}
```

Das Programm ruft den Default-Konstruktor und die drei Varianten des Kopier-Konstruktors auf; seine Ausgabe ist:

```
Default-Konstruktor fuer Ol
Ol&
Default-Konstruktor fuer Ol
const Ol&
Default-Konstruktor fuer Ol
volatile Ol&
```

GNU-C/C++-spezifisch

(Ein interessanter Bug in GNU C++: Umlaute (= ASCII-Codes > 127) innerhalb von Strings werden im Kontext einer "normalen" Funktion problemlos akzeptiert. Das "ü" in "für", innerhalb des Konstruktors mit `cout` ausgegeben, würde jedoch eine Serie von Fehlermeldungen auslösen; aus diesem Grund musste hier "fuer" geschrieben werden.)

Funktionen werden nur dann als *overloaded* behandelt, wenn sie sich innerhalb desselben Gültigkeitsbereichs befinden. Funktionen innerhalb eines übergeordneten Gültigkeitsbereichs werden verborgen; ebenso werden Funktionen einer Basisklasse mit gleichem Namen verborgen. Das folgende Beispiel definiert zwei Funktionen `func1`, eine mit einem Argument vom Typ **`double`** und eine mit einem Argument vom Typ **`int`**, die einerseits global, andererseits in einer anderen Übersetzungseinheit definiert und lokal deklariert werden, sowie zwei Klassenelement-Funktionen `func` in einer Basisklasse und in einer abgeleiteten Klasse, die sich in gleicher Weise durch den Typ ihrer Argumente unterscheiden. Obwohl beide Funktionen in `main()` mit einem Argument vom Typ **`double`** aufgerufen werden, sollte eigentlich (laut offizieller Definition von C++) in beiden Fällen die Funktion mit dem **`int`**-Argument ausgeführt werden, weil sie jeweils die Funktion mit dem **`double`**-Argument verbirgt. Tatsächlich betrachtet jedoch (inkorrekterweise) der GNU C++-Compiler beide Funktionen `func1` als im selben Gültigkeitsbereich und wählt `func1(double)`:

GNU-C/C++-spezifisch

Demo-Programm 5_09_04.cc

```
// Übersetzungseinheit A

#include <iostream.h>

void func1 (int i) { cout << "Lokale Funktion func1(int)\n"; }

// Übersetzungseinheit B

#include <iostream.h>

class Basis                                     // Basisklasse
{
public:
    void func (double x) { cout << "func(double) in Basis\n"; }
};

class Abgel : public Basis                      // Abgeleitete Klasse
{
public:
    void func (int i) { cout << "func(int) in Abgel\n"; }
};

void func1 (double x)                           // als global definiert
{
    cout << "Globale Funktion func1(double)\n";
}

int main ()
{
    extern void func1 (int i);   // als lokal für main() deklariert
    Abgel a;                     // Objekt der Klasse Abgel

    a.func (3.141592);
        // Klassenelement-Funktion mit double-Argument aufgerufen

    func1 (1.4142);
        // "gewöhnliche" Funktion mit double-Argument aufgerufen
}
```

Damit ist die Ausgabe dieses Programms (nicht ganz den Spezifikationen entsprechend):

```
func(int) in Abgel
Globale Funktion func1(double)
```

5.9.2. *Operator Overloading*

5.9.2.1. Allgemeines

Die Funktion der meisten Operatoren von C++ kann bei Bedarf für bestimmte Datentypen geändert werden. Diese Neudefinition ist entweder global oder im Rahmen einer Klasse möglich. Die Implementierung erfolgt durch globale oder Klassenelement-Funktionen. Der Name einer Funktion zur Definition eines *overloaded* Operators ist **`operator`** ***op*** oder **`operator`*op***, wobei ***op*** einer der Operatoren in der nachfolgenden Tabelle ist:

Operatoren, die durch *Operator Overloading* umdefiniert werden können:

Operator	Bezeichnung	Type
,	Komma	binär
!	Logische Negation	unär
!=	Ungleichheit	binär
%	Modulo	binär
%=	Modulo — Zuweisung	binär
&	Bitweises UND	binär
&	Adresse von	unär
&&	Logisches UND	binär
&=	Bitweises UND — Zuweisung	binär
()	Funktionsaufruf	—
*	Multiplikation	binär
*	Zeiger dereferenzieren (Indirektion)	unär
*=	Multiplikation — Zuweisung	binär
+	Addition	binär
+	Unäres Plus	unär
++	Inkrement (Präfix oder Postfix)	unär
+=	Addition — Zuweisung	binär
-	Subtraktion	binär
-	Unäres Minus	unär
--	Dekrement (Präfix oder Postfix)	unär
-=	Subtraktion — Zuweisung	binär
->	Elementauswahl	binär
->*	Elementauswahl mit Zeiger auf Element	binär
/	Division	binär
/=	Division — Zuweisung	binär
<	Kleiner als	binär
<<	Linksverschiebung	binär
<<=	Linksverschiebung — Zuweisung	binär
<=	Kleiner oder gleich	binär
=	Zuweisung	binär
==	Gleichheit	binär
>	Größer als	binär
>=	Größer oder gleich	binär

Operator	Bezeichnung	Type
`>>`	Rechtsverschiebung	binär
`>>=`	Rechtsverschiebung — Zuweisung	binär
`[]`	Feldindex	—
`^`	Exklusiv-ODER	binär
`^=`	Exklusiv-ODER — Zuweisung	binär
`|`	Bitweises ODER	binär
`|=`	Bitweises ODER — Zuweisung	binär
`||`	Logisches ODER	binär
`~`	Einer-Komplement	unär
`delete`	Speicherverwaltung — Rückgabe	—
`new`	Speicherverwaltung — Allokierung	—

Operatoren, die *nicht* durch *Operator Overloading* umdefiniert werden können:

Operator	Bezeichnung
`.`	Elementauswahl
`.*`	Elementauswahl mit Zeiger auf Element
`::`	Gültigkeitsbereich
`? :`	Bedingte Ausführung
`sizeof`	Objektgröße
`#`	Präprozessor-Befehl
`##`	Präprozessor-Befehl

Die folgenden Regeln für *Operator Overloading* gelten generell (mit Ausnahme der Operatoren **`new`** und **`delete`**):

- Operator-Funktionen müssen entweder Klassenelement-Funktionen sein oder ein Argument haben, das einer Klassen- oder Aufzählungstype angehört oder eine Referenz auf eine solche ist.
- Es dürfen beliebig viele Implementierungen für einen bestimmten Operator vorgesehen werden (siehe auch das Beispiel auf Seite 130).
- Für benutzerdefinierte Implementierungen eines Operators gelten weiterhin die gleichen Regeln bezüglich Präzedenz sowie Anzahl und Anordnung der Operanden wie für die Standard-Definition.
- Wenn ein *unärer* Operator als Klassenelement-Funktion definiert wird, hat er *kein* Argument; als globale Funktion benötigt er *ein* Argument.
- Wenn ein *binärer* Operator als Klassenelement-Funktion deklariert wird, benötigt er *ein* Argument; als globale Funktion definiert benötigt er *zwei*.
- Die Verwendung von voreingestellten Argumenten für *overloaded* Operatoren ist unzulässig.
- Abgeleitete Klassen erben alle Operatoren ihrer Basisklasse mit Ausnahme der Funktion **`operator=()`**.
- Das erste Argument von als Klassenelement-Funktionen definierten *overloaded* Operatoren hat immer die Type der Klasse, für die/mit der der Operator aufgerufen wird (also die Type der Klasse, in der er definiert wurde, oder einer Basisklasse davon). Für dieses Argument werden keine wie immer gearteten Typ-Konversionen vorgenommen.

Operator Overloading erlaubt eine völlige Umdefinition des Verhaltens von Operatoren. Dies kann dazu führen, dass Ausdrücke, die mit der Standard-Definition der Operatoren identisch wären, sich völlig verschiedenartig verhalten können. Zweckmäßigerweise sollten Operatoren so

definiert werden, dass die erwartete Semantik erhalten bleibt, dass also beispielsweise die folgenden Ausdrücke äquivalent bleiben:

```
var = var + 1;
var += 1;
var++;
++var;
```

5.9.2.2. Unäre Operatoren

Als Klassenelement-Funktionen werden unäre *overloaded* Operatoren folgendermaßen deklariert:

Ergebnistype **operator** ***op*** **();**

Als globale Funktionen müssen sie so deklariert werden:

Ergebnistype **operator** ***op*** **(***arg***);**

Dabei ist *Ergebnistype* die (beliebige und durch nichts eingeschränkte) Type des Ergebnisses der Operator-Funktion, ***op*** einer der unären Operatoren aus der Tabelle von Seite 158, und *arg* ein Argument mit Klassen- oder Aufzählungstyp.

Bei den Inkrement- und Dekrement-Operatoren ("`++`" bzw. "`--`") wird folgendermaßen zwischen der Präfix- ("`++i`") und der Postfix-Version ("`i++`") unterschieden:

- Die Präfix-Version des Operators wird exakt wie jede andere unäre Operator-Funktion deklariert;
- die Postfix-Version hat ein (zusätzliches) Argument vom Typ **`int`** (das zusätzliche Argument *muss* vom Typ **`int`** sein!).

Das folgende Beispiel illustriert die Definition und Anwendung dieser beiden Operatoren am Beispiel einer Klasse `Punkt`:

Demo-Programm 5_09_05.cc

```
#include <iostream.h>

class Punkt
{
public:
    // Deklaration der Inkrement-Operatoren:
    Punkt& operator++();                        // Präfix
    Punkt operator++(int);                      // Postfix

    // Definition der Dekrement-Operatoren:
    Punkt& operator--()                         // Präfix
    {
        _x--;                                   // Standard-Dekrement
        _y--;
        return *this;
    }

    Punkt operator--(int)                       // Postfix
    {
        Punkt temp = *this;                     // temporäres Objekt
        --*this;                                // ruft operator--() auf
        return temp;
    }


    Punkt (const short& x = 0, const short& y = 0) { _x = x; _y = y; }
        // (Default-) Konstruktor

    Zeige (char *s) { cout << s << ": x = " << _x << ", y = " << _y << "\n"; }

private:
    short _x, _y;
};
```

```
// Definition der Inkrement-Operatoren:

Punkt& Punkt::operator++()                    // Präfix
{
    _x++;                                     // Standard-Inkrement
    _y++;
    return *this;
}

Punkt Punkt::operator++(int)                  // Postfix
{
    Punkt temp = *this;                       // temporäre Kopie
    ++*this;                                  // ruft operator++() auf
    return temp;
}

int main ()
{
    Punkt p1 (3, 5);

    p1.Zeige("p1");
    ++p1;                                     // Präfix-Inkrement
    p1.Zeige("p1");

    Punkt p2 = p1++;                          // Postfix-Inkrement

    p1.Zeige("p1");
    p2.Zeige("p2");

    --p1;                                     // Präfix-Dekrement
    p1.Zeige("p1");
    p2 = p1--;                                // Postfix-Dekrement

    p1.Zeige("p1");
    p2.Zeige("p2");
}
```

Einige Bemerkungen zu diesem Demo-Programm:

- Die Definition von **operator*x*()**-Klassenelement-Funktionen kann wie die jeder anderen Klassenelement-Funktion innerhalb oder außerhalb der Definition der Klasse erfolgen. Zur Demonstration der Syntax wurden hier beide Methoden für die Dekrement- bzw. die Inkrement-Operatoren verwendet.
- Als Konstruktor der Klasse wurde eine Funktion verwendet, die sowohl als "gewöhnlicher" Konstruktor mit funktionsartiger Übergabe der Initialisierungswerte als auch als Default-Konstruktor (voreingestellte Argumente) verwendet werden kann. Die Verwendung von **const short&**-Argumenten ist wegen des Referenz-Typs der Argumente hier notwendig, um eine Initialisierung mit Konstanten zu erlauben (`Punkt p1(3, 5);`).

Das Programm erzeugt, wie aus der Standard-Funktion der Prä- und Postfix-Operatoren erwartet, die folgende Ausgabe:

```
p1: x = 3, y = 5
p1: x = 4, y = 6
p1: x = 5, y = 7
p2: x = 4, y = 6
p1: x = 4, y = 6
p1: x = 3, y = 5
p2: x = 4, y = 6
```

Alternativ hätten die **operator++()** und **operator--()**-Funktionen auch global definiert und als **friend** deklariert werden können:

Demo-Programm 5_09_06.cc

```
#include <iostream.h>

class Punkt
{
public:
    friend Punkt& operator++(Punkt&);   // Deklaration der Präfix-Operatoren
    friend Punkt& operator--(Punkt&);
                                        // Deklaration der Postfix-Operatoren
    friend Punkt operator++(Punkt&, int);
    friend Punkt operator--(Punkt&, int);

    Punkt (const short& x = 0, const short& y = 0) { _x = x; _y = y; }
        // (Default-) Konstruktor

    Zeige(char *s) { cout << s << ": x = " << _x << ", y = " << _y << "\n"; }

private:
    short _x, _y;
};

// Definition der Inkrement-Operatoren:

Punkt& operator++(Punkt& p)                    // Präfix
{
    p._x++;                                    // Standard-Inkrement
    p._y++;
    return p;
}

Punkt operator++(Punkt& p, int)                // Postfix
{
    Punkt temp = p;                            // temporäre Kopie
    ++p;                                       // ruft operator++() auf
    return temp;
}

// Definition der Dekrement-Operatoren

Punkt& operator--(Punkt& p)                    // Präfix
{
    p._x--;                                    // Standard-Dekrement
    p._y--;
    return p;
}

Punkt operator--(Punkt& p, int)                // Postfix
{
    Punkt temp = p;                            // temporäre Kopie
    --p;                                       // ruft operator--() auf
    return temp;
}
```

Mit der gleichen Funktion `main()` wie im vorigen Beispiel erhält man wieder das selbe Ergebnis wie zuvor.

Das **int**-Argument der Postfix-Version der Inkrement- und Dekrement-Operatoren wird normalerweise nicht benutzt, obwohl dies prinzipiell möglich wäre. Es enthält bei gewöhnlichem Aufruf des Operators den Wert 0. Das folgende Beispiel zeigt eine potentielle Verwendung dieses Arguments:

Demo-Programm 5_09_07.cc

```
#include <iostream.h>

class Int                                  // Neudefinition von int
{
public:
    Int (const int& i = 0) { _i = i; } // (Default- und Kopier-) Konstruktor
```

```
    Int& operator++(int n)
    {
        if (n)                                  // Spezial-Aufruf
            _i += n;
        else                                    // "gewöhnlicher" Aufruf
            _i++;

        return *this;
    }

    Show () { cout << "i = " << _i << "\n"; }

private:
    int _i;
};

int main()
{
    Int i = 3;                                  // "Int", NICHT "int"
    i++;                                        // "gewöhnlicher" Aufruf
    i.Show();
    i.operator++ (7);                           // Spezial-Aufruf
    i.Show();
}
```

Das Programm hat die Ausgabe:

```
i = 4
i = 11
```

Eine Übergabe des **int**-Parameters an die Funktion **operator++()** ist nur durch einen expliziten Aufruf wie im obigen Beispiel möglich.

5.9.2.3. Binäre Operatoren

Als Klassenelement-Funktionen werden binäre *overloaded* Operatoren wie folgt deklariert:

Ergebnistype **operator** ***op*** **(***arg***);**

Als globale Funktionen müssen sie so deklariert werden:

Ergebnistype **operator** ***op*** **(***arg1, arg2***);**

Dabei ist *Ergebnistype* die (beliebige und durch nichts eingeschränkte) Type des Ergebnisses der Operator-Funktion, ***op*** einer der binären Operatoren aus der Tabelle von Seite 158, und *arg* (bzw. *arg1* und *arg2*) ein Argument; *arg* kann von beliebiger Type sein, während von *arg1* und *arg2* mindestens eines eine Klassen- oder Aufzählungstype haben muss. Spezielle binäre Operatoren sind der Zuweisungs-Operator ("**=**"), der Funktionsaufruf ("**( )**"), der Feldindex-Operator ("**[ ]**") und die Elementauswahl-Operatoren ("**->**" und "**->***").

5.9.2.4. Der Zuweisungsoperator (**operator=**)

Die Deklaration des Zuweisungsoperators ist identisch zu der aller anderen binären Operatoren, mit den folgenden Ausnahmen:

- Die Funktion **operator=** *muss* eine nicht-statische Klassenelement-Funktion sein. Eine Definition als globale Funktion ist unzulässig.
- Die Funktion **operator=** wird nicht von abgeleiteten Klassen geerbt.
- Sofern keine explizite Definition von **operator=** erfolgte, generiert der Compiler eine solche Funktion.

Das folgende Beispiel zeigt eine Implementierung von **operator=**:

Demo-Programm 5_09_08.cc

```
#include <iostream.h>

class Point
{
public:
    // Standard- und Default-Konstruktor:
    Point (int x = 0, int y = 0) { _x = x; _y = y; }

    Point& operator=(const Point& pt)  // Zuweisungs-Operator
    {
        _x = 3*pt._x;
        _y = 5*pt._y;
        return *this;
    }

    show () { cout << "x = " << _x << ", y = " << _y << "\n"; }

private:
    short _x, _y;
};

int main ()
{
    Point pt1 (10, 20);
    Point pt2, pt3;

    pt1.show();
    pt2.show();
    pt3.show();
    pt3 = pt2 = pt1;             // mehrfache benutzerdefinierte Zuweisung
    pt1.show();
    pt2.show();
    pt3.show();
}
```

Die **operator=**-Funktion gibt den Wert des Objekts, für das sie aufgerufen wurde (***this**), als Resultat zurück. Das ist notwendig, um die auch sonst in C/C++ übliche mehrfache Zuweisung (wie auch in unserem Beispiel) realisieren zu können. Zur Illustration wurden die beiden Koordinaten des zuzuweisenden Wertes mit 3 bzw. 5 multipliziert (was erlaubt, aber ansonsten eher nicht üblich ist).

Das Programm hat die Ausgabe:

```
x = 10, y = 20
x = 0, y = 0
x = 0, y = 0
x = 10, y = 20
x = 30, y = 100
x = 90, y = 500
```

5.9.2.5. Der Funktionsaufruf (**operator()**)

Der Funktionsaufruf-Operator muss — ähnlich wie der Zuweisungs-Operator — eine nicht-statische Klassenelement-Funktion sein. Er erlaubt die Ausführung von Operationen, die eine beliebige Zahl von Parametern (= Operanden) benötigen. Das folgende Beispiel zeigt eine Anwendung, die eine Änderung der Koordinaten eines Punktes in *einer* Operation erlaubt:

Demo-Programm 5_09_09.cc

```
#include <iostream.h>

class Punkt
{
public:
    Punkt() { _x = _y = 0; }
    Punkt& operator() (short dx, short dy) { _x += dx; _y += dy; return *this; }
    Zeige() { cout << "x = " << _x << ", y = " << _y << "\n"; }
```

```
private:
    short _x, _y;
};

int main()
{
    Punkt pt;
    pt.Zeige();
    pt (3, 5);
    pt.Zeige();
    pt (7, 9);
    pt.Zeige();
}
```

Durch die Benutzerdefinition der Funktionsklammern im Kontext der Klasse `Punkt` wird der ansonsten sinnlose (und zu einem fatalen Fehler führende Ausdruck) "`pt (3, 5);`" sinnvoll. Beachten Sie bitte, dass die Funktionsklammern hier dem Namen eines *Objekts* und nicht dem einer *Funktion* folgen! *Operator Overloading* des Funktionsaufruf-Operators definiert nicht den existierenden Funktionsaufruf-Operator neu, sondern *erweitert* seine Anwendbarkeit im Zusammenhang mit Objekten, die *nicht* Funktionen sind.

Das Programm erzeugt die Ausgabe:

```
x = 0, y = 0
x = 3, y = 5
x = 10, y = 14
```

5.9.2.6. Der Feldindex-Operator (**operator[]**)

Der Feldindex-Operator ist wie der Funktionsaufruf-Operator ein *binärer* Operator. Er muss — ähnlich wie der Zuweisungs-Operator — eine nicht-statische Klassenelement-Funktion sein, darf aber nur genau ein Argument von beliebiger Type haben, das den gewünschten Feldindex angibt. Das folgende Beispiel zeigt eine Anwendung eines *overloaded* Feldindex-Operators, die eine Prüfung der Feldgrenzen eines eindimensionalen Feldes (Vektors) beinhaltet:

Demo-Programm 5_09_10.cc

```
#include <iostream.h>

class IntVektor
{
public:
    int& operator[](int iIndex)
    {
        static int iFehler = 0;              // "Schutz"-Variable

        if (iIndex >= 0 && iIndex < _iG)
            return _iE [iIndex];
        else                                 // Überlauf
        {
            cout << "Felduebeerlauf Element ["<< iIndex << "] !\n";
            return iFehler;
        }
    }

    IntVektor (int nElem = 1)                // Konstruktor
    {
        _iE = new int [nElem];               // allokieren
        _iG = nElem;                         // Feldgröße
    }

    ~IntVektor() { delete[] _iE; }           // Destruktor

private:
    int *_iE;                                // Zeiger auf Feldelemente
    int _iG;                                 // Feldgröße
};
```

```
int main()
{
    IntVektor v (10);                          // Vektor mit 10 Elementen

    for (int i = -1; i <= 10; i++)
        v[i] = i;                              // initialisiere Feld

    v[2] = v[7];                               // teste Zuweisung
    cout << "Lesen ...\n";

    for (i = -1; i <= 10; i++)                 // Ausgabe
        cout << "Element v[" << i << "] = " << v[i] << "\n";
}
```

Die Funktion `IntVektor::operator[]()` greift auf ein Feldelement nur dann zu, wenn der Index im zulässigen Bereich (`0` … *`Feldgröße`* `- 1`) liegt. Anderenfalls wird eine Referenz auf eine statische (und daher permanent verfügbare) Hilfsvariable `iFehler` als Ergebnis zurückgegeben. Beachten Sie bitte, dass das Resultat der **`operator[]`**-Funktion eine *Referenz* auf **`int`** ist, dass sie also sowohl zum Auslesen als auch zum Setzen eines Feldelementes verwendet werden kann. Die Variable `iFehler` wird daher bei jedem unzulässigen Schreibzugriff auf den zu schreibenden Wert gesetzt; in unserem Demo-Programm also zuletzt auf den Wert `10`. Gleichzeitig wird eine Fehlermeldung generiert. Beim Lesen eines unzulässigen Feldelementes wird der (zuletzt geschriebene) Wert von `iFehler` zurückgegeben. Die Ausgabe des Programms ist damit:

```
Felduebеrlauf Element [-1] !
Felduebеrlauf Element [10] !
Lesen ...
Felduebеrlauf Element [-1] !
Element v[-1] = 10
Element v[0] = 0
Element v[1] = 1
Element v[2] = 7
Element v[3] = 3
Element v[4] = 4
Element v[5] = 5
Element v[6] = 6
Element v[7] = 7
Element v[8] = 8
Element v[9] = 9
Felduebеrlauf Element [10] !
Element v[10] = 10
```

GNU-C/C++-spezifisch

Der bereits auf Seite 156 erwähnte Bug in GNU C++ verbietet wieder die Verwendung des "`ü`" in "`Feldüberlauf`".

Falls es stört, dass der Rückgabewert im Fehlerfall undefiniert ist, ist eine einfache Modifikation der Funktion **`operator[]()`** möglich:

Demo-Programm 5_09_11.cc

```
int& IntVektor::operator[](int iIndex)
{
    static int iFehler;          // "Schutz"-Variable

    iFehler = 0;                 // Zuweisung des Wertes 0 bei jedem Durchlauf

    if (iIndex >= 0 && iIndex < _iG)
        return _iE [iIndex];
    else                         // Überlauf
    {
        cout << "Felduebеrlauf Element [" << iIndex << "] !\n";
        return iFehler;
    }
}
```

Diese Variante der **`operator[]`**-Funktion könnte beispielsweise so gestaltet werden, dass im Fehlerfall der (für eine "normale" **`int`**-Variable unzulässige) Wert `0x80000000` (für 32-Bit-Systeme) bzw. `0x8000` (für 16-Bit-Systeme) zurückgegeben werden könnte; das aufrufende Programm könnte dann auf Vorhandensein dieses Wertes testen und korrektive Maßnahmen vorsehen.

5.9.2.7. Die Elementauswahl-Operatoren

Die Elementauswahl-Operatoren "`->`" und "`->*`" können ebenfalls umdefiniert werden; sie gelten als *unäre* Operatoren. (Eine Umdefinition von "`.`" und "`.*`" ist jedoch nicht möglich.) Sie müssen im Rahmen einer Klasse definiert werden und einen Zeiger auf diese Klasse als Ergebnis zurückgeben. Die umdefinierten Elementauswahl-Operatoren könnten beispielsweise verwendet werden, um den Zugriff auf Objekte über ihre Zeiger zu sichern:

Demo-Programm 5_09_12.cc

```
#include <iostream.h>

const int magic = 0x12345678;                  // "magische" Zahl

class Punkt
{
public:
    Punkt *operator->()
    {
        if (_magic == magic)                   // Zeiger ok
            return this;
        else                                   // falscher Zeiger
        {
            cout << "Falscher Zeiger!\n";
            return &_ptFehler;
        }
    }

    Punkt (short x = 0, short y = 0) { _x = x; _y = y; _magic = magic;}

    Zeige () { cout << "x = " << _x << ", y = " << _y << "\n"; }

private:
    short _x, _y;
    int _magic;                                // "magische" Zahl
    static Punkt _ptFehler;
        // Objekt, das im Fehlerfall angesprochen werden kann
};

Punkt Punkt::_ptFehler (3, 5);   // Definition des statischen Objekts

int main ()
{
    Punkt pt (11, 13);           // Objekt
    Punkt *ppt = &pt;            // Zeiger darauf

    ppt->Zeige();                // Zugriff ok - Resultat ist (11, 13)

    ppt++;                       // ppt zeigt irgendwo hin
    ppt->Zeige();                // Fehler - Resultat sollte (3, 5) sein
}
```

Der Grundgedanke hinter diesem Programm ist, dass in jedem Objekt der Klasse `Punkt` im Element `_magic` die gleiche "magische Zahl" (hier `0x12345678`) stehen sollte. Wenn diese nicht vorhanden ist, wird statt des Zeigers auf das aufrufende Objekt (**`this`**) ein Zeiger auf ein statisches Hilfs-Objekt der Klasse (`_ptFehler`) zurück- und eine Fehlermeldung ausgegeben.

GNU-C/C++-spezifisch

Tatsächlich generiert der GNU C++-Compiler zwar eine Funktion für **`operator->()`** (mit dem internen dekorierten Namen `___rf__5Punkt`), ruft diese aber in keinem der beiden Fälle auf, in denen in `main()` ein Zeiger auf `Punkt` dereferenziert wird. (Er tut dies auch nicht, wenn nicht eine Klassenelement-Funktion, sondern ein Datenelement angesprochen werden soll.) Damit entfällt die Schutzfunktion, derentwegen der *Overload* für "`->`" ja definiert wurde. Nicht ganz den Intentionen des Programms gemäß ist daher seine Ausgabe:

```
x = 11, y = 13
x = 0, y = 0
```

5.9.2.8. Die Operatoren **new** und **delete**

Wie schon auf Seite 81 erwähnt, können auch Benutzer-Definitionen des Operators **new** (und auch von **delete)** vorgesehen werden. Die Funktion **operator new()** muss mindestens ein Argument der Type `size_t` haben (die in `stddef.h` definiert ist); weitere Argumente sind mit beliebiger Zahl und Type möglich. Das Ergebnis von **operator new()** hat immer die Type **void***. Die Funktion **operator delete()** muss mindestens ein Argument der Type **void*** haben; ein weiteres Argument der Type `size_t`, das die Größe des zurückzugebenden Speicherblocks enthält, ist optional. **operator delete()** hat die Ergebnistype **void**. Bei Definition als Klassenelement-Funktionen sind beide Funktionen statische; sie können daher nicht virtuell sein.

Für die Allokierung von Feldern wird *immer* der globale (Standard-) **operator new()** verwendet; ebenso für die Allokierung von Objekten der fundamentalen Typen und von Klassen, die keine Definition für **operator new()** vorgesehen haben.

Im folgenden Beispiel wird eine benutzerdefinierte Implementierung von **operator new()** verwendet, um Speicher (unter Verwendung der ANSI-C-Speicherverwaltungs-Funktion `malloc()`) vom *Heap* zu allokieren und (mit der Standard-C-Funktion `memset()`) mit einem bestimmten Wert zu initialisieren:

Demo-Programm 5_09_13.cc

```
#include <iostream.h>
#include <malloc.h>                             // für malloc()
#include <memory.h>                             // für memset()

class Obj
{
public:
    Show () { cout << "Objekt = " << hex << _obj << "\n"; }

    void *operator new (size_t n, char Ini)
    {
        void *pTemp = malloc (n);               // allokiere Speicher

        if (pTemp)                              // malloc() war ok
            memset (pTemp, Ini, n);             // initialisiere Speicher

        return pTemp;
    }

private:
    int _obj;
};

int main ()
{
    Obj *po1 = new (0x5a) Obj;
    po1->Show();                                // Einzel-Objekt

    Obj *po2 = new (0x71) Obj [10];
    po2[3].Show();                              // Feld
}
```

Das Demo-Programm definiert einen **operator new()**, der das gesamte allokierte Objekt (in diesem Fall eine **int**-Variable) auf einen beliebigen, als Argument übergebenen Wert setzt. Nicht ganz konform mit der offiziellen C++-Spezifikation (wonach für die Allokierung von Feldern ja der *globale* **operator new()** verwendet werden sollte) wird der benutzerdefinierte **operator new()** auch bei der Allokierung eines Feldes (`new (0x71) Obj [10]`) verwendet, sodass die folgende Ausgabe resultiert:

```
Objekt = 5a5a5a5a
Objekt = 71717171
```

Im folgenden Beispiel führen die global definierten Operatoren **new** und **delete** über ihre Speicherallokationen Buch:

Demo-Programm 5_09_14.cc

```
#include <iostream.h>
#include <stdlib.h>                     // für size_t
#include <malloc.h>

int _fLog = 0;                          // Flag: "Buchführung" ein/aus
int BAll = 0;                           // Zahl der allokierten Speicherblöcke

void *operator new (size_t n)           // global definierter operator new
{
    static int in_new = 0;              // Schutz-Flag

    if (_fLog && ! in_new)
    {
        in_new = 1;                     // verhindere rekursive Zugriffe
        cout << "Speicherblock " << ++BAll << " mit " << n <<
            " Bytes allokiert\n";
        in_new = 0;
    }

    return malloc (n);
}

void operator delete (void *pMem)       // global definierter operator delete
{
    static int in_del = 0;              // Schutz-Flag

    if (_fLog && ! in_del)
    {
        in_del = 1;                     // verhindere rekursive Zugriffe
        cout << "Speicherblock deallokiert; noch " << --BAll << " Blöcke\n";
        in_del = 0;
    }

    free (pMem);
}

int main ()
{
    char *Blocks[5];

    _fLog = 1;                          // Buchführung einschalten

    for (int i = 0; i < 5; i++)
        Blocks[i] = new char[100];

    for (i = 0; i < 5; i++)
        delete Blocks[i];
}
```

Die "Buchführungs-Funktion" der Operatoren **new** und **delete** wird mit der globalen Variablen `_fLog` ein- und ausgeschaltet. In den beiden Funktionen **operator new()** und **operator delete()** wird je eine Flag-Variable (`in_new` bzw. `in_del`) verwendet, um mehrfache Meldungen zu vermeiden, die dann auftreten würden, wenn im Zuge der Stream-Ausgabe mit `cout` Speicher allokiert wird. Die Ausgabe des Programms ist:

```
Speicherblock 1 mit 100 Bytes allokiert
Speicherblock 2 mit 100 Bytes allokiert
Speicherblock 3 mit 100 Bytes allokiert
Speicherblock 4 mit 100 Bytes allokiert
Speicherblock 5 mit 100 Bytes allokiert
Speicherblock deallokiert; noch 4 Blöcke
Speicherblock deallokiert; noch 3 Blöcke
Speicherblock deallokiert; noch 2 Blöcke
Speicherblock deallokiert; noch 1 Blöcke
Speicherblock deallokiert; noch 0 Blöcke
```

5.10. Konventionelle und objektorientierte Programmierung

Grundsätzlich sind alle Problemstellungen auch mit *konventioneller* Programmierung (also z.B. in Standard-C) bearbeitbar. (Frühe Versionen von C++-Compilern setzten den C++-Quellcode zunächst in Standard-C-Code um, der dann mit einem gewöhnlichen C-Compiler übersetzt werden konnte.) *Objektorientierte* Programmierung bietet jedoch entscheidende *Vorteile* in den folgenden Situationen:

- In einem System (Programm oder Gruppe von Programmen) kommt eine größere Zahl von unterschiedlichen Objekten vor, die *ähnliche*, aber doch *unterschiedliche* Behandlung erfordern, wie:
  - Datenbanken für heterogene Objekte;
  - graphische Systeme, die Ausgabefunktionen für diverse geometrische Figuren oder sonstige Ausgabeelemente benötigen.
- Es sollen Objekte in einer transparenten Form miteinander in Beziehung gesetzt werden, für die es keine einfachen arithmetischen oder logischen Operatoren gibt, beispielsweise:
  - komplexe Zahlen;
  - Vektoren.
- Die Details der Implementierung gewisser Funktionen sind uninteressant.
- Es sollen *abstrakte Konzepte* für Operationen an oder mit möglichst vielen verschiedenartigen Objekten definiert werden, beispielsweise für:
  - Warteschlangen, Stacks und ähnliches;
  - Daten-Kommunikation.

Der große Vorteil einer objektorientierten gegenüber einer konventionellen Programmierung beruht auf den folgenden in C++ verfügbaren Mechanismen:

➡ *Function* und *Operator Overloading*; virtuelle Funktionen*:*

Klassenspezifische Funktionen und Operatoren können als Teil einer Klasse definiert werden. Damit ist es unmöglich, ein Objekt mit einer ungeeigneten Funktion zu bearbeiten. Funktionen oder Operatoren, die mit *Overloading* mehrfach definiert wurden, scheinen unter *einem* konsistenten Namen auf.

➡ Vererbung*:*

Verwandte Objekt-Typen können sich auf eine oder mehrere gemeinsame Basis-Typen beziehen. Die Gemeinsamkeiten stehen in den Basisklassen, die Unterschiede in den abgeleiteten Klassen. Änderungen von Struktur oder Definition der Objekt-Typen betreffen damit immer nur begrenzte Teile des Programms; Nebenwirkungen sind weniger wahrscheinlich.

➡ Einkapselung:

Die Daten eines bestimmten Typs sind mit den für ihre Bearbeitung benötigten Funktionen eng verbunden, unter Umständen mit begrenzten Zugriffsmöglichkeiten auf die Daten und/oder Funktionen. Damit werden unerwünschte Auswirkungen von Änderungen oder Erweiterungen des Programms unwahrscheinlicher.

➡ Schablonen:

Grundlegende Funktionen können in allgemeiner Form definiert werden. Die Programmentwicklung wird dadurch einfacher, die Programme übersichtlicher und die Gefahr von Fehlern geringer.

Damit bietet die objektorientierte Programmierung gegenüber der konventionellen die folgenden *Vorteile*:

- Programme mit (sinnvoll eingesetzten) Funktionen der objektorientierten Programmierung sind besser strukturiert und daher übersichtlicher.
- Fehlerquellen fallen weg, oder Fehler können vom Compiler erkannt werden. Einen wesentlichen Beitrag dazu leistet die strikte Typenprüfung und der durch die Einführung der Klassen erzielbare Wegfall einer großen Zahl globaler Variablen. Manche Datenobjekte können komplett innerhalb der Definition einer Klasse eingekapselt werden.
- Die *Wartbarkeit* eines Programms wird verbessert. Eine Erweiterung des Funktionsumfangs erfordert weniger Aufwand; Nebenwirkungen können leichter vermieden werden.

Der Einsatz objektorientierter Methoden erfordert im allgemeinen einen etwas größeren Planungsaufwand: Konzepte müssen *vor* dem Beginn der eigentlichen Programmierarbeit erstellt und potentielle Erweiterungen zumindest in ihren Grundzügen überlegt werden.

Objektorientierte Methoden können — je nach Aufgabenstellung — in unterschiedlichem Umfang eingesetzt werden; sinnvoll sind sie dann *und nur dann*, wenn sie einen Vorteil im Hinblick auf die Klarheit und Wartbarkeit des Programms bringen. Eine zweckmäßige Strategie könnte sein:

- Enthält das Programm Daten, die irgendwie zusammengehören (logisch oder physikalisch, weil sie beispielsweise gemeinsam abzuspeichern sind)?

  Wenn ja: Zusammengehörige Daten in Klassen-Objekten (typisch **`struct`**s) zusammenfassen.

  Beispiele: Konfigurations-Parameter eines Programms; lokale Daten einer Funktion, die diese an weitere untergeordnete Funktionen weitergeben soll.

  Bemerkung: Selbst wenn es nur *ein* Objekt einer Klasse in einem Programm gibt, trägt die Zusammenfassung zur Klarheit des Programmcodes bei und hilft Fehler zu vermeiden.

- Ist die Anzahl der zusammenzufassenden Datenelemente groß, und lassen sie sich sinnvoll in *Untergruppen* einteilen?

  Wenn ja: Klassen (wieder meist **`struct`**s) für die Untergruppen definieren; diese Untergruppen können zweckmäßigerweise in einer "Container-Klasse" zusammengefasst werden.

  Beispiele: Konfigurations-Daten; generell: stark strukturierte Daten in wenigen Objekten mit vielen Elementen.

- Gibt es Datentypen, die teils gemeinsame und teils individuell verschiedene Elemente oder Eigenschaften haben?

  Wenn ja: Die gemeinsamen Eigenschaften (Elemente) in einer Basisklasse zusammenfassen; die individuellen Unterschiede in von der Basisklasse abgeleiteten Klassen definieren.

  Beispiele: Datenbanken mit heterogenen Objekten; Behandlung der Ein- oder Ausgabe in graphisch orientierten Systemen.

  Bemerkung: Eine Strukturierung derartiger Daten ist auch *ohne* Zuhilfenahme von Funktionen der objektorientierten Programmierung möglich, aber problematisch, weil sie den Zugriff auf die Datenelemente erschwert, die Übersichtlichkeit des Programmcodes beeinträchtigt und Probleme bei späteren Änderungen oder Erweiterungen verursachen kann.

- Gibt es Funktionen, die ausschließlich oder vorwiegend auf Datenelemente *einer* Klasse zugreifen, und die sich je nach dem Typ des betroffenen Klassenobjekts (potentiell) unterschiedlich verhalten sollen?

  Wenn ja: Diese Funktionen als Elemente dieser Klassen unter Verwendung von *Function Overloading* und virtuellen Funktionen definieren.

  Beispiele: Funktionen für die Ein- und Ausgabe von Daten in einem Datenbankprogramm; Ausgabe graphischer Objekte.

- Wird die Erstellung und/oder Wartung des Programms durch Verwendung speziell für die Bearbeitung von bestimmten Objekten definierter Operatoren erleichtert?

  Wenn ja: Operatoren als Klassenelement-Funktionen (mit *Operator Overloading*) definieren.

  Beispiele: Komplexe Arithmetik; Vektorrechnung.

- Soll ein *Konzept* auf mehrere verschiedene Datentypen angewandt werden?

  Wenn ja: Dieses Konzept als Schablone definieren.

  Beispiele: Warteschlangen, Stacks, Sortieren …

6. Ein-/Ausgabe in C++

Dieser Abschnitt fasst jene — zum größten Teil als Bibliotheksroutinen realisierten — Funktionen von C/C++ zusammen, mit deren Hilfe eine Kommunikation des Benützers mit dem Programm sowie der Zugriff auf Dateien, also eine Ein- und Ausgabe von Daten, bewerkstelligt wird. Grundsätzlich sind Ein-/Ausgabe-Operationen möglich unter Verwendung

- einer Datenübergabe über zusätzliche Parameter beim Aufruf des Programms;
- der bereits in Standard-C implementierten Ein-/Ausgabefunktionen; und
- C++-spezifischer Ein-/Ausgabefunktionen.

Abschließend werden die Gesichtspunkte diskutiert, die bei einer binären Abspeicherung von Klassen-Objekten in einer Plattendatei berücksichtigt werden müssen.

6.1. Befehlszeilenparameter und Rückgabewerte

6.1.1. Befehlszeilenparameter

In C/C++ können Parameter, die mit dem Programmaufruf übergeben wurden, vom Programm aus eingelesen und verwendet werden. Dafür sind zwei optionale Argumente der Funktion `main()` vorgesehen, nämlich die Anzahl der Argumente (`argc`) und ein Feld von Zeigern von Typ **`char`** auf die einzelnen Argumente (`argv`). Der (vom Compiler eingebundene) Startup-Code des Programms "bricht" die Eingabezeile an den Leerzeichen in separate Argumente auf, die den Elementen von `argv[1]` an zugewiesen werden. Führende Leerzeichen werden entfernt. Argumente, die in doppelten Anführungszeichen (" **"** ") stehen, werden nicht getrennt. Das erste Element des Feldes `argv`, `argv[0]`, enthält vereinbarungsgemäß einen Zeiger auf jenen Befehlsstring, der zum Aufruf des Programms verwendet wurde. Der vollständige Prototyp von `main()` ist damit

```
int main (int argc, char *argv[]);
```

Das folgende Demo-Programm liest die mit seinem Aufruf übergebenen Befehlszeilenparameter ein und gibt sie der Reihe nach aus:

Demo-Programm 6_01_01.cc

```
#include <iostream.h>

int main (int argc, char *argv[])
{
    cout << argc << " Elemente in argv[]:\n";
    for (int i = 0; i < argc; i++)
        cout << "argv[" << i << "]: \"" << argv[i] << "\"\n";
}
```

Für die folgende Befehlszeile

```
C:\>go32 6_01_01 The    quick brown fox "jumps over" "the lazy white dog."
```

ist die Ausgabe dieses Demo-Programms:

```
7 Elemente in argv[]:
argv[0]: "6_01_01"
argv[1]: "The"
argv[2]: "quick"
argv[3]: "brown"
argv[4]: "fox"
argv[5]: "jumps over"
argv[6]: "the lazy white dog."
```

Der vom Compiler eingebaute Startup-Code übergibt noch ein drittes Argument an `main()`, nämlich die Startadresse eines Feldes von Zeigern auf den Typ **`char`**, die auf je eine Eintragung im Umgebungsbereich des Rechners (*Environment*) zeigen. Ebenso wie sich die Praxis eingebürgert hat, die ersten beiden Argumente von `main()` als `argc` und `argv` zu bezeichnen (was an sich nicht zwingend notwendig wäre), hat sich für das dritte Argument der Name `envp` (*Environment Pointer*) eingebürgert. Der vollständige Prototyp von `main()` ist daher:

```
int main (int argc, char *argv[], char *envp[]);
```

Wegen der Äquivalenz von Feldern und Zeigern kann man auch schreiben:

```
int main (int argc, char **argv, char **envp);
```

Das Feld `envp` wird von einem Null-Zeiger (einem Zeiger mit dem numerischen Wert `0`) abgeschlossen.

Beachten Sie bitte, dass die drei verschiedenen Aufrufs-Möglichkeiten für `main()` *kein Function Overloading* darstellen. `main()` wird vom *Startup*-Code *immer* mit den genannten drei Argumenten aufgerufen; es bleibt dem aufgerufenen Code überlassen, von den ihm übergebenen Argumenten Gebrauch zu machen oder nicht. Intern ist `main()` daher deklariert als

```
int main (...);
```

Das folgende Programm schreibt die Eintragungen im Umgebungsbereich des Rechners aus:

Demo-Programm 6_01_02.cc

```
#include <iostream.h>

int main (int argc, char *argv[], char **envp)
{
    while (*envp && **envp)
        // solange ein Zeiger != 0 vorhanden ist und auf einen nicht-leeren
        // String zeigt
            cout << *envp++ << "\n";
}
```

(Wegen des potentiell großen Umfangs der Ausgabe dieses Programms wird darauf verzichtet, hier ein Beispiel anzuführen.)

Der gleiche Zeiger, der an `main()` als `envp` übergeben wird, steht übrigens global als Variable `_environ` zur Verfügung. Zugriffe auf den Umgebungsbereich sind außerdem mit den Standard-C-Bibliotheksfunktionen

```
char *getenv(const char *name);
```

und

```
int setenv(const char *name, const char *value, int rewrite);
```

möglich (siehe On-Line-Info von GNU C++).

6.1.2. Rückgabewerte

Die meisten Betriebssysteme, so auch MS-DOS, sehen eine Möglichkeit vor, einen numerischen Wert von einem Programm an das Betriebssystem zurückzugeben, der üblicherweise den Status des Programms dokumentiert. Es hat sich eingebürgert, für einen erfolgreichen Abschluss eines Programms den Wert `0` zurückzugeben; Werte ungleich `0` zeigen üblicherweise Fehlersituationen an. (Unter MS-DOS kann beispielsweise der Rückgabewert mit dem DOS-Befehl "IF ERRORLEVEL" abgefragt werden.)

Für die Rückgabe von numerischen Resultaten aus einem C/C++-Programm stehen die folgenden Möglichkeiten offen:

- Verwendung des Befehls **`return`** mit einem (ganzzahligen) numerischen Parameter in `main()`;
- Aufruf der Funktionen `exit()` oder `_exit()` mit einem (ganzzahligen) numerischen Argument; dieser Aufruf kann aus jeder beliebigen Funktion, nicht nur aus `main()` heraus, erfolgen. `exit()` nimmt noch etliche "Aufräumungsarbeiten" vor, bevor es die Kontrolle ans Betriebssystem übergibt (insbesondere das Ausschreiben von Ausgabe-Puffern), während `_exit()` sofort und bedingungslos die Ausführung des Programms abbricht.

In beiden Fällen wird der übergebene Parameter als Rückgabewert verwendet. (***Vorsicht***: In MS-DOS werden Rückgabewerte modulo 256 behandelt!)

6.2. Standard-C-Ein-/Ausgabe

6.2.1. Konsol-Ausgabe

Die einfachsten Standard-Funktionen für die Ausgabe auf die Konsole (also auf den Bildschirm) sind

```
int putchar(int c);
```

und

```
int puts(const char *string);
```

Diese Funktionen schreiben ein Zeichen bzw. einen String, gefolgt von einem Zeilenvorschub, auf der Konsole aus.

Für formatierte Ausgaben ist die Funktion

```
int printf(const char *format, ...);
```

vorgesehen. Diese Funktion erlaubt die gleichzeitige Ausgabe einer beliebigen Anzahl von Elementen mit voneinander unabhängigen Typen und unabhängiger Formatierung. Die Formatierung wird durch den als erstes Argument übergebenen String `format` festgelegt. Dieser kann beliebigen Text enthalten, der wörtlich ausgegeben wird; eingebettet in den Text kann eine beliebige Anzahl von *Formatangaben* (*Format Specifiers*) sein, die durch ein Prozentzeichen ("%") eingeleitet werden. Jeder Formatangabe muss genau ein auf `format` folgendes Argument entsprechen; wenn mehr Formatangaben als Argumente vorhanden sind, ist das Ergebnis undefiniert (und kann bis zum Programmabsturz reichen) (siehe auch Seite 189).

Auf das einleitende Prozentzeichen der Formatangabe können die nachstehenden Felder in der angegebenen Reihenfolge folgen:

➡ Steuerzeichen (*Flags*) (optional; mehrere sind zulässig):

- `-` Datenfeld wird linksbündig ausgegeben (Standard ist rechtsbündig)
- `+` Positiven Zahlenwerten wird ein "`+`" vorangestellt.
- *`b1`* ("*`b1`*" = "*blank*"): Bewirkt ein Leerzeichen an Stelle des Vorzeichens in der Ausgabe.
- `#` Alternative Formatierung: Oktalzahlen wird "`0`" vorangestellt, Hexadezimalzahlen "`0x`" oder "`0X`", und Gleitkommazahlen werden immer mit Dezimalpunkt ausgegeben (auch wenn sie ganzzahlig sind).
- `0` Die spezifizierte Breite des Ausgabefeldes wird mit führenden Nullen aufgefüllt.

➡ (Minimale) Breite des Ausgabefeldes (optional):

Wenn die tatsächlich erforderliche Anzahl von Ausgabestellen größer als die angegebene Breite ist, wird die Ausgabe nicht abgeschnitten, sondern in der tatsächlich erforderlichen Anzahl von Stellen vorgenommen. An dieser Stelle kann auch ein "`*`" stehen; dieser bewirkt, dass die Breite des Ausgabefeldes durch das nächste Argument in der Argumentliste vorgegeben wird. Dieses Argument muss dem zu formatierenden Wert in der Argumentliste *vorangehen*.

➡ Dezimalpunkt und Auflösung (optional):

Dieses Feld gibt an für

Ganze Zahlen	Minimale Anzahl auszugebender Stellen; fehlende Stellen werden mit "`0`" aufgefüllt. Standardeinstellung: `1`
Gleitkommazahlen	Anzahl der Stellen nach dem Dezimalpunkt; letzte Stelle wird gerundet. Für "`g`"- oder "`G`"-Format: Maximale Anzahl der signifikanten Stellen. Standardeinstellung: für "`e`"- und "`E`"-Format: `6`, für "`f`": `0`, für "`g`"- oder "`G`"-Format: alle.
Strings	Maximale Zahl der auszugebenden Zeichen (Standardeinstellung: alle).

➡ Format-Attribut (optional):

- `h` Auszugebender Wert ist **`short int`** oder **`short unsigned`**.
- `l` Auszugebender Wert ist **`long int`**, **`long unsigned`** oder (implementierungsspezifisch) **`double`.**
- `L` Wie oben; implementierungsspezifisch bei Gleitkommazahlen: **`long double`**.

➡ Typenangabe (muss vorhanden sein):

- `c` **`char`**-Variable (einzelnes Zeichen).
- `d` Vorzeichenbehaftete ganze Zahl im Dezimalformat.
- `D` Implementierungsspezifisch: **`long int`** als vorzeichenbehaftete ganze Zahl im Dezimalformat.
- `e,E` Gleitkommazahl in wissenschaftlicher Notation; für "`e`" mit "`e`" im Exponenten, für "`E`" mit "`E`".
- `f` Gleitkommazahl ohne Exponenten.
- `g,G` Gleitkommazahl, nur dann in wissenschaftlicher Notation, wenn dies erforderlich ist; für "`g`" mit "`e`" im Exponenten, für "`G`" mit "`E`".
- `i` Vorzeichenbehaftete ganze Zahl im Dezimalformat.
- `n` Zugehöriges Argument ist ein Zeiger auf **`int`**; die Anzahl der bislang geschriebenen Zeichen wird in dieser **`int`**-Variablen gespeichert.
- `o` Nicht vorzeichenbehaftete Oktalzahl.
- `p` Zeiger; wird als Hexadezimalzahl (entsprechend "`x`"-Format) ausgegeben.
- `s` String, mit (ASCII-)Null abgeschlossen.
- `u` Nicht vorzeichenbehaftete ganze Zahl im Dezimalformat.
- `U` Implementierungsabhängig: **`unsigned long`** im Dezimalformat.
- `x,X` Nicht vorzeichenbehaftete ganze Zahl im Hexadezimalformat, für "`x`" in Kleinbuchstaben, für "`X`" in Großbuchstaben.
- `%` Prozentzeichen.

Das Ergebnis von `printf()` ist die Anzahl der mit diesem Funktionsaufruf ausgegebenen Zeichen.

Das folgende Beispiel illustriert die verschiedenen mit `printf` erzielbaren Formatierungen für ganze Zahlen und Gleitkommazahlen. (Die darin verwendeten Eingabe-Funktionen `gets()` und `sscanf()` werden auf Seite 179 und 184 bzw. 181 besprochen.)

Demo-Programm 6_02_01.cc

```
#include <stdio.h>        // enthält Prototypen der Ein-/Ausgabe-Funktionen

int main ()
{
    char buffer[128];                          // Eingabepuffer
    int i;
    double d;

    for ( ; ; )                                // Endlosschleife
    {
        printf ("\nBitte ganze Zahl eingeben; Ende mit Leer-Eingabe: ");

        gets (buffer);                         // Eingabe einlesen
        if (! *buffer)                         // Leer-Eingabe
            break;                             // Schleife verlassen

        sscanf (buffer, "%i", &i);             // konvertieren in int

        printf ("String: \"%s\":\ndezimal: %10i, oktal: %12o, hex: %#010x, "
            "char: %c\n", buffer, i, i, i, i);
    }
```

```
    for ( ; ; )                                 // Endlosschleife
    {
        printf ("\nBitte Gleitkommazahl eingeben; Ende mit Leer-Eingabe: ");

        gets (buffer);                          // Eingabe einlesen
        if (! *buffer)                          // Leer-Eingabe
            break;                              // Schleife verlassen

        sscanf (buffer, "%lf", &d);             // konvertieren in double

        printf ("String: \"%s\":\n\"e\": %16.7e, \"f\": %16.7f, \"g\": "
            "%16.15g\n", buffer, d, d, d);
    }
}
```

Beachten Sie bitte bei diesem Demo-Programm und seiner Ausgabe die folgenden Punkte:

- Das Programm besteht aus zwei Endlosschleifen für die Ein- und Ausgabe von ganzen Zahlen bzw. Gleitkommazahlen. Die beiden Schleifen werden jeweils abgebrochen, wenn die Eingabetaste gedrückt wurde, ohne dass irgendwelche Daten eingegeben wurden.
- Die zweistufige Eingabe (Einlesen eines Strings mit `gets()`, und Umsetzung der in diesem String enthaltenen Information in einen Zahlenwert mit geeignetem Format mit `sscanf()`) erlaubt ein Verhalten des Programms, das den Erwartungen eines Benutzers eher entspricht als eine kombinierte Eingabe und Konversion mit `scanf()` (siehe Seite 184). Insbesondere ist die Fehlerbehandlung (bei Eingabe von Daten in einem unzulässigen Format) bei der gewählten Vorgangsweise unproblematisch; der vorhergehende Wert der Variablen, die die Eingabe aufnehmen sollen, bleibt erhalten.
- Eine gewöhnliche **`int`**-Variable kann auch als Zeichen (mit der Typenangabe "`c`") ausgegeben werden; dies bedeutet jedoch *keine* Konversion eines Zahlenwertes in eine Ziffer, sondern einfach eine andere Interpretation.
- Eine Breiten-Angabe für die Ausgabentype "`c`" bedeutet *nicht*, dass das betreffende Zeichen mehrfach wiederholt wird, sondern nur, dass ihm `(Breite-1)` Leerzeichen vorangestellt werden.
- Das Format "`g`" oder "`G`" für die Ausgabe von Gleitkommazahlen passt sich dem auszugebenden Zahlenwert insofern an, als dort, wo dies leicht möglich ist, normale Gleitkomma-Notation und ansonsten wissenschaftliche Notation verwendet wird. Im Gegensatz zu den übrigen Gleitkomma-Formaten gibt der Auflösungs-Parameter *nicht* die Anzahl der Dezimalstellen, sondern die der *gesamten* ausgegebenen Stellen an. Während bei den anderen Gleitkomma-Formaten der Wert für die Auflösung deutlich kleiner sein sollte als für die Breite des Ausgabefeldes ("`%16.7e`", "`%16.7f`"), um eine gleichmäßige Spaltenbreite sicherzustellen, können beim "`g`"-Format die beiden Werte nahe beisammen liegen ("`%16.15g`").
- Der Format-String kann beliebige Escape-Sequenzen ("`\n`", "`\t`", "`\a`", "`\b`") und auch sonst beliebigen Text enthalten.
- Die Ausgabe muss nicht unbedingt mit einem Zeilenvorschub ("`\n`") abgeschlossen werden, sodass (wie im Beispiel) Eingabe-Prompts möglich sind. Allerdings werden Ausgaben mit `putchar()` und `printf()` in manchen Umgebungen (z.B. GNU C/C++) intern gepuffert und erst dann ausgeschrieben, wenn entweder der interne Puffer voll ist, ein Zeilenvorschub ausgegeben wird oder eine Eingabe mit `getchar()` oder `gets()` erfolgt (nicht aber bei `getkey()`!).

GNU-C/C++-spezifisch

6.2.2. Konsol-Eingabe

Die einfachsten Standard-Funktionen für die Eingabe von der Konsole (also von der Tastatur) sind

```
int getchar(void);
```

und

```
char *gets(char *buffer);
```

Diese Funktionen lesen ein Zeichen bzw. einen String von der Konsole ein. In beiden Fällen ist ein Editieren der Eingabe möglich, bevor die Eingabedaten an das Programm weitergegeben werden.

Bei beiden Funktionen verweilt das Programm so lange im Funktionsaufruf, bis die Eingabezeile durch Drücken der Eingabetaste abgeschlossen wurde. Während `gets()` einen Zeiger auf den ihm als Argument übergebenen Puffer zurückgibt, in dem die gesamte Eingabezeile abgespeichert wurde, liefert `getchar()` ein Zeichen nach dem anderen aus dem Eingabestring. Diese beiden Funktionen sind implementierungsunabhängig auf allen Plattformen verfügbar; unter MS-DOS benützen sie die entsprechenden Funktionen von MS-DOS (d.h., alle von MS-DOS vorgesehenen Vorverarbeitungen der Tastatureingabe werden auch bei `getchar()` und `gets()` vorgenommen.)

Zusätzlich bieten die Bibliotheken der meisten Compiler hardwarenähere Eingabefunktionen, die in der MS-DOS-Umgebung in der Regel auf Funktionen des BIOS aufsetzen und "rohe" Eingabedaten, teils ohne Echo auf den Bildschirm, liefern. Die Namen dieser Funktionen sind relativ implementierungsabhängig; Beispiele sind:

```
int getkey(void);          und
int getxkey(void);         (GNU C/C++)

int getch(void);           (Microsoft)
```

Diese Funktionen warten grundsätzlich auf die Eingabe eines Zeichens; das Echo des eingegebenen Zeichens auf den Bildschirm unterbleibt. `getkey()` und `getxkey()` unterscheiden sich durch ihre Behandlung der Steuertasten des PC (wie etwa der Funktionstasten oder der Tasten des Cursorblocks). Für diese wird statt des ASCII-Wertes der Taste der *Scan Code*, ODER-verknüpft mit `0x100` oder `0x200`, zurückgegeben. (Bei Zuweisung der Tastatureingabe an eine Variable vom Typ **`char`** geht diese Information wieder verloren; das Ergebnis ist dann oft unerwartet.)

Das folgende Demo-Programm vergleicht die einfachen Eingabefunktionen:

Demo-Programm 6_02_02.cc

```
#include <stdio.h>
#include <pc.h>                                  // für getkey()

int main ()
{
    int ch;
    char buf[128];

    printf ("\nEingabe mit gets():  ");

    char *bptr = gets (buf);
    if (! bptr)
        printf ("NULL-Zeiger von gets() erhalten.\n");
    else
        if (! *bptr)
            printf ("Leerer String von gets() erhalten.\n");
        else
            printf ("Eingabe mit gets(): \"%s\"\n", bptr);

    printf ("\nEingabe mit getchar() (Ende mit Eingabetaste):  ");

    do
    {
        ch = getchar();

        printf ("\neingegeben: %c (0x%02x)", ch, ch);
    }
    while (ch != '\n');

    printf ("\nEingabe mit getkey() (Ende mit Eingabetaste):\n");

    for ( ; ; )                                  // Endlosschleife
    {
        if ((ch = getkey()) == '\r')
            break;
        printf ("eingegeben: %c (0x%04x)\n", ch, ch);
    }
}
```

Beachten Sie bitte, dass bei einer Eingabe mit `getchar()` der Code der Eingabetaste in "`\n`" (`0x0a`) umgesetzt wird, während `getkey()` oder `getxkey()` den "rohen" Code "`\r`" (`0x0d`)

zurückgibt. Die Ergebnisse von `getchar()`, `getkey()` und `getxkey()` sind zwar vom Typ **`int`**, sie könnten aber ebenso gut einer **`char`**-Variablen zugewiesen werden, wobei allerdings die Information bei `getkey()` und `getxkey()` verlorenginge, ob das Zeichen als ASCII-Code oder als *Scan Code* zu verstehen ist. Ein weiterer Nachteil der Verwendung einer **`char`**-Variablen würde sich bei der hexadezimalen Ausgabe von Codes > `0x7f` zeigen, die als negative Zahlen interpretiert und bei der Übergabe an `printf()` in eine negative **`int`** umgesetzt würden. Damit würde beispielsweise für den IBM-ASCII-Code von "ä" nicht "`0x84`", sondern "`0xffffff84`" ausgegeben.

Um zu testen, ob überhaupt eine Eingabe an der Konsole vorliegt, kann die Funktion `kbhit()` verwendet werden, die sofort wieder zurückkehrt und nur dann einen von Null verschiedenen Wert zurückgibt, wenn eine Taste gedrückt wurde:

```
int kbhit(void);
```

Damit sind Eingaben im "Hintergrund" eines Programms möglich, wie das folgende einfache Beispiel zeigt:

Demo-Programm 6_02_03.cc

```
#include <stdio.h>
#include <pc.h>

int main ()
{
    int ch;

    printf ("Abbruch mit Eingabetaste!\n");

    for ( ; ; )
    {
        puts ("*****");

        if (kbhit())                          // Taste gedrückt
        {
            if ((ch = getxkey()) == '\r') break;

            printf ("< Taste \"%c\" (0x%04x) gedrückt >\n", ch, ch);
        }

        int j = 0;

        for (int i = 0; i < 1000000; i++)     // Verzögerungsschleife
            j += i;
    }
}
```

Das Demo-Programm durchläuft eine Endlosschleife, in der bei jedem Durchlauf eine Reihe von Sternen ausgegeben und der Tastaturpuffer abgefragt wird. Die Verzögerungsschleife dient nur dazu, die Geschwindigkeit des Programms auf ein vernünftiges Maß zu reduzieren.

Für formatierte Eingaben (also für die Umsetzung der Eingaben in Zahlenwerte von geeignetem Typ oder Zeichen) ist die Funktion

```
int scanf (const char *format, ...);
```

vorgesehen. Diese Funktion erlaubt die gleichzeitige Eingabe einer beliebigen Anzahl von Elementen mit voneinander unabhängigen Typen und unabhängiger Formatierung in einer Eingabezeile. Die Interpretation der Eingabezeile wird durch den als erstes Argument übergebenen String `format` festgelegt. Dieser enthält üblicherweise eine oder mehrere *Formatangaben* (*Format Specifiers*), die durch ein Prozentzeichen ("%") eingeleitet werden. Jeder Formatangabe muss genau ein auf `format` folgendes Argument entsprechen, das grundsätzlich ein *Zeiger* auf eine Variable mit dem entsprechenden Typ sein muss; wenn mehr Formatangaben als Argumente vorhanden sind oder die Typen der Argument-Zeiger nicht mit den durch die Formatangaben festgelegten Typen übereinstimmen, können schwerwiegende Fehler bis zu einem Programmabsturz auftreten (siehe Seite 190). Der Format-String kann zusätzlich zu den Formatangaben auch beliebigen Text enthalten; dieser wird bei der Interpretation der Eingabedaten folgendermaßen interpretiert:

- *White Spaces* (Leerzeichen, Tabulatoren, Zeilenvorschübe) im Format-String entsprechen einer beliebigen Anzahl (einschließlich Null) von beliebigen *White Spaces* in der Eingabe.

- Alle Konversionen (außer "`c`" und "`[ ]`") überlesen führende *White Spaces* in der Eingabe.
- Alle gewöhnlichen Zeichen im Format-String müssen exakt in gleicher Weise und an der gleichen Stelle auch in der Eingabe vorkommen. Ist dies nicht der Fall, wird `scanf()` abgebrochen; ein nachfolgender Aufruf von `scanf()` setzt an der Stelle fort, wo die Eingabe abgebrochen wurde. (***Vorsicht***: Wenn die Eingabe in einer Schleife immer mit demselben Format erfolgt, dann führt ein falsches Zeichen in den Eingabedaten effektiv zu einem Blockieren des Programms!)

Auf das einleitende Prozentzeichen einer Formatangabe können die nachstehenden Felder in der angegebenen Reihenfolge folgen:

➡ Ein Stern ("*") (optional):

Das dieser Formatangabe entsprechende Eingabefeld wird entsprechend der Formatangabe gelesen und interpretiert, aber sein Ergebnis keiner Variablen zugewiesen. (Das Feld wird also überlesen.)

➡ Maximale Breite des Eingabefeldes (optional):

Dieser Parameter gibt an, wie viele Zeichen in der Eingabe maximal dem aktuellen Eingabefeld zugeordnet werden. Das aktuelle Eingabefeld endet ansonsten beim ersten *White Space* oder beim ersten Zeichen, das für die gewünschte Konversion nicht zulässig ist.

➡ Format-Attribut (optional):

`h` Die Variable, der der eingelesene Wert zugewiesen werden soll, ist **`short int`** oder **`short unsigned`**.

`l` Die Variable, der der eingelesene Wert zugewiesen werden soll, ist **`long int`**, **`long unsigned`** oder **`double.`**

➡ Typenangabe (muss vorhanden sein):

`c` Das zugehörige Argument ist vom Typ **`char*`**. Das nächste Zeichen in der Eingabe (oder bei Angabe einer Breite die entsprechende Anzahl von Zeichen) werden in den Puffer kopiert, auf den das Argument zeigt.

`d` Die Eingabe wird als vorzeichenbehaftete ganze Zahl im Dezimalformat interpretiert.

`e,E f g,G` Die Eingabe wird als Gleitkommazahl interpretiert. Das zugehörige Argument muss vom Typ **`float*`** sein.

`i` Die Eingabe wird als vorzeichenbehaftete ganze Dezimal-, Oktal- oder Hexadezimal-Zahl interpretiert.

`n` Das zugehörige Argument ist vom Typ **`int*`**; die Anzahl der bislang gelesenen Zeichen wird in dieser **`int`**-Variablen gespeichert.

`o` Die Eingabe wird als nicht vorzeichenbehaftete Oktal-Zahl interpretiert.

`p` Die Eingabe wird als hexadezimaler Zeiger interpretiert.

`s` Die folgenden regulären Zeichen bis zum nächsten *White Space* werden in den Puffer kopiert und als NULL-terminierter String gespeichert. Das zugehörige Argument muss vom Typ **`char*`** sein.

`u` Die Eingabe wird als nicht vorzeichenbehaftete ganze Zahl im Dezimalformat interpretiert.

`x,X` Die Eingabe wird als Hexadezimal-Zahl interpretiert.

`[ ]` Ähnlich zum "`c`"-Format; es werden jedoch nur jene Zeichen kopiert, die den zwischen den eckigen Klammern stehenden entsprechen (z.B. `[abcd]`).

`%` Entspricht einem erforderlichen Prozentzeichen in der Eingabe.

In manchen (16-Bit-) Implementierungen stehen Großbuchstaben in der Typenbezeichnung für **`long`** statt **`int`** als Zielvariable.

Das Ergebnis von `scanf()` ist die Anzahl der erfolgreich eingelesenen Elemente.

Das folgende Programm illustriert einige dieser Funktionen:

Demo-Programm 6_02_04.cc

```
#include <stdio.h>

int main ()
{
    int i;
    double d;
    char buf[80], buf1[80];

    printf ("Eingabe: Integer Double # String; Ende mit Integer == 0\n\n");

    for ( ; ; )
    {
        int j = scanf ("%i %lf # %s", &i, &d, buf);

        if (j < 3)                              // Eingabefehler
        {
            scanf ("%s", buf1);                 // Eingabepuffer ausleeren
            printf ("FEHLER bei \"%s\"\n", buf1);
        }

        printf ("\n%i Elemente: Int = %i; Double = %g;\nString = \"%s\"\n\n",
            j, i, d, buf);

        if (! i)                                // Abbruchbedingung
            break;
    }
}
```

Der folgende Dialog mit dem Demo-Programm soll einige Eigenschaften von `scanf()` illustrieren. Die Eingaben über die Tastatur sind in Fettdruck wiedergegeben:

```
Eingabe: Integer Double # String; Ende mit Integer == 0

12345    987.65432      # Hello, world!
3 Elemente: Int = 12345; Double = 987.654;
String = "Hello,"

FEHLER bei "world!"
0 Elemente: Int = 12345; Double = 987.654;
String = "Hello,"

0 1 2
FEHLER bei "2"
2 Elemente: Int = 0; Double = 1;
String = "Hello,"
```

Der erste der beiden `scanf()`-Befehle erwartet eine ganze und eine Gleitkomma-Zahl, ein "`#`" und einen String. Beim ersten Aufruf bearbeitet er die beiden eingegebenen Zahlenwerte, das "`#`" und das Wort "`Hello,`" beendet dann aber seine Konversion bei dem Leerzeichen vor "`world!`". Beim nächsten Aufruf versucht `scanf()` dort fortzusetzen, wo die letzte Leseoperation endete; "`world!`" ist aber kein gültiger Wert für eine Zahl, sodass `scanf()` abgebrochen wird. Das Ergebnis von `scanf()` ist in diesem Fall `0`; die bereits gelesenen Werte werden nicht verändert, und das Programm verzweigt in die "Fehler-Sequenz", die einen String (in diesem Fall "`world!`") aus dem Eingabepuffer liest und ausgibt. Bei der nächsten Eingabe findet `scanf()` zwei gültige Zahlenwerte, aber kein "`#`" vor, und bricht daher vorzeitig (mit nur 2 gelesenen Elementen und einer Fehlermeldung) ab.

Der grundsätzliche Nachteil einer derartigen Verwendung von `scanf()` liegt darin, dass die Eingabe ein hohes Maß an Disziplin vom Benützer erfordert. Falsche Eingaben müssen entweder (wie im Beispiel) zusätzlich abgefangen werden; anderenfalls können sie zu einem "Blockieren" des Programms führen, weil die fehlerhaften Daten nie aus dem Eingabepuffer entfernt werden. Weiters ist mit `scanf()` eine Eingabe von Strings, die aus mehreren Worten bestehen (also *White Spaces* enthalten) nicht praktikabel: Die Format-Type "`%s`" liest immer nur ein Wort; mit "`%`*n*`c`" können zwar auch die trennenden *White Spaces* mit eingelesen werden, `scanf()` erwartet aber immer genau *n* Zeichen in der Eingabe (die sich dann unter Umständen auch über mehrere Zeilen erstrecken kann). Im Gegensatz dazu hat sich für die formatierte Eingabe die Verwendung der Funktionen `gets()` und `sscanf()` (siehe Seite 184) weit besser bewährt:

- `gets()` liest immer genau eine Zeile ein, die bei Auftreten eines Fehlers implizit verworfen werden kann.
- `sscanf()` wird daher nicht wie `scanf()` immer wieder mit der selben nicht mehr korrigierbaren Eingabe konfrontiert.

Noch zweckmäßiger ist ein Einlesen der gesamten Eingabezeile mit `gets()` und ein Aufbrechen dieser Zeile in ihre Elemente, die dann gezielt einfachen Konversionsfunktionen zugeführt werden können.

6.2.3. Ein-/Ausgabe von/auf Puffer

Eine formatierte Ausgabe *in* einen Puffer (also in ein Feld von **char**s) und eine formatierte Dateneingabe *aus* einem Puffer ist in den folgenden Fällen zweckmäßig oder unumgänglich:

- Ausgaben erfolgen nicht unter Verwendung des gewöhnlichen sequentiellen Ausgabekanals des Rechners, sondern mit speziellen Funktionen (insbesondere in einer semi-graphischen oder graphischen Umgebung).
- Formatierte Ausgabedaten sollen in gleicher Weise auf den Bildschirm und in irgendeinen anderen Ausgabekanal (Datei, Drucker) geschrieben werden.
- Die Eingabe soll benutzerfreundlicher und stabiler gestaltet werden, als dies unter Verwendung von `scanf()` möglich wäre.

Für diese Anwendungen stehen zwei komplementäre Funktionen zur Verfügung:

```
int sprintf(char *buffer, const char *format, ...);
```

für die Ausgabe;

```
int sscanf(const char *string, const char *format, ...);
```

für die Eingabe.

Die Behandlung der im String `format` übergebenen Formatierungsanweisungen erfolgt exakt wie bei `printf()` bzw. `scanf()`; Ausgaben werden aber als NULL-terminierter String in `buffer` geschrieben, und als Quelle für die Eingabedaten wird statt des Konsol-Eingabepuffers der NULL-terminierte String in `string` verwendet.

Die Eingabe mittels `gets()` und `sscanf()` wird im folgenden Beispiel illustriert, das auf dem Beispiel von Seite 183 aufbaut:

Demo-Programm 6_02_05.cc

```
#include <stdio.h>

int main ()
{
    int i;
    double d;
    char buf[80], buf1[80];

    printf ("Eingabe: Integer Double # String; Ende mit Integer == 0\n\n");

    for ( ; ; )
    {
        gets (buf);                             // Zeile von Konsole einlesen

        int j = sscanf (buf, "%i %lf # %s", &i, &d, buf1);
                                                // Zeile konvertieren
        if (j < 3)                              // Eingabefehler
            printf ("FEHLER bei \"%s\"\n", buf);

        printf ("%i Elemente: Int = %i; Double = %g;\nString = \"%s\"\n\n",
            j, i, d, buf1);

        if (! i)                                // Abbruchbedingung
            break;
    }
}
```

Mit den gleichen Eingabezeilen wie im Beispiel des vorhergegangenen Abschnitts erhält man (Eingaben fett gedruckt):

```
Eingabe: Integer Double # String; Ende mit Integer == 0

12345    987.65432      # Hello, world!
3 Elemente: Int = 12345; Double = 987.654;
String = "Hello,"

0 1 2
FEHLER bei "0 1 2"
2 Elemente: Int = 0; Double = 1;
String = "Hello,"
```

In der ersten Eingabezeile wird "`world!`" einfach ignoriert. Die Prüfung auf fehlerhafte Eingaben (`if (j < 3)`) ist — im Gegensatz zur direkten Eingabe mit `scanf()` — hier nicht funktionell notwendig; sie wurde nur zur Illustration des Verhaltens des Programms vorgesehen. Wesentlich ist auch, dass für eine Eingabezeile genau *ein* Satz von Eingabedaten übergeben wird; hingegen kann `scanf()` aus einer Eingabezeile auch *mehrere* Sätze von Eingabedaten in *mehreren* aufeinanderfolgenden Aufrufen "abholen", was vielfach ein unerwartetes Verhalten eines Programms zur Folge hat.

6.2.4. Ein-/Ausgabe auf Dateien

Für die Eingabe von und die Ausgabe auf Dateien gibt es in Standard-C/C++ (mindestens) zwei Familien von Funktionen:

➡ Funktionen für die ungepufferte Ein-/Ausgabe:

Diese Familie umfasst (unter anderen) die Funktionen `open`, `creat`, `read`, `write`, `lseek`, `tell` und `close`, mit denen Dateien zum Lesen oder Schreiben geöffnet, neu erstellt, gelesen, geschrieben und geschlossen werden können sowie die aktuelle Position innerhalb der Datei gesetzt bzw. bestimmt werden kann. Ausgaben unter Verwendung dieser Funktionen werden grundsätzlich direkt an die entsprechenden Funktionen des Betriebssystems weitergegeben; sie sind dann sehr effizient, wenn größere Datenmengen manipuliert werden sollen (zweckmäßigerweise ein Vielfaches der Blockgröße des Massenspeichers). Dateien werden durch einen **`int`**-Wert (*File Descriptor* oder *File Handle*) identifiziert, der als Resultat der Funktionen `open` oder `creat` zurückgegeben wird, und der bei allen anderen Funktionen zur Identifikation der Datei als eines der Argumente übergeben werden muss. C/C++ unterscheidet zwischen zwei verschiedenen Modi für die Handhabung von Dateien:

- Binär-Modus: Im Binär-Modus wird der Inhalt einer Datei unverändert ausgelesen bzw. der Inhalt eines Ausgabepuffers unverändert in die Datei geschrieben.
- Text-Modus: In diesem Modus werden die folgenden Umsetzungen vorgenommen:
  * Beim Lesen einer Datei werden *Carriage Return–Line Feed*-Sequenzen ("`\r\n`") durch einfache Zeilenvorschübe ("`\n`") ersetzt.
  * Beim Schreiben einer Datei werden Zeilenvorschübe ("`\n`") durch *Carriage Return–Line Feed*-Sequenzen ("`\r\n`") ersetzt.
  * *Ctrl-Z* (`0x1a`) wird als Dateiende-Markierung betrachtet und beim Lesen entfernt.

Standardmäßig werden alle Dateien im Text-Modus behandelt; eine Behandlung im Binär-Modus kann durch ein Argument von `open` oder durch Setzen der globalen Variablen `_fmode` auf eine von zwei vordefinierten Konstanten erreicht werden.

Die ungepufferten Datei-Ein-/Ausgabefunktionen sind *nicht* mehr Teil der ANSI-C-Norm, werden aber aus Kompatibilitätsgründen weiter unterstützt. Vielfach wird ihren Namen ein "`_`" vorangesetzt, um sie als Nicht-ANSI-Standard-Funktionen zu kennzeichnen (z.B. `_open`).

➡ Funktionen für die gepufferte Ein-/Ausgabe:

Die Namen dieser Funktionen sind (in der Regel) durch ein vorangestelltes "f" gekennzeichnet (z.B. `fopen`, `fread`, `fgets`, `fgetc`, `fscanf`, `fwrite`, `fputs`, `fputc`, `fprintf`, `fseek`, `ftell`, `fclose`,…). Sie verwenden grundsätzlich einen C-internen Puffer zur Zwischenspeicherung

bei Ein- und Ausgabeoperationen, was sie bei der Manipulation kleinerer Datenmengen nominell effizienter macht als die ungepufferten Funktionen. (Da auf jedem einigermaßen sinnvoll aufgesetzten Rechner eine viel wirkungsvollere Pufferung, z.B. durch das Betriebssystem oder SMARTDRV, erfolgt, als sie die C-Funktionen in ihrer Standardkonfiguration ermöglichen, kann von einer Steigerung der Effizienz kaum die Rede sein; die Mehrfach-Pufferung ist oft sogar kontraproduktiv.) Jedenfalls steht eine weit größere Palette an Funktionen für die gepufferte Ein-/Ausgabe zur Verfügung als für die ungepufferte.

Dateien werden in den gepufferten Ein-Ausgabe-Funktionen grundsätzlich über einen Zeiger vom Typ `FILE*` identifiziert (also einen Zeiger auf ein Objekt, das Informationen über die Datei enthält). Ebenso wie bei den Funktionen für die ungepufferte Ein-/Ausgabe können auch hier Dateien im Binär- oder Textmodus behandelt werden; die Auswahl zwischen den Moden erfolgt beim Öffnen einer Datei oder mit der globalen Variablen `_fmode`.

Fünf Standard-Ein-/Ausgabe-Kanäle (*Streams*) sind vordefiniert und können über die folgenden vordefinierten Dateizeiger angesprochen werden:

`stdin`	Standard-Eingabekanal (i.a. Tastatur)
`stdout`	Standard-Ausgabekanal (i.a. Bildschirm)
`stderr`	Standard-Ausgabekanal für Fehlermeldungen (i.a. Bildschirm)
`stdaux`	Standard-Hilfskanal (serielle Schnittstelle)
`stdprn`	Standard-Drucker

Die Zuordnung dieser Standard-Kanäle kann auch umdefiniert werden (analog zu den Befehlszeilen-Operatoren "<", ">" und "|" von UNIX oder MS-DOS).

Die Familie der Funktionen für die gepufferte Ein- und Ausgabe umfasst auch Funktionen für die formatierte Eingabe (`fscanf`) und Ausgabe (`fprintf`).

Eine gemischte Verwendung von Funktionen für die gepufferte und ungepufferte Ein-/ Ausgabe ist zwar (theoretisch) möglich, wird aber nicht empfohlen.

Die folgenden beiden Beispiele vergleichen die Verwendung der Funktionen für die ungepufferte bzw. die gepufferte Standard-C-Ein-/Ausgabe am Beispiel eines einfachen Telefonregister-Programms:

Demo-Programm 6_02_06.cc

```
#include <stdio.h>
#include <string.h>
#include <osfcn.h>                              // für ungepufferten I/O
#include <fcntl.h>                              // für ungepufferten I/O
#include <sys/stat.h>                           // für ungepufferten I/O

class tellist
{
public:

    tellist () { *_name = 0; *_nummer = 0; }  // Default-Konstruktor

    tellist (char *Name, char *Nummer)
    {
        strncpy (_name, Name, 40);
        _name[39] = 0;
        strncpy (_nummer, Nummer, 20);
        _nummer[19] = 0;
    }
    void disp ()
        { printf ("Name: %-40s: Nummer: %-20s\n", _name, _nummer); }

    static int zahl;                            // Zahl der Eintragungen

private:
    char _name[40], _nummer[20];
};
```

```
// Funktionen für die ungepufferte Ein-/Ausgabe
void lesen (tellist *liste, int max_zahl)
{
    int fd;                             // File Descriptor
    int bytes_rd = 0;                   // Anzahl der gelesenen Bytes

    if ((fd = open ("TELLIST.DAT", O_RDONLY | O_BINARY)) > 0)
    {                                   // erfolgreich geöffnet
        bytes_rd = read (fd, liste, max_zahl * sizeof(tellist));
        close (fd);
    }
    tellist::zahl = bytes_rd / sizeof (tellist);
    printf ("%i Eintragungen gelesen.\n", tellist::zahl);
}

void schreiben (tellist *liste)
{
    int fd;                             // File Descriptor
    int bytes_wr = 0;                   // Anzahl der geschriebenen Bytes

    if ((fd = open ("TELLIST.DAT", O_WRONLY | O_CREAT | O_BINARY,
        S_IREAD | S_IWRITE)) > 0)       // erfolgreich geöffnet
    {
        bytes_wr = write (fd, liste, tellist::zahl * sizeof(tellist));
        close (fd);
    }
    printf ("%i Eintragungen geschrieben.\n", bytes_wr / sizeof (tellist));
}

int tellist::zahl = 0;                  // statisches Element
const int max_zahl = 20;                // maximale Zahl der Einträge
tellist liste[max_zahl];

int main ()
{
    lesen (liste, max_zahl);            // Liste einlesen

    for (int i = 0; i < tellist::zahl; i++)
        liste[i].disp();                // Liste ausschreiben

    printf ("\nNeue Eintragungen eingeben; Ende mit leerem Namensfeld:\n");

    while (tellist::zahl < max_zahl)
    {
        char buffer1[128], buffer2[128];

        printf ("Name:    ");
        gets (buffer1);
        if (! *buffer1)                 // leerer String
            break;
        printf ("Nummer:  ");
        gets (buffer2);
        if (! *buffer2)                 // leerer String
            break;

        liste[tellist::zahl++] = tellist (buffer1, buffer2);
    }

    for (i = 0; i < tellist::zahl; i++)
        liste[i].disp();                // Liste ausschreiben

    schreiben (liste);                  // Liste in Datei schreiben
}
```

In der obigen Version verwendet das Demo-Programm die Funktionen für die ungepufferte Ein-/Ausgabe. Es wird zunächst eine Datenstruktur (`class tellist`) definiert, die neben einem Default- und einem Standard-Konstruktor auch eine Ausgabefunktion für Namen und Telefonnummer sowie eine statische Variable (`zahl`) enthält, die die Anzahl der (aktiven) Einträge angibt. Die Daten selbst werden einfachheitshalber als Strings abgelegt (`_name`, `_nummer`). Die Telefonliste soll mit der Funktion `schreiben()` in eine Datei `tellist.dat` geschrieben und mit `lesen()` von dort wieder eingelesen werden können. Dazu muss in beiden Fällen die Datei durch Aufruf von `open()` zunächst geöffnet werden; die *manifesten Konstanten* im Aufruf von `open()` geben an, ob es sich um einen Zugriff zum Lesen oder einen solchen zum Schreiben handelt, und ob die Datei als Text- oder Binärdatei (wie in unserem Fall) interpretiert werden soll. Der Aufruf von `open()` in `schreiben()` erlaubt, je nachdem, ob die Ausgabedatei bereits existiert oder nicht, diese Datei zu überschreiben oder neu zu erstellen; die zusätzlichen Parameter `S_IREAD | S_IWRITE` legen fest, dass eine neu erstellte Datei gelesen *und* überschrieben werden kann.

Beachten Sie, dass die Schalt-Parameter für `open()` durch binäre "ODER"-Operationen ("`|`") miteinander verknüpft sind!

Die Funktionen `read()` bzw. `write()` verwenden den von `open()` zurückgegebenen *File Descriptor* `fd` als ihr erstes Argument. Das zweite Argument zeigt auf einen Puffer (in unserem Fall das Feld von Listeneinträgen, `liste`); das dritte Argument gibt die Anzahl von Bytes an, die gelesen bzw. geschrieben werden sollen. Nach Gebrauch ist eine Datei grundsätzlich (mit `close()`) zu schließen, da nur eine begrenzte Anzahl von Dateien simultan geöffnet sein dürfen.

Die Funktion `main()` liest die Datei `tellist.dat` ein (soweit sie bereits existiert) und schreibt ihren Inhalt aus. Anschließend können den bestehenden Einträgen neue hinzugefügt werden; die statische Zählvariable `tellist::zahl` wird als Index auf das letzte Element im Feld `liste` verwendet und inkrementiert. Das aktualisierte Feld wird nochmals am Bildschirm ausgegeben und schließlich in die Datei `tellist.dat` geschrieben.

Bei Verwendung der Standard-C-Funktionen für die *gepufferte* Ein-/Ausgabe brauchen nur die Funktionen `lesen()` und `schreiben()` des Demo-Programms geändert zu werden. Weiters sind die *Include*-Dateien `osfcn.h`, `fcntl.h` und `sys/stat.h` nicht mehr erforderlich; die Deklarationen der Funktionen für die gepufferte Ein-/Ausgabe sind in `stdio.h` enthalten.

Demo-Programm 6_02_07.cc

```
// Funktionen für die gepufferte Ein-/Ausgabe
void lesen (tellist *liste, int max_zahl)
{
    FILE *file;                         // File Pointer
    int rec_rd = 0;                     // Anzahl der gelesenen Sätze

    if (file = fopen ("TELLIST.DAT", "rb"))
    {                                   // erfolgreich geöffnet
        rec_rd = fread (liste, sizeof(tellist), max_zahl, file);
        fclose (file);
    }

    tellist::zahl = rec_rd;
    printf ("%i Eintragungen gelesen.\n", tellist::zahl);
}

void schreiben (tellist *liste)
{
    FILE *file;                         // File Pointer
    int rec_wr = 0;                     // Anzahl der geschriebenen Sätze

    if (file = fopen ("TELLIST.DAT", "wb"))
    {                                   // erfolgreich geöffnet
        rec_wr = fwrite (liste, sizeof(tellist), tellist::zahl, file);
        fclose (file);
    }

    printf ("%i Eintragungen geschrieben.\n", rec_wr);
}
```

Wieder muss wie im Beispiel von Seite 186 die Datei `tellist.dat` zum Lesen bzw. Schreiben geöffnet werden; die Funktion `fopen()` verwendet einen String ("`rb`" oder "`wb`") statt manifester Konstanten zur Qualifikation der Datei. Ihr Ergebnis ist nicht ein ganzzahliger Wert, sondern ein Zeiger vom (in `stdio.h` definierten) Typ `FILE*`. Die Funktionen `fread()` und `fwrite()` benöti-

gen als Argument einerseits die Größe der zu lesenden Einheiten (in unserem Fall Objekte der Type `tellist`), und andererseits die Anzahl dieser Einheiten; ihr Ergebnis ist die Anzahl der gelesenen bzw. geschriebenen *Objekte* und nicht die der gelesenen bzw. geschriebenen Bytes. Da die Datenstruktur die gleiche ist wie im vorangegangenen Beispiel, und in beiden Fällen in der "Datenbank-Datei" `tellist.dat` eine Kopie der aktiven Elemente des Feldes von `tellist`-Objekten steht, ist die Datei `tellist.dat` mit beiden Versionen des Programms (und auch mit der auf Seite 201 vorgestellten Variante unter Verwendung der C++-*Stream*-Ein-/Ausgabefunktionen) kompatibel.

6.2.5. Einschränkungen bei Ein-/Ausgabe mit Standard-C-Funktionen

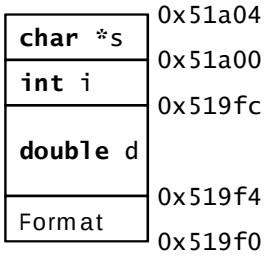
Generell ist bei den Funktionen `printf()`, `scanf()`, `fprintf()`, `fscanf()`, `sprintf()` und `sscanf()` für die formatierte Ein- und Ausgabe keine Typenprüfung der für die Ein- oder Ausgabe übergebenen Argumente und insbesondere keine Prüfung der Korrelation zwischen den Typen und der Anzahl dieser Argumente und den zugehörigen Format-Typen möglich. Der Format-String dieser Funktionen wird (wie jeder andere String auch) vom Compiler *nicht* interpretiert und nur (allenfalls nach Vornahme einfacher Code-Umsetzungen) als Stringkonstante abgelegt. Eine Interpretation des Formats erfolgt erst bei der Ausführung des Programms durch die Ein- oder Ausgabefunktionen.

Das folgende Beispiel illustriert die möglichen Konsequenzen einer Nicht-Übereinstimmung zwischen Format und Argumenten bei `printf()`:

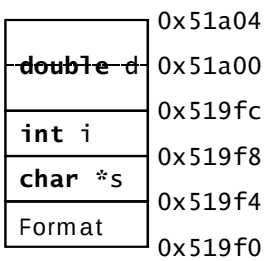
```
void outfunc (int i, double d, char *s)
{
    printf ("String s = %s; Int i = %i, Double d = %lf", d, i, s);
}
```

In diesem Beispiel sind im Format bzw. in der Argumentliste von `printf()` der String und die **`double`**-Variable vertauscht worden. Die Argumente des Aufrufs von `printf()` werden (beispielsweise vom GNU C++-Compiler) entsprechend der Argumentliste folgendermaßen am Stack übergeben:

GNU-C/C++-spezifisch



Diese Argumente werden nun aber von `printf()` entsprechend den im Format-String vorhandenen Typenangaben interpretiert. Ausgegeben wird also:



Es werden also die niederwertigen vier Bytes des **`double`**-Arguments als Zeiger auf einen String und seine höherwertigen vier Bytes als ganze Zahl interpretiert; aus dem Zeiger und dem **`int`**-Argument wird eine Gleitkommazahl "gebaut". Während die ausgegebenen Zahlenwerte "nur" sinnlos sind, kann — je nach Implementierung — der Speicherzugriff über einen "falschen" Zeiger zu fatalen Fehlern (*General Protection Faults*) führen und hat zumindest in der Regel die Ausgabe von "Mist" zur Folge.

Die Argumente von `printf()` und den verwandten Ausgabefunktionen werden grundsätzlich den folgenden automatischen Typenumwandlungen unterworfen:

- **`float`** → **`double`**;
- **`char`**, **`short`**, Aufzählungen, Bitfelder → **`int`** oder **`unsigned int`**;
- Klassen-Objekte (**`struct`**, **`union`**, **`class`**) werden binär kopiert als Datenstrukturen übergeben.

Noch drastischer können die Konsequenzen sein, wenn Anzahl und Typen der Argumente eines Aufrufs von `scanf()`, `fscanf()` oder `sscanf()` nicht mit den Angaben im Format-String übereinstimmen. Insbesondere kann keine Warnung erfolgen, wenn als Argument ein *Objekt* statt eines *Zeigers* übergeben wurde. Die Eingabefunktionen interpretieren den (möglicherweise nicht einmal definierten) Wert des Objekts als Adresse und überschreiben mit den Eingabedaten, was immer an dieser Adresse steht. (Bei 16-Bit-8086-Programmen sind dies in der Regel "nur" Daten; im 32-Bit-Modus des GNU-C/C++-Compilers kann das auch Programmcode sein.) Ebenso fatal sind die Folgen, wenn der Format-String mehr Eingabefelder enthält als Argumente übergeben wurden: In diesem Fall wird der Inhalt des Stacks als Adressen interpretiert. Bedingt durch die Struktur des Stacks wird es sehr wahrscheinlich, dass Programm-Code und/oder vitale Daten dabei beschädigt werden.

GNU-C/C++-spezifisch

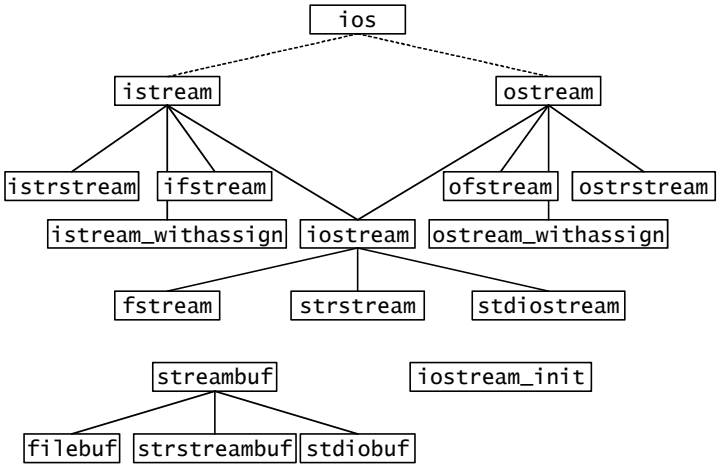
6.3. Die C++ *Class Library* `iostream`

C++-spezifisch

6.3.1. Allgemeines

Die *Class Library* `iostream` besteht aus einer Reihe standardmäßig definierter Klassen, die zum größten Teil von der Basisklasse `ios` virtuell abgeleitet sind. (`iostream` selbst ist nur eine dieser abgeleiteten Klassen.)

Die Beziehungen dieser Klassen zueinander kann graphisch folgendermaßen dargestellt werden:



Die Klasse `istream` und die von ihr abgeleiteten Klassen beziehen sich auf Eingabe-*Streams* (also Kanäle im weitesten Sinn, die zur Eingabe von Daten dienen); `ostream` und ihre abgeleiteten Klassen beziehen sich auf Ausgabe-*Streams*. Die Klassen `ifstream` und `ofstream` beschreiben die Eingabe von bzw. die Ausgabe auf Dateien; die Klassen `istrstream` und `ostrstream` die Eingabe von bzw. die Ausgabe auf einen Puffer, und die Klassen `istream_withassign` und `ostream_withassign` die Eingabe von der bzw. die Ausgabe auf die Konsole, die bei Bedarf auch auf eine andere Quelle umgelegt werden kann (daher "`withassign`"). `cin` ist ein vordefiniertes *Objekt* (keine Funktion!) von `istream_withassign`, und `cout` (und die im folgenden Abschnitt beschriebenen verwandten Objekte) ein solches von `ostream_withassign`. `iostream` ist eine Klasse für Ein- *und* Ausgabeoperationen; die davon abgeleiteten Klassen beziehen sich wieder auf Dateien (`fstream`), String-Puffer (`strstream`) sowie auf Ein-/Ausgabeoperationen, die mit den Standard-C-Ein-/Ausgabefunktionen (in `stdio.h`) kompatibel sind (`stdiostream`).

Die Klasse `streambuf` beschreibt allgemeine *gepufferte* Ein-/Ausgabefunktionen; ihre abgeleiteten Klassen beschreiben die Pufferung von Dateien (`filebuf`), Strings (`strstreambuf`) und von Ein-/Ausgaben über das Standard-C-Ein-/Ausgabesystem (`stdiobuf`).

`iostream_init` dient zur Initialisierung der Konsol-Ein-/Ausgabeobjekte `cin`, `cout`, `cerr` und `clog`. Diese Klasse wird nur intern von den C++-Bibliotheksfunktionen verwendet.

6.3.2. Konsol-Ausgabe

Eine Ausgabe auf die Konsole unter Verwendung der Funktionen der Klasse `iostream` kann mittels der folgenden *Objekte* erfolgen:

`cout` Standard-Ausgabe (entspricht `stdout`).

`cerr` Ausgabekanal für Fehlermeldungen (entspricht `stderr`) mit begrenzter Pufferung.

`clog` wie `cerr`, aber mit voller Pufferung.

Andere *Stream*-Objekte (die beispielsweise einen Puffer oder eine Datei beschreiben) können explizit konstruiert werden (siehe Seite 199).

Beachten Sie bitte, dass (zum Beispiel) `cout` ein *Objekt* der Klasse `ostream` ist (eigentlich einer davon abgeleiteten Klasse) und keine *Funktion*!

Eine Ausgabe ist grundsätzlich in formatierter Form möglich. Die Klassenelement-Funktionen der Klasse `ostream` sehen eine Default-Formatierung in Abhängigkeit vom Typ eines für die Ausgabe vorgesehenen Objekts vor (die in unseren Demo-Programmen fast ausschließlich verwendet wurde). (Der auf ein *Stream*-Objekt anzuwendende Operator "`<<`" ist natürlich durch *Operator Overloading* definiert; eine derartige Definition existiert für alle vordefinierten Objekttypen (siehe Datei `iostream.h`).) Die Default-Formatierung kann durch Klassenelement-Funktionen und *Manipulatoren* geändert werden. Die wichtigsten dieser Funktionen und Manipulatoren sind im folgenden Demo-Programm enthalten:

Demo-Programm 6_03_01.cc

```
#include <iostream.h>
#include <iomanip.h>                    // für Manipulatoren

int ai[] = { 3, 17, 123, 4321, 23456 };
double ad[] = { .1234, 43.21, 543.456, 2345.6, 765432.109876 };
char *as[] = { "The", "quick", "brown", "fox", "jumps" };

int main ()
{
    cout << "\nDefault-Formatierung:\n";

    for (int i = 0; i < 5; i++)
        cout << "int: " << ai[i] << ", double: " << ad[i] << ", string: " <<
            as[i] << endl;              // endl == "\n"

    cout << "\nKonstante Spaltenbreite:\n";

    for (i = 0; i < 5; i++)
    {
        cout.width (10);
            // Spaltenbreite = 10; gilt nur für das folgende Feld!
        cout << di[i] << endl;
    }

    for (i = 0; i < 5; i++)
        cout << "int: " << setw(7) << ai[i] << ", double: "<< setw(10) <<
            ad[i] << ", string: " << setw(7) << as[i] << endl;
                // setw gilt nur für das folgende Feld!

    cout << "\nSpalte auffüllen:\n";

    cout.fill ('*');                    // Füllzeichen "*"

    for (i = 0; i < 5; i++)
        cout << "int: " << setw(7) << ai[i] << ", double: "<< setw(10) <<
            ad[i] << ", string: " << setw(7) << as[i] << endl;

    cout.fill (' ');                    // zurück zum Default

    cout << "\nLinksbündige Ausgabe:\n" << setiosflags (ios::left);

    for (i = 0; i < 5; i++)
        cout << "int: " << setw(7) << ai[i] << ", double: " << setw(10) <<
            ad[i] << ", string: " << setw(7) << as[i] << endl;

    cout << resetiosflags (ios::left) << "\nAuflösung: 2 Stellen:\n" <<
        setprecision (2);

    for (i = 0; i < 5; i++)
        cout << "int: " << setw(7) << ai[i] << ", double: " << setw(10) <<
            ad[i] << ", string: " << setw(7) << as[i] << endl;

    cout << "\nExponentenschreibweise:\n" << setiosflags(ios::scientific);
```

```
    for (i = 0; i < 5; i++)
        cout << "int: " << setw(7) << ai[i] << ", double: " << setw(10) <<
            ad[i] << ", string: " << setw(7) << as[i] << endl;

    cout << "\nGleitkommaschreibweise:\n" << resetiosflags (ios::scientific)
        << setiosflags (ios::fixed);

    for (i = 0; i < 5; i++)
        cout << "int: " << setw(7) << ai[i] << ", double: " << setw(10) <<
            ad[i] << ", string: " << setw(7) << as[i] << endl;

    cout << "\nRadix:\n" << setprecision(6);

    for (i = 0; i < 5; i++)
        cout << "Dezimal: " << dec << ai[i] << ", oktal: " << oct << ai[i] <<
            ", hex: " << hex << ai[i] << endl;
}
```

Das Programm erzeugt die folgende Ausgabe:

```
Default-Formatierung:
int: 3, double: 0.1234, string: The
int: 17, double: 43.21, string: quick
int: 123, double: 543.456, string: brown
int: 4321, double: 2345.6, string: fox
int: 23456, double: 765432, string: jumps

Konstante Spaltenbreite:
          3
         17
        123
       4321
      23456

int:       3, double:     0.1234, string:     The
int:      17, double:      43.21, string:   quick
int:     123, double:    543.456, string:   brown
int:    4321, double:     2345.6, string:     fox
int:   23456, double:     765432, string:   jumps

Spalte auffüllen:
int: ******3, double: ****0.1234, string: ****The
int: *****17, double: *****43.21, string: **quick
int: ****123, double: ***543.456, string: **brown
int: ***4321, double: ****2345.6, string: ****fox
int: **23456, double: ****765432, string: **jumps

Linksbündige Ausgabe:
int: 3      , double: 0.1234    , string: The
int: 17     , double: 43.21     , string: quick
int: 123    , double: 543.456   , string: brown
int: 4321   , double: 2345.6    , string: fox
int: 23456  , double: 765432    , string: jumps

Auflösung: 2 Stellen:
int:       3, double:       0.12, string:     The
int:      17, double:         43, string:   quick
int:     123, double:    5.4e+02, string:   brown
int:    4321, double:    2.3e+03, string:     fox
int:   23456, double:    7.7e+05, string:   jumps

Exponentenschreibweise:
int:       3, double:   1.23e-01, string:     The
int:      17, double:   4.32e+01, string:   quick
int:     123, double:   5.43e+02, string:   brown
int:    4321, double:   2.35e+03, string:     fox
int:   23456, double:   7.65e+05, string:   jumps
```

```
Gleitkommaschreibweise:
int:       3, double:       0.12, string:     The
int:      17, double:      43.21, string:   quick
int:     123, double:     543.46, string:   brown
int:    4321, double:    2345.60, string:     fox
int:   23456, double:  765432.11, string:   jumps

Radix:
Dezimal: 3, oktal: 3, hex: 3
Dezimal: 17, oktal: 21, hex: 11
Dezimal: 123, oktal: 173, hex: 7b
Dezimal: 4321, oktal: 10341, hex: 10e1
Dezimal: 23456, oktal: 55640, hex: 5ba0
```

Der Manipulator `flush` bewirkt ein sofortiges Ausschreiben des Ausgabepuffers, der bei gepufferter Konsol-Ausgabe ansonsten erst bei einem Eingabebefehl ausgeschrieben würde (entspricht der Standard-C-Funktion `fflush()`):

Demo-Programm 6_03_02.cc

```
#include <iostream.h>
#include <time.h>

int main ()
{
    time_t endz = time (NULL) + 5;

    cout << "Ausgabe ohne \"flush\":\n";
    cout << "Bitte 5 Sekunden warten...";

    while (time (NULL) < endz);               // warten

    cout << "Wartezeit vorbei.\n";
    endz = time (NULL) + 5;
    cout << "Ausgabe mit \"flush\":\n";
    cout << "Bitte 5 Sekunden warten..." << flush;

    while (time (NULL) < endz);               // warten

    cout << "Wartezeit vorbei.\n";
}
```

Die ersten beiden Zeilen werden hier nicht, wie erwartet, *vor* dem Beginn der Wartezeit ausgeschrieben, sondern erst an ihrem Ende. Im zweiten Teil des Programms stellt der `flush`-Manipulator sicher, dass der Hinweis auf die Wartezeit *vor* ihrem Beginn erfolgt.

Der Operator "`<<`" kann auch für eigene Datenobjekte definiert werden. Das folgende Beispiel zeigt eine Definition für eine Klasse `Punkt`:

Demo-Programm 6_03_03.cc

```
#include <iostream.h>

class Punkt
{
public:
    Punkt(short x = 0, short y = 0) { _x = x; _y = y; }

private:
    short _x, _y;

friend ostream& operator << (ostream&, Punkt&);
};

ostream& operator << (ostream& os, Punkt& p)
{
    os << "x = " << p._x << ", y = " << p._y;
    return os;
}
```

```
int main ()
{
    Punkt pt1 (3, 5);                                   // Objekt
    Punkt *ppt2 = new Punkt (11, 23);                   // Zeiger

    cout << "Punkt 1: " << pt1 << "; Punkt 2: " << *ppt2 << endl;
}
```

Damit kann eine Ausgabe von Objekten der Klasse `Punkt` ebenso einfach erfolgen wie die der vordefinierten Typen. Die Ausgabe des Programms ist:

```
Punkt 1: x = 3, y = 5; Punkt 2: x = 11, y = 23
```

Auch Manipulatoren, die keine Argumente erfordern, können einfach selbst festgelegt werden. Im folgenden Beispiel wird ein Manipulator `sterne` definiert, der eine Folge von fünf Sternen ausgibt:

Demo-Programm 6_03_04.cc

```
#include <iostream.h>

ostream& sterne (ostream& os)
{
    return os << "*****";
}

int main ()
{
    cout << "12345" << sterne << "67890\n";
}
```

Die Ausgabe dieses Programms ist, wie erwartet:

```
12345*****67890
```

Es gibt auch ein Äquivalent zur Standard-C-Ausgabefunktion `putchar()`, nämlich die Element-Funktion `put()`. Im Gegensatz zur formatierten *Stream*-Ausgabe wird die Ausgabe von `put()` aber nicht durch Format-Angaben wie `width()` oder `fill()` bestimmt (was insofern irrelevant ist, weil `width()` nur die erste auf den Funktionsaufruf folgende *numerische* Ausgabe, nicht aber die Ausgabe von Zeichen beeinflusst, wie das folgende Beispiel zeigt):

Demo-Programm 6_03_05.cc

```
#include <iostream.h>

int main ()
{
    cout.width (10);
    cout.fill ('#');

    cout << 'A' << 'B' << 'C' << endl;

    cout.put ('A');
    cout.put ('B');
    cout.put ('C');
    cout.put ('\n');

    cout << 1 << 2 << 3 << endl;
}
```

Die Ausgabe des Programms ist:

```
ABC
ABC
#########123
```

(Beachten Sie bitte, dass die Element-Funktionen `width()` und `fill()` erst (und allein) bei der Ausgabe der Zahl `1` in der letzten Zeile wirksam werden.)

6.3.3. Konsol-Eingabe

Eine Eingabe von der Konsole unter Verwendung der Funktionen der Klasse `iostream` erfolgt zweckmäßigerweise mittels des Objekts `cin`. Andere *Stream*-Objekte (die beispielsweise einen Puffer oder eine Datei beschreiben) können explizit konstruiert werden (siehe Seite 199).

Die Standard-Eingabe unter Verwendung von *Stream*-Objekten erfolgt (wie in zahlreichen Demo-Programmen) unter Verwendung des (*overloaded*) Operators "`>>`". Die für die Ausgabeoperationen definierten Manipulatoren gelten zwar im Prinzip auch für die Eingabe; wirksam sind jedoch nur die drei Radix-Manipulatoren `dec`, `oct` und `hex`. Bei Verwendung des Manipulators `hex` sind die folgenden Eingabe-Strings zulässig und ergeben denselben Wert:

```
abcd
ABCD
aBcD
0xabcd
0XaBcD
...
```

Die Eingabe eines fehlerhaften Wertes, wie etwa Zahlenwerte außerhalb des Wertebereichs oder unzulässige Zeichen in einem numerischen String, wird ein internes Fehler-Flag gesetzt und der *Stream* wird unbrauchbar. Der Status eines *Streams* kann mit den folgenden Klassenelement-Funktionen geprüft werden:

`bad()` Hat das Resultat `TRUE`, wenn ein nicht korrigierbarer Fehler vorliegt.

`fail()` Hat das Resultat `TRUE`, wenn ein nicht korrigierbarer oder ein "vorhersehbarer" Fehler vorliegt.

`good()` Hat das Resultat `TRUE`, wenn kein Fehler vorliegt und das Ende der Eingabedaten noch nicht erreicht wurde.

`eof()` Hat das Resultat `TRUE`, wenn das Ende der Eingabedaten erreicht wurde.

`clear(0)` Setzt den internen Fehler-Status; ein Aufruf mit dem Argument `0` setzt alle Fehler-Flags zurück.

`rdstate()` Gibt den aktuellen Fehler-Status zurück.

Weiters ist der Operator "`!`" durch *Overloading* so definiert, dass er die selbe Funktion erfüllt wie `fail()`. Die folgenden beiden Ausdrücke sind daher äquivalent ("wenn die Eingabe fehlerhaft war"):

```
if (! cin) ...
if (cin.fail()) ...
```

Der Operator **`void*()`** ist als das Gegenteil von `fail()` definiert; die beiden folgenden Ausdrücke bedeuten daher "wenn bei der Eingabe *kein* Fehler aufgetreten ist":

```
if (cin) ...
if (! cin.fail()) ...
```

Beachten Sie die Beziehung

```
cin.good() == ! cin.fail() && ! cin.eof()
```

Das folgende Demo-Programm soll den Einsatz und die Wirkungsweise der Status-Klassenelement-Funktionen demonstrieren:

Demo-Programm 6_03_06.cc

```
#include <iostream.h>

int main ()
{
    int do_clear = 1;                  // Flag-Variable - "clear" aktiv
    int do_flush = 1;                  // Flag-Variable - Rest einlesen
    char antwort;

    cout << "\"clear\" nach einem Fehler (J/n)?  ";
```

```
    cin >> antwort;
    if (antwort == 'n' || antwort == 'N') do_clear = 0;

    cout << "Eingabezeile nach einem Fehler ausleeren (J/n)?  ";
    cin >> antwort;

    if (antwort == 'n' || antwort == 'N') do_flush = 0;

    for (int i = 0; i < 10; i++)
    {
        int x;

        cout << "Bitte eine ganze Zahl eingeben:  ";
        cin >> x;

        if (cin.good()) cout << "Status ist \"good\"";

        if (cin.bad()) cout << "Status ist \"bad\"";

        if (cin.fail())
        {
            cout << "Status ist \"failed\"";

            if (do_clear)
                cin.clear ();

            if (do_flush)
            {
                char buffer[128];
                cin >> buffer;
            }
        }

        cout << " - Eingabe war: " << x << endl;
    }
}
```

Das Programm soll numerische Eingaben von der Konsole einlesen und den dabei erhaltenen Status sowie den eingegebenen Wert wieder ausschreiben. Wenn ein Fehler aufgetreten ist (`cin.fail() != 0`), dann soll je nach dem Zustand zweier Flag-Variablen der Status des *Streams* `cin` zurückgesetzt und/oder der Inhalt der Eingabezeile als String eingelesen (und ignoriert) werden. Die Anzahl der Durchläufe durch die Eingabeschleife ist auf 10 begrenzt, um ein "Aufhängen" des Programms zu verhindern, wenn ungültige Eingaben gemacht werden und der Eingabepuffer nicht entleert werden kann.

Wenn *beide* Maßnahmen zur Fehlerbehebung (Fehler-Status zurücksetzen *und* Eingabezeile ausleeren) verwendet werden, kann man beispielsweise folgenden Dialog erhalten (Konsol-Eingaben sind durch Fettdruck hervorgehoben):

```
"clear" nach einem Fehler (J/n)?  j
Eingabezeile nach einem Fehler ausleeren (J/n)?  j
Bitte eine ganze Zahl eingeben:  123
Status ist "good" - Eingabe war: 123
Bitte eine ganze Zahl eingeben:  45678
Status ist "good" - Eingabe war: 45678
Bitte eine ganze Zahl eingeben:  654.321
Status ist "good" - Eingabe war: 654
Bitte eine ganze Zahl eingeben:  Status ist "failed" - Eingabe war: 654
Bitte eine ganze Zahl eingeben:  987
Status ist "good" - Eingabe war: 987
Bitte eine ganze Zahl eingeben:  das ist keine Zahl
Status ist "failed" - Eingabe war: 987
Bitte eine ganze Zahl eingeben:  Status ist "failed" - Eingabe war: 987
Bitte eine ganze Zahl eingeben:  Status ist "failed" - Eingabe war: 987
Bitte eine ganze Zahl eingeben:  Status ist "failed" - Eingabe war: 987
Bitte eine ganze Zahl eingeben:  -1
Status ist "good" - Eingabe war: -1
```

Die Eingabe ganzzahliger Werte ist erwartungsgemäß unproblematisch. Bei Eingabe von "654.321" wird der erste Teil ("654") als ganze Zahl interpretiert und korrekt umgesetzt; der Eingabezeiger steht danach am Dezimalpunkt. Bei der nächsten Eingabeoperation befinden sich noch Daten (der Rest der vorhergehenden Eingabezeile, also ".321") im Eingabepuffer. Der Dezimalpunkt ist aber kein zulässiges Zeichen für eine ganze Zahl und bewirkt daher einen Fehler; der Wert der Variablen `x` ändert sich nicht. Das "`fail`"-Bit wird anschließend zurückgesetzt und der Rest des Eingabepuffers ausgeleert. Bei der Eingabe von "`das ist keine Zahl`" wird die Eingabeschleife insgesamt viermal (mit Eingabefehler) durchlaufen, weil immer nur ein Wort des Strings aus dem Eingabepuffer eingelesen und entfernt wird.

Wird entweder der Fehler-Status nicht mit `clear()` zurückgesetzt oder der Eingabepuffer im Fehlerfall nicht entleert, so wird nach einer fehlerhaften Eingabe der Ausdruck `cin >> x` effektiv ignoriert; das Programm durchläuft die Schleife mit dem Eingabebefehl so lange, bis die maximale Anzahl der Schleifendurchläufe erreicht wurde.

GNU-C/C++-spezifisch

Aufgrund eines Bugs in GNU C++ funktioniert dieses Demo-Programm nicht wie erwartet, wenn `x` statt als **`int`**-Variable als Variable vom Typ **`double`** deklariert wurde. In diesem Fall bewirken Eingabefehler *nicht* das Setzen des Fehler-Status-Bits, und *jede* Eingabe wird als "`good`" interpretiert.

Ähnlich wie bei den Standard-C-Eingabefunktionen ist es auch hier zweckmäßiger, immer eine ganze Zeile einzulesen und anschließend zu konvertieren. Dafür stehen in der Familie der *Stream*-Ein-/Ausgabefunktionen die Funktionen

```
istream& get (char* ptr, int len, char delim = '\n');
```

und

```
istream& getline (char* ptr, int len, char delim = '\n');
```

zur Verfügung. `len` ist die maximale Anzahl von Zeichen, die in den Puffer gelesen werden sollen, auf den `ptr` zeigt. Beide Funktionen lesen Zeichen bis zum ersten Zeichen, das gleich `delim` ist; `getline()` entfernt den Delimiter aus dem Eingabepuffer, `get()` nicht. Keine der beiden Funktionen kopiert den Delimiter nach `ptr`. Für `get()` existieren übrigens auch andere *overloaded* Definitionen; näheres in `iostream.h`.

Die Verwendung eines Puffers als Quelle für Eingabedaten wird im Detail auf Seite 199 beschrieben.

Das folgende Demo-Programm zeigt eine derartige "stabilere" Eingaberoutine:

Demo-Programm 6_03_07.cc

```
#include <iostream.h>
#include <strstream.h>                          // für istrstream

int main ()
{
    char buffer[128];

    for (int i = 0; i < 10; i++)
    {
        cout << "Ganze Zahl: ";
        cin.getline (buffer, 128);              // eine komplette Zeile einlesen

        istrstream bStream (buffer);
            // Konstruktor für Objekt der Klasse istrstream

        int x;
        bStream >> x;                           // Zuweisung an x

        if (bStream.good())                     // OK
            cout << "OK:     ";

        if (bStream.fail())                     // Fehler
        {
            bStream.clear();
            cout << "Fehler: ";
        }

        cout << x << endl;
    }
}
```

Ähnlich wie das Programm von Seite 184 liest auch dieses Demo-Programm erst eine komplette Eingabezeile in einen Puffer, um sie anschließend zu konvertieren. Die Eingabezeile wird auf jeden Fall nach der Konversion verworfen, sodass auch beliebig fehlerhafte Eingabedaten maximal eine einzige falsche Eingabeoperation bewirken können. Der Dialog mit diesem Programm kann etwa so aussehen:

```
Ganze Zahl: 123
OK:     123
Ganze Zahl: 456
OK:     456
Ganze Zahl: 654.321
OK:     654
Ganze Zahl: das ist keine Zahl
Fehler: 654
Ganze Zahl: 54321
OK:     54321
```

Der Dezimalpunkt in "654.321" verursacht in diesem Fall überhaupt keinen Eingabefehler; der String "das ist keine Zahl" einen einzigen.

6.3.4. Ein-/Ausgabe von/auf Puffer

Ähnlich wie unter Verwendung der Standard-C-Ein-/Ausgabefunktionen `sscanf()` und `sprintf()` kann auch in der C++-*Class Library* `iostream` ein Puffer als Quelle bzw. Ziel der Ein-/Ausgabedaten fungieren. Für eine Eingabe oder Ausgabe auf einen derartigen Puffer muss ein Objekt der geeigneten Klasse (`istrstream`, `ostrstream` oder `strstream`) konstruiert werden; der Puffer (ein Feld von **`char`**s) muss zu diesem Zeitpunkt existieren und dem Konstruktor als Argument übergeben werden. Die solcherart erzeugten Objekte können genauso verwendet werden wie etwa `cin` oder `cout`:

Eingabe von Puffer:

```
char b[128];         // Puffer
istrstream str(b);   // Konstruktor

double x;
str >> x;
```

Eingabe von Konsole:

```
double x;
cin >> x;
```

Ausgabe auf Puffer:

```
char b[128];         // Puffer
ostrstream str(b);   // Konstruktor

double x = 123.456;
str << "x = " << x << endl;
```

Ausgabe auf Konsole:

```
double x = 123.456;
cout << "x = " << x << endl;
```

6.3.5. Ein-/Ausgabe auf Dateien

Ähnlich wie bei der Ein-/Ausgabe unter Verwendung eines Pufferspeichers muss ein *Stream*-Objekt der geeigneten Klassen-Type zuerst konstruiert werden. Die Type richtet sich danach, ob ausschließlich Eingabe- oder ausschließlich Ausgabeoperationen vorgenommen werden sollen, oder eine Mischung von beiden:

Type der Operation	***Stream*-Klasse**
Eingabe	`ifstream`
Ausgabe	`ofstream`
Ein- und Ausgabe	`fstream`

(Simultane Ein- und Ausgabeoperationen auf einen Puffer sind unter Verwendung der Klasse `strstream` möglich.)

Ein *Stream*-Objekt für die Datei-Ein-/Ausgabe muss generell in zwei Stufen vorbereitet werden:

- Erstellung des Objekts selbst unter Verwendung eines geeigneten Konstruktors, und
- Verknüpfung dieses Objekts (das im wesentlichen Kontrollstrukturen enthält) mit der gewünschten Datei.

Dabei sind alternativ die folgenden Vorgangsweisen möglich:

- Konstruktion eines Objekts unter Verwendung des Default-Konstruktors, und anschließender Aufruf der Klassenelement-Funktion `open()`:

```
ifstream myfile;                              // Stream-Objekt
myfile.open ("Datei", iosmode);
```

("Datei" ist der Dateiname und evtl. -Pfad als Null-terminierter ASCII-String; `iosmode` ist einer der in `iostream.h` definierten Enumeratoren, die den Zugriffsmodus auf die Datei angeben:

`ios::in`	Öffnen für Eingabe, oder Öffnen für Ausgabe, ohne dass die Datei verkürzt wird.
`ios::out`	Öffnen für Ausgabe.
`ios::nocreate`	Öffnen, ohne dass eine nicht existierende Datei neu erstellt wird.
`ios::app`	Öffnen für Anhängen von Daten an das Ende der Datei.
`ios::ate`	Ähnlich wie `ios::app`.
`ios::noreplace`	Falls die Ausgabedatei bereits existiert, misslingt das Öffnen.
`ios::binary`	Datei wird im Binär-Mode (statt im Text-Mode) geöffnet.

Andere *Flags* steuern die gemeinsame Benützung von Dateien (*File Sharing*).

Alternativ kann ein *Stream*-Objekt in Speicher vom *Free Store* erstellt werden:

```
ifstream *pmyfile = new ifstream;             // Zeiger auf ein Stream-Objekt
pmyfile->open ("Datei", iosmode);
```

- Die Konstruktion des *Stream*-Objekts und das Öffnen der zugehörigen Datei kann in einem gemeinsamen Konstruktor-Aufruf geschehen:

```
ifstream myfile ("Datei", iosmode);
```

- Einem *Stream*-Objekt kann ein Datei-Deskriptor zugeordnet werden, der mit einer der Standard-C-Funktionen erhalten wurde:

```
int fd = open ("Datei", mode);
ifstream myfile1 (fd);                        // gepufferter Modus (Default)
ifstream myfile2 (fd, NULL, 0);               // ungepufferter Modus
ifstream myfile3 (fd, buffer, size);          // gepufferter Modus mit
                                              // benutzerdefiniertem Puffer
```

Ausgabe-Dateien können analog geöffnet werden.

Ein explizites Schließen einer geöffneten Datei (mit der Klassenelement-Funktion `close()`) ist zwar zulässig, in der Regel aber nicht notwendig, weil die Datei automatisch geschlossen wird, wenn die Gültigkeit des *Stream*-Objekts endet. Die C++-Ein-/ Ausgabe-Funktionen sollen anhand einer adaptierten Version des Telefonregister-Programms von Seite 186 illustriert werden:

Demo-Programm 6_03_08.cc

```
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <string.h>

class tellist
{
public:
    static int zahl;                          // Zahl der Eintragungen

    tellist () { *_name = 0; *_nummer = 0; }  // Default-Konstruktor

    tellist (char *Name, char *Nummer)
    {
        strncpy (_name, Name, 40);
        _name[39] = 0;
        strncpy (_nummer, Nummer, 20);
        _nummer[19] = 0;
    }

    void disp ()
    {
        cout << setiosflags (ios::left) << "Name: " << setw (40) << _name <<
            ": Nummer: " << setw (20) << _nummer << endl <<
            resetiosflags (ios::left);
    }

private:
    char _name[40], _nummer[20];
};

// Funktionen für die Stream-Ein-/Ausgabe
void lesen (tellist *liste, int max_zahl)
{
    int bytes_rd = 0;                         // gelesene Bytes

    ifstream is ("TELLIST.DAT", ios::in | ios::binary | ios::nocreate);

    if (is)                                   // erfolgreich geöffnet
    {
        is.read ((char*) liste, max_zahl * sizeof(tellist));
        bytes_rd = is.gcount (); // gelesene Bytes
    }

    tellist::zahl = bytes_rd / sizeof (tellist);
    cout << tellist::zahl << " Eintragungen gelesen.\n";
}

void schreiben (tellist *liste)
{
    int bytes_wr = 0;                         // geschriebene Bytes

    ofstream os ("TELLIST.DAT", ios::out | ios :: trunc | ios::binary);

    if (os)                                   // erfolgreich geöffnet
    {
        os.write ((char*) liste, tellist::zahl * sizeof(tellist));

        bytes_wr = os.tellp();
            // Position des Ausgabezeigers = Anzahl der geschriebenen Bytes
    }

    cout << bytes_wr / sizeof (tellist) << " Eintragungen geschrieben.\n";
}

int tellist::zahl = 0;                        // statisches Element
const int max_zahl = 20;                      // maximale Zahl der Einträge
```

```
tellist liste[max_zahl];

int main ()
{
    lesen (liste, max_zahl);                    // Liste einlesen

    for (int i = 0; i < tellist::zahl; i++)
        liste[i].disp();                        // Liste ausschreiben

    cout << "\nNeue Eintragungen eingeben; Ende mit leerem Namensfeld:\n";

    while (tellist::zahl < max_zahl)
    {
        char buffer1[128], buffer2[128];

        cout << "Name:    ";
        cin.getline (buffer1, 128);
        if (! *buffer1)                         // leerer String
            break;

        cout << "Nummer:  ";
        cin.getline (buffer2, 128);
        if (! *buffer2)                         // leerer String
            break;

        liste[tellist::zahl++] = tellist (buffer1, buffer2);
    }

    for (i = 0; i < tellist::zahl; i++)
        liste[i].disp();                        // Liste ausschreiben

    schreiben (liste);                          // Liste in Datei schreiben
}
```

GNU-C/C++-spezifisch

Leider ist in der uns zur Verfügung stehenden GNU-C++-Bibliothek ein Fehler in den Ausgabefunktionen enthalten, der beim Aufruf von `os.write()` zu einem *General Protection Fault* führt. Das obige Programm ist daher unter GNU C++ nur lauffähig, wenn die Funktion `schreiben()` folgendermaßen definiert wird:

```
#include <osfcn.h>                              // für ungepufferten I/O
#include <fcntl.h>                              // für ungepufferten I/O
#include <sys/stat.h>                           // für ungepufferten I/O

void schreiben (tellist *liste)
{
    int fd;                                     // File Descriptor
    int bytes_wr = 0;                           // geschriebene Bytes

    if ((fd = open ("TELLIST.DAT", O_WRONLY | O_CREAT | O_BINARY,
        S_IREAD | S_IWRITE)) > 0)               // erfolgreich geöffnet
    {
        bytes_wr = write (fd, liste, tellist::zahl * sizeof(tellist));
        close (fd);
    }

    cout << bytes_wr / sizeof (tellist) << " Eintragungen geschrieben.\n";
}
```

Die Definition der Klasse `tellist` ist im wesentlichen die gleiche wie auf Seite 186; nur die Klassenelement-Funktion `disp()` wurde unter Verwendung der C++-*Stream*-Funktionen neu gestaltet.

Die Funktion `lesen()` verwendet ein Objekt `is` der Klasse `ifstream` für die Eingabe, das unter Verwendung des Konstruktors `ifstream (const char *name, int mode=ios::in, int prot=0664)` definiert wurde. Beachten Sie die bitweise ODER-Verknüpfung der `mode`-Enumeratoren! Die Klassenelement-Funktion `read()` liest (maximal) `max_zahl * sizeof(tellist)` Bytes von der Datei `tellist.dat` ein und speichert diese im Feld `liste` (dessen Startadresse durch einen *Type Cast* in einen Zeiger auf **`char`** umgewandelt wurde, um der Deklaration von `read()`

zu entsprechen). Die Anzahl der tatsächlich gelesenen Bytes kann als Ergebnis der Klassenelement-Funktion `gcount()` erhalten werden; sie wird anschließend in die Anzahl der Elemente der Type `tellist` in `liste` konvertiert. Mit dem Ende der Funktion `lesen()` endet die Gültigkeit des Objekts `is`; der Destruktor der Klasse `ifstream` schließt die Datei `tellist.dat`. Ein separater Aufruf von `close()` ist daher hier nicht erforderlich.

Die Funktion `schreiben()` wäre im Prinzip symmetrisch aufzubauen; es wird dort ein Objekt `os` der Klasse `ofstream` definiert, das verwendet werden sollte, um mittels der Klassenelement-Funktion `write()` eine aus der Anzahl der Elemente vom Typ `tellist` in `liste` und ihrer Größe resultierende Anzahl von Bytes zu schreiben. Die Anzahl der tatsächlich geschrieben Bytes wird anschließend mit der Klassenelement-Funktion `tellp()` ermittelt. (Es ist zweckmäßig, beim Schreiben einer Datei zu prüfen, ob die Datei tatsächlich komplett geschrieben wurde. Wenn beispielsweise der Platz im Ziellaufwerk nicht für die gesamte Datei ausreicht, wird eine verstümmelte Version der Datei geschrieben, ohne dass irgendeine Fehlermeldung erfolgen würde. Die einzige Möglichkeit, diesen Fehler zu erkennen, besteht in einem Vergleich der Anzahl der zu schreibenden und der tatsächlich geschriebenen Bytes. Dies wurde bei allen Versionen dieses Demo-Programms berücksichtigt.)

Infolge des oben erwähnten Fehlers in den C++-Bibliotheksfunktionen des GNU-C++-Compilers ist die *Stream*-Variante von `schreiben()` nicht lauffähig; als *Work-Around* wurde die Version für die ungepufferte Ein-/Ausgabe geringfügig modifiziert.

In `main()` werden letztlich die *Stream*-Konsol-Ein-/Ausgabefunktionen anstelle der Standard-C-Funktionen verwendet.

6.3.6. Standard-C- und C++-*Stream*-Ein-/Ausgabe im Vergleich

Grundsätzlich sind (funktionierende Bibliotheksroutinen vorausgesetzt) unter Verwendung der C++-*Stream*-Funktionen die gleichen Möglichkeiten gegeben wie bei den Standard-C-Funktionen und umgekehrt. Von der unterschiedlichen Konzeption her ergeben sich aber unterschiedliche Vor- und Nachteile beider Systeme:

6.3.6.1. Vorteile der C++-*Stream*-Ein-/Ausgabe-Funktionen

➡ Vollständige Typenprüfung:

Objekte der fundamentalen Typen werden auf jeden Fall korrekt eingelesen oder ausgegeben; Fehlinterpretationen wie bei den Standard-C-Funktionen sind unmöglich (siehe Seite 189).

➡ Einfache formatierte Ausgabe:

Sofern die voreingestellten Ausgabeformate ausreichen, ist eine formatierte Ausgabe mit den C++-*Stream*-Funktionen einfacher als mit den Standard-C-Funktionen.

➡ Weniger kritische Eingabe:

Die Eingabefunktionen von C++ verhalten sich etwas weniger kritisch in ihren Anforderungen an die Formatierung der Eingabedaten als die Standard-C-Funktionen.

➡ Vereinheitlichte Handhabung:

Konzeptuell ist es bei den C++-*Stream*-Funktionen gleichgültig, ob beispielsweise eine formatierte Ausgabe auf die Konsole, in einen Puffer oder in eine Datei erfolgt. (*De facto* müssen aber Puffer- oder Datei-Ausgabe-Objekte ganz anders definiert werden als der Standard-Konsol-*Stream*, sodass sich die Unterschiede in der Behandlung verschiedener Typen von Ein-/Ausgabekanälen nur verschieben.)

➡ Erweiterungsmöglichkeiten für benutzerdefinierte Datentypen:

Durch *Overloading* der Operatoren "`<<`" und "`>>`" können auch benutzerdefinierte Klassen-Typen unmittelbar in Ein-/Ausgabe-Operationen verwendet werden.

6.3.6.2. Vorteile der Standard-C-Ein-/Ausgabe-Funktionen

➡ Geringerer Overhead:

Im Hinblick sowohl auf die Programmgröße als auch auf die Geschwindigkeit der Ausführung sind die Standard-C-Funktionen deutlich effizienter.

➡ Einfachere Realisierung komplizierter Formate:

Eine aufwendige Formatierung von Ausgabedaten ist unter Verwendung von `printf()`, `sprintf()` und `fprintf()` wesentlich einfacher zu realisieren als mit den C++-*Stream*-Funktionen.

➡ Bessere Fehlererkennung:

Die Standard-C-Ein-/Ausgabefunktionen geben fast durchwegs ein Ergebnis zurück, das eine Prüfung auf allfällige Fehler erlaubt. Die äquivalente Information muss bei den C++-*Stream*-Funktionen mühsam durch separate Funktionsaufrufe ermittelt werden.

➡ Dynamischer Aufbau von Formaten:

Die Format-Information ist bei den Standard-C-Funktionen in einem String enthalten, der bei Bedarf auch dynamisch je nach den Erfordernissen des Programms erstellt oder geändert werden kann. Im Gegensatz dazu wird bei den C++-*Stream*-Funktionen das Format von Ein- und Ausgaben durch den *Programmcode* und nicht durch *Daten* festgelegt; dynamische Änderungen sind daher dort viel schwieriger vorzunehmen.

6.4. Abspeichern von Klassen-Objekten in Dateien

C++-spezifisch

Für die Abspeicherung der in einem oder mehreren Klassen-Objekten enthaltenen Daten in einer Datei stehen grundsätzlich zwei Wege offen:

- Umsetzung der Daten in Text; oder
- binäre Abspeicherung des oder der Klassen-Objekte.

Eine Umsetzung in (ASCII-)Text erleichtert den Austausch von Daten mit anderen Applikationen; allenfalls können die solcherart erstellten Dateien auch mit einem gewöhnlichen Text-Editor modifiziert werden.

Die binäre Abspeicherung ist kompakter, schneller und einfacher als eine Umsetzung in lesbaren Text; insbesondere erfordert das Wiedereinlesen der Daten nicht eine mühsame Interpretation einer Text-Datei. (Außerdem wird bei der binären Abspeicherung von Gleitkommawerten ein Genauigkeitsverlust durch Rundungsfehler vermieden.)

Da bei binärer Abspeicherung von Klassenobjekten *alle* Elemente der Klasse abgespeichert werden, auch solche, die vom Compiler unsichtbar angelegt werden (z.B. zur Handhabung virtueller Funktionen), kann die volle Funktionalität der objektorientierten Programmierung gewahrt bleiben.

Das folgende Beispiel erweitert das "Bibliotheksverwaltungsprogramm" von Seite 114 bzw. 117 und 153 um eine Ein-/Ausgabe auf eine Plattendatei. Dabei werden eine Basisklasse, `Dokument`, und drei davon abgeleitete Klassen, `Buch`, `Manual` und `Datei`, mit jeweils spezifischen Eigenschaften definiert:

Demo-Programm 6_04_01.cc

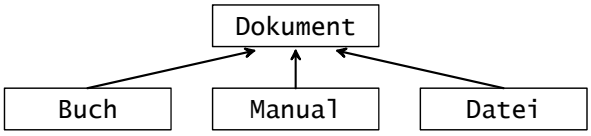
```
#include <iostream.h>
#include <iomanip.h>
#include <strstrea.h>
#include <stdio.h>
#include <string.h>

class Dokument
{
public:
    Dokument() { anzahl++; }            // Default-Konstruktor
    virtual ~Dokument() { anzahl--; }   // Destruktor
    virtual void zeige() { }

    static int anzahl;                  // Anzahl der definierten Objekte

protected:
    int getval()                        // Eingabefunktion für numerische Daten
    {
        char buf[80];
        cin.getline (buf, 80);
        istreamstr str (buf);           // Stream-Konstruktor
        str >> _i;                      // Wert einlesen
    }

    char _s1[40], _s2[20];
    int _i;
};

int Dokument::anzahl = 0;               // Definiere anzahl
```

```
class Buch : public Dokument
{
public:
    Buch ()                             // Default-Konstruktor
    {
        cout << "Buch-Autor:  ";
        cin.getline (_s2, 20);
        cout << "Buch-Titel:  ";
        cin.getline (_s1, 40);
        cout << "Buch-Seiten: ";
        getval ();
    }

    void zeige ()
    {
        cout << setiosflags (ios::left) << "Autor:  " << setw (20) << _s2 <<
            " Titel: " << setw (40) << _s1 << "\n\t" << _i << " Seiten\n";
    }
};

class Manual : public Dokument
{
public:
    Manual ()                           // Default-Konstruktor
    {
        cout << "Manual-Titel:  ";
        cin.getline (_s1, 40);
        cout << "Manual-Seiten: ";
        getval ();
    }

    void zeige ()
    {
        cout << setiosflags (ios::left) << "Manual: " << setw (40) << _s1 <<
            " - " << _i << " Seiten\n";
    }
};

class Datei : public Dokument
{
public:
    Datei ()                            // Default-Konstruktor
    {
        cout << "Datei-Inhalt: ";
        cin.getline (_s1, 40);
        cout << "Datei-Pfad:   ";
        cin.getline (_s2, 20);
        cout << "Datei-Bytes:  ";
        getval ();
    }

    void zeige ()
    {
        cout << setiosflags (ios::left) << "Datei:  "<< setw (40) << _s1 <<
            " Pfad: " << setw (20) << _s2 << "\n\t" << _i << " Bytes\n";
    }
};

// Definition der Funktionen für die Datei-Ein-/Ausgabe

void lesen (Dokument **ppD, int max_zahl)
{
    FILE *file;                         // File Pointer
    int i = 0;
```

```
    if (file = fopen ("LITLIST.DAT", "rb"))
    {                                   // erfolgreich geöffnet
        for ( ; i < max_zahl; i++)
        {
            ppD[i] = new Dokument;

            if (! fread (ppD[i], sizeof(Dokument), 1, file))
            {                           // am Ende der Datei
                delete ppD[i];
                break;
            }
        }

        fclose (file);
    }
    cout << i << " Eintragungen gelesen.\n";
}

void schreiben (Dokument **ppD)
{
    FILE *file;                         // File Pointer
    int i = 0;

    if (file = fopen ("LITLIST.DAT", "wb"))
    {                                   // erfolgreich geöffnet
        for ( ; i < Dokument::anzahl; i++)
            if (! fwrite (ppD[i], sizeof(Dokument), 1, file))
                break;                  // Fehler beim Schreiben

        fclose (file);
    }
    cout << i << " Eintragungen geschrieben.\n";
}

const int max_Eintr = 20;
Dokument *doc[max_Eintr];               // Feld von Zeigern

int main ()
{
    lesen (doc, max_Eintr);             // Liste einlesen

    for (int i = 0; i < Dokument::anzahl; i++)
        doc[i]->zeige();                // Liste ausschreiben

    cout << Dokument::anzahl << " Eintragungen.\n";

    while (Dokument::anzahl < max_Eintr)
    {
        char buf[10];

        if (! cin)                      // Eingabe-Fehler
        {
            cout << "\nFehlerhafte Eingabe!\n";
            cin.clear();                // Fehler-Flags löschen
        }

        cout << "'B'uch, 'M'anual, 'D'atei, oder 'E'nde ?  ";

        cin.getline (buf, 10);          // Schaltzeichen

        switch (*buf)
        {
            case 'B':
            case 'b':
                doc[Dokument::anzahl - 1] = new Buch;
                break;
```

```
            case 'M':
            case 'm':
                doc[Dokument::anzahl - 1] = new Manual;
                break;

            case 'D':
            case 'd':
                doc[Dokument::anzahl - 1] = new Datei;
                break;

            case 'E':
            case 'e':
                goto weiter;

            default:
                cout << "\nFalscher Befehl!\n";
        }
    }

weiter:                                 // Einsprung bei Programm-Abbruch

    for (i = 0; i < Dokument::anzahl; i++)
        doc[i]->zeige();                // Liste ausgeben

    cout << Dokument::anzahl << " Eintragungen.\n";

    schreiben (doc);                    // Liste in Datei schreiben
}
```

Das Demo-Programm verwendet also die folgende Klassen-Hierarchie:



Die Basisklasse, `Dokument`, enthält einen Default-Konstruktor, der die statische Klassenelement-Variable `Dokument::anzahl` inkrementiert. Weiters ist für die Klasse `Dokument` auch ein Destruktor vorgesehen, der `Dokument::anzahl` dekrementiert. Die Destruktor-Funktion wurde als virtuell definiert, um eine Warnung des Compilers zu vermeiden (wenn eine Klasse virtuelle Klassenelement-Funktionen enthält, dann sollte auch ihr Destruktor virtuell sein.) Die virtuelle Klassenelement-Funktion `Dokument:: zeige()` ist leer; sie soll erzwingen, dass die in den abgeleiteten Klassen definierten Funktionen `zeige()` als virtuell interpretiert werden. (Eine Definition als reine virtuelle Funktion, also mit `virtual void zeige() = 0`, ist nicht zweckmäßig, weil damit `Dokument` zu einer abstrakten Klasse würde und die in der Funktion `lesen()` vorgenommene Allokierung eines Objektes der Type `Dokument` nicht mehr möglich wäre.)

Die Struktur der Definition der drei abgeleiteten Klassen ist im wesentlichen ähnlich: Die Default-Konstruktoren der abgeleiteten Klassen fragen interaktiv jeweils nach den erforderlichen Eingabedaten; es wird generell die *Stream*-Funktion `getline()` verwendet, die auch gleich die Eingabestrings auf die zulässige Länge begrenzt. Die Zuordnung der Strings erfolgt (ziemlich willkürlich) jeweils so, dass das größere **char**-Feld `_s1` für die Daten mit dem größeren Platzbedarf verwendet wird. Die numerische Eingabe erfolgt über die Funktion `Dokument::getval()`, die ebenfalls eine Eingabezeile mit `cin.getline()` einliest und anschließend interpretiert. (Würde man eine direkte Zuweisung `cin >> _i` vornehmen, so würde der Zeilenvorschub am Ende der Eingabezeile nicht entfernt, und der nächste Aufruf von `cin.getline()` würde eine leere Eingabezeile zurückgeben.) In einem Fall, in der Klasse `Manual`, wird eines der beiden **char**-Felder nicht benötigt und bleibt daher uninitialisiert. Neben dem Konstruktor enthalten die abgeleiteten Klassen nur noch je eine spezifische Variante der Funktion `zeige()`.

Die Funktionen `lesen()` und `schreiben()` für die Datei-Ein- bzw. Ausgabe verwenden die Standard-C-Funktionen für die gepufferte Ein-/Ausgabe. Beide Funktionen erhalten als Argument einen Zeiger auf das erste Element der heterogenen Liste von Objekten der Klassen `Buch`, `Manual` und `Datei`. (Die Liste besteht aus einem Feld von Zeigern auf die Type `Dokument`; der Zeiger auf das erste Element dieser Liste ist daher ein Zeiger auf einen Zeiger auf die Type `Dokument`, also ein `Dokument**`.)

Die Funktion `lesen()` öffnet die Datei `litlist.dat` zum Lesen; wenn dies erfolgreich möglich war, wird mit dem Operator **`new`** ein Objekt der Klasse `Dokument` allokiert, in das ein Eintrag aus der Datei eingelesen wird. (Dieser Eintrag enthält die eigentlichen Datenelemente zuzüglich der zusätzlichen für den korrekten Aufruf der virtuellen Funktionen für die von `Dokument` abgeleiteten Klassen erforderlichen Informationen.) Jedes Objekt wird also separat eingelesen und in einem individuell allokierten Speicherblock abgelegt, dessen Adresse dem entsprechenden Feldelement in der heterogenen Liste zugewiesen wird. Wenn alle in der Datei enthaltenen Einträge eingelesen wurden, misslingt der nächste Aufruf von `fread()`; die Funktion gibt als Anzahl der gelesenen Datenelemente nicht mehr `1`, sondern `0` zurück. Da das letzte Datenelement damit nicht mehr gültig war, kann der dafür allokierte Speicher mit **`delete`** wieder zurückgegeben werden; die Leseschleife wird abgebrochen.

Die Funktion `schreiben()` ist noch einfacher: Nach dem Öffnen der Datei `litlist.dat` wird der Reihe nach jedes Element der heterogenen Liste in die Datei geschrieben. Falls das Schreiben misslingt (weil beispielsweise die Ausgabe-Platte voll ist), wird die Schleife vorzeitig abgebrochen.

In der Funktion `main()` wird zunächst die Liste eingelesen (falls schon eine solche existiert) und ihr Inhalt auf die Konsole ausgegeben. Anschließend tritt die Funktion in eine Schleife ein, die so lange durchlaufen wird, bis entweder kein Feldelement im Feld `doc` mehr frei ist, oder bis die Eingabe explizit abgebrochen wurde. Zunächst wird ein allfällig beim vorangehenden Schleifendurchlauf aufgetretener Eingabefehler diagnostiziert und das Fehler-Flag wieder gelöscht; solche Fehler können beispielsweise dann auftreten, wenn der Tastatur-Eingabe-String länger als der mit `cin.getline()` zu verwendende Puffer war. (Wird die Prüfung auf einen Fehler nicht vorgenommen und das Fehler-Flag nicht gelöscht, bleibt `cin` im Fehlerfall deaktiviert. Die Schleife wird dann so oft durchlaufen, bis alle Elemente von `doc` belegt wurden; der Inhalt der den Elementen von `doc` zugeordneten Objekte ist jedoch nicht definiert.) Anschließend wird die Eingabe eines Schalt-Zeichens angefordert, das eine Verzweigung zu einem Aufruf des passenden Konstruktors bewirkt. (Da Groß- oder Kleinbuchstaben eingegeben werden können, muss das Schalt-Zeichen entweder in eine der beiden Schreibweisen umgewandelt werden, oder es müssen wie im vorliegenden Programm beide Möglichkeiten zugelassen werden.)

Beim Aufruf eines der Konstruktoren der abgeleiteten Klassen geschieht folgendes:

- Mit dem Operator **`new`** wird der erforderliche Speicherplatz für ein Objekt, 68 Bytes, allokiert.
- Der Default-Konstruktor der Basisklasse, `Dokument`, wird aufgerufen. Dieser inkrementiert das statische Klassenelement `Dokument::anzahl` und markiert das Objekt zunächst als ein Objekt der Klasse `Dokument`.
- Anschließend wird der Default-Konstruktor der jeweiligen abgeleiteten Klasse ausgeführt, der die Datenelemente des Klassenobjekts interaktiv entsprechend setzt. Der Compiler fügt zusätzliche Daten hinzu, die einen Aufruf der korrekten virtuellen Funktion `zeige()` erlauben.
- Die Adresse des solchermaßen initialisierten Speicherblocks wird dem ersten unbelegten Element des Feldes `doc` zugewiesen. Da `Dokument::anzahl` bereits vom Konstruktor von `Dokument` inkrementiert wurde, muss der Index um `1` verringert werden, um das jeweils erste unbelegte Element des Feldes adressieren zu können.

Bei einem Abbruch durch Eingabe von "`E`" oder "`e`" wird mit einem **`goto`**-Befehl aus der Schleife herausgesprungen. (Das ist nicht die einzige denkbare Lösung, wie eine Schleife abgebrochen werden kann, aber sicher die einfachste.) Der Inhalt der Liste wird nochmals am Bildschirm ausgegeben und anschließend mit `schreiben()` in die Datei `litlist.dat` geschrieben.

Der Aufruf der korrekten virtuellen Funktionen ist auch nach dem Abspeichern und Wiederladen der Daten deshalb möglich, weil *alle* Datenelemente in der Klasse `Dokument` konzentriert sind, und weil auch die Information über die zu verwendende virtuelle Funktion in `Dokument` (unsichtbar) als Datenelement abgelegt wird. Die Größe eines Objektes der Klasse `Dokument` (68 Bytes) setzt sich zusammen aus dem Speicherplatzbedarf der beiden **`char`**-Felder `_s1` und `_s2` (40 bzw. 20 Bytes), der **`int`**-Variablen `_i` (4 Bytes) und einem Funktionszeiger auf die korrekte virtuelle Funktion `zeige()` (4 Bytes) (genau genommen, einem Zeiger auf einen Tabelleneintrag mit dem Zeiger auf `zeige()`).

Die Verwendung eines Funktionszeigers zur Auswahl der passenden virtuellen Funktion ist zwar von ihrer Effizienz her unübertroffen, sie ist aber mit einem schwerwiegenden Nachteil verbunden: Selbst geringfügige Änderungen des Programms können die Adressen der diversen Programm-Funktionen verändern; ein abgespeicherter Zeiger zeigt dann nicht mehr auf den richtigen Tabelleneintrag mit der Startadresse einer Funktion, sondern irgendwohin. Ein Absturz des Programms ist also fast unvermeidlich, wenn nach einer Programmänderung die mit der Vorgängerversion erstellte Datei eingelesen wird und bei der Ausgabe der gespeicherten Daten die virtuelle Funktion `zeige()` aufgerufen werden soll. Der hier gewählte Weg ist daher nur dann prak-

tikabel, wenn sichergestellt werden kann, dass das selbe Programm die Daten wieder liest, das sie auch geschrieben hat. (Das ist beispielsweise bei temporär angelegten Dateien implizit der Fall.)

In allen anderen Fällen, in denen die gespeicherten Daten portabel sein sollen, muss hingegen ein alternativer Weg beschritten werden. In der folgenden Version des Demo-Programms sind die Änderungen gegenüber der vorigen Version fett gedruckt:

Demo-Programm 6_04_02.cc

```
#include <iostream.h>
#include <iomanip.h>
#include <strstrea.h>
#include <stdio.h>
#include <string.h>

// Aufzählungs-Type zur Charakterisierung des Objekts

enum dtype { dokument, buch, manual, datei };

class Dokument
{
public:
    Dokument()                          // Default-Konstruktor
    {
        anzahl++;
        _dt = dokument;                 // Daten-Type
    }

    virtual ~Dokument() { anzahl--; }   // Destruktor
    virtual void zeige() { }

    static int anzahl;                  // Anzahl der definierten Objekte

protected:
    int getval()                        // Eingabefunktion für numerische Daten
    {
        char buf[80];
        cin.getline (buf, 80);
        istreamstr str (buf);           // Stream-Konstruktor
        str >> _i;                      // Wert einlesen
    }

    char _s1[40], _s2[20];
    int _i;

    dtype _dt;                          // Daten-Type

    friend void lesen (Dokument **, int);
};

int Dokument::anzahl = 0;               // Definiere anzahl

class Buch : public Dokument
{
public:
    Buch ()                             // Default-Konstruktor
    {
        cout << "Buch-Autor:  ";
        cin.getline (_s2, 20);
        cout << "Buch-Titel:  ";
        cin.getline (_s1, 40);
        cout << "Buch-Seiten: ";
        getval ();
        _dt = buch;                     // Daten-Type
    }
```

```
    Buch (Dokument &d)                  // alternativer Konstruktor
    {
        strcpy (_s1, d._s1);
        strcpy (_s2, d._s2);
        _i = d._i;
        _dt = buch;                     // Daten-Type
    }

    void zeige ()
    {
        cout << setiosflags (ios::left) << "Autor:  " << setw (20) << _s2 <<
            " Titel: " << setw (40) << _s1 << "\n\t" << _i << " Seiten\n";
    }
};

class Manual : public Dokument
{
public:
    Manual ()                           // Default-Konstruktor
    {
        cout << "Manual-Titel:  ";
        cin.getline (_s1, 40);
        cout << "Manual-Seiten: ";
        getval ();
        _dt = manual;                   // Daten-Type
    }

    Manual (Dokument &d)                // alternativer Konstruktor
    {
        strcpy (_s1, d._s1);
        _i = d._i;
        _dt = manual;                   // Daten-Type
    }

    void zeige ()
    {
        cout << setiosflags (ios::left) << "Manual: " << setw (40) << _s1 <<
            " - " << _i << " Seiten\n";
    }
};

class Datei : public Dokument
{
public:
    Datei () // Default-Konstruktor
    {
        cout << "Datei-Inhalt: ";
        cin.getline (_s1, 40);
        cout << "Datei-Pfad:   ";
        cin.getline (_s2, 20);
        cout << "Datei-Bytes:  ";
        getval ();
        _dt = datei;                    // Daten-Type
    }

    Datei (Dokument &d)                 // alternativer Konstruktor
    {
        strcpy (_s1, d._s1);
        strcpy (_s2, d._s2);
        _i = d._i;
        _dt = datei;                    // Daten-Type
    }
```

```
    void zeige ()
    {
        cout << setiosflags (ios::left) << "Datei:  "<< setw (40) << _s1 <<
            " Pfad: " << setw (20) << _s2 << "\n\t" << _i << " Bytes\n";
    }
};

// Definition der Funktionen für die Datei-Ein-/Ausgabe

void lesen (Dokument **ppD, int max_zahl)
{
    FILE *file;                         // File Pointer
    int i = 0;
    Dokument Temp;                      // Hilfs-Objekt

    if (file = fopen ("LITLIST1.DAT", "rb"))
    {                                   // erfolgreich geöffnet
        for ( ; i < max_zahl; i++)
        {
            if (! fread (&Temp, sizeof(Dokument), 1, file))
                break;                  // am Ende der Datei

            switch (Temp._dt)           // verzweige je nach Datentype
            {
                case dokument:
                    cout << "Falsche Objekt-Type!\n";
                    i--;                // kein Feldelement definiert
                    break;

                case buch:
                    ppD[i] = new Buch (Temp);
                    break;

                case manual:
                    ppD[i] = new Manual (Temp);
                    break;

                case datei:
                    ppD[i] = new Datei (Temp);
                    break;
            }
        }

        fclose (file);
    }

    cout << i << " Eintragungen gelesen.\n";
}

void schreiben (Dokument **ppD)
{
    FILE *file;                         // File Pointer
    int i = 0;

    if (file = fopen ("LITLIST1.DAT", "wb"))
    {                                   // erfolgreich geöffnet
        for ( ; i < Dokument::anzahl; i++)
            if (! fwrite (ppD[i], sizeof(Dokument), 1, file))
                break;                  // Fehler beim Schreiben

        fclose (file);
    }

    cout << i << " Eintragungen geschrieben.\n";
}

const int max_Eintr = 20;
```

```
Dokument *doc[max_Eintr];              // Feld von Zeigern

int main ()
{
    // wie im Demo-Programm von Seite 207
}
```

Diese Version des Programms (die nur eine von mehreren möglichen Alternativen zur Erstellung einer portablen Datei darstellt) verwendet einen Enumerator (das Klassenelement `_dt`), um die Type eines Objekts einer abgeleiteten Klasse festzuhalten. Es muss daher in jedem Konstruktor `_dt` auf die passende Type gesetzt werden. (Die Aufzählungs-Type `dtype` muss global deklariert werden, um nicht nur innerhalb der Klassenelement-Funktionen von `Dokument` und den abgeleiteten Klassen, sondern auch in der Funktion `lesen()` zugänglich zu sein.) Zusätzlich wird für alle abgeleiteten Klassen — `Buch`, `Manual` und `Datei` — ein alternativer Konstruktor vorgesehen, der ein Objekt dieser Klasse aus den Informationen eines als Argument übergebenen Objekts der Klasse `Dokument` erstellt (und natürlich ebenfalls `_dt` entsprechend setzt).

Die Funktion für das Abspeichern der Daten ist — abgesehen vom geänderten Namen der Datei (`litlist1.dat`) — gegenüber der ursprünglichen Version unverändert. Hingegen wurden in der Funktion `lesen()` mehrere Änderungen vorgenommen:

- In `lesen()` wird ein Objekt der Klasse `Dokument`, `Temp` (mit automatischer Speicherklasse) definiert. Dieses Objekt (und nicht ein mit **`new`** allokierter Speicherblock) wird als Ziel der Leseoperation verwendet.
- Das Klassenelement `Temp._dt`, das beim Einlesen eines Objekts ja entsprechend dem ursprünglichen Datentyp dieses Objekts gesetzt wurde, wird verwendet, um in einem **`switch`**-Befehl zwischen den Aufrufen der verschiedenen alternativen Konstruktoren zu wählen. Diese initialisieren jeweils einen mit **`new`** allokierten Speicherblock und weisen seine Adresse dem entsprechenden Feldelement von `doc` zu. Der Wert `dokument` von `Type._dt` sollte nie vorkommen; im Fehlerfall wird eine Meldung ausgegeben und die Zählvariable `i` dekrementiert, um die Inkrementierung in der **`for`**-Schleife zu kompensieren. (In diesem Fall wird ja kein Element von `doc` definiert; ohne diese Maßnahme würde das Feld `doc` "Lücken" bekommen.)
- Am Ende der Funktion `lesen()` wird das automatische Objekt `Temp` wieder zerstört. Das statische Datenelement `Dokument::anzahl` erhält damit wieder die korrekte Anzahl der aktiven Elemente der im Feld `doc` abgespeicherten heterogenen Liste.

Jedes Objekt der Klasse `Dokument` und ihrer abgeleiteten Klassen (und damit auch jeder Eintrag in der Datei `litlist1.dat`) umfasst nun 72 Bytes: 40 bzw. 20 Bytes für die **`char`**-Felder `_s1` und `_s2`, 4 Bytes für `_i`, 4 Bytes für `_dt`, und 4 Bytes für den Zeiger zum Aufruf der virtuellen Funktion `zeige()`. Dieser Zeiger wird aber nur mehr dann verwendet, wenn ein Objekt tatsächlich im Arbeitsspeicher steht; sein Wert wird vom Konstruktor der jeweiligen Klasse festgelegt, also auch in den Konstruktor-Aufrufen nach dem Einlesen einer Datei. Der in der Datei enthaltene Wert des Zeigers wird *de facto* ignoriert. (Zumindest, solange nicht eine virtuelle Funktion für das Objekt `Temp` aufgerufen wird, nachdem es von einem Eintrag der Datei überschrieben wurde.) Damit kann die selbe Datei von verschiedenen Versionen des Programms verwendet werden, vorausgesetzt, die Größe der Felder und die Anzahl der Datenelemente ändert sich nicht. (Dieser Umstand kann durch abwechselnden Aufruf der beiden Demo-Programme `6_04_02.cc` und `6_04_03.cc` demonstriert werden, die sich durch einen zusätzlichen Ausgabebefehl in `main()` voneinander unterscheiden, der tatsächlich in unterschiedlichen Zeigern auf die virtuellen Funktionen resultiert.)

Demo-Programm 6_04_03.cc

7. Der C++-Präprozessor

In diesem Abschnitt werden die Operationen beschrieben, die der C++-Präprozessor vor der eigentlichen Übersetzung eines Programms vornimmt. Diese umfassen

- die Definition von manifesten Konstanten und Makros;
- die Auflösung von Makros;
- die Einbindung von beliebigen Dateien, insbesondere von *Header*-Dateien;
- die bedingte Übersetzung von Teilen des Programms; und
- verschiedene Funktionen für die Entwicklung und das Testen von Programmcode.

7.1. Die Funktionen des C++-Präprozessors

7.1.1. Allgemeines

C++- (und Standard-C-) Programme werden vor ihrer eigentlichen Übersetzung einer Vorverarbeitung im *Präprozessor* unterworfen. Dieser ist ein reiner *Textprozessor*, das heißt, er ersetzt gewisse Passagen im Quellcode durch andere Inhalte. Dazu gehört

- die Ersetzung symbolischer Namen durch andere oder durch Konstanten;
- die Expansion von *Makros*;
- das Einbinden anderer Quelldateien;
- das Ausblenden von Teilen des Quellcodes in Abhängigkeit vom Ergebnis der Prüfung einer Bedingung;
- die Einstellung gewisser (implementierungsspezifischer) Compilerparameter.

Ziel dieser Vorverarbeitung ist es, Quellprogramme einfacher, leichter wartbar und portabler zu gestalten. Dies wird erreicht durch:

- Festlegung konstanter Programmparameter an einer leicht zugänglichen Stelle;
- kompaktere Darstellung oft gebrauchter Code-Sequenzen durch Makros;
- Replizieren gewisser Deklarationen oder Definitionen mit verringerter Fehleranfälligkeit in einer beliebigen Anzahl von Übersetzungseinheiten;
- Übersetzung von Programmen mit unterschiedlichen Parametern oder zu unterschiedlichen Teilen.

Generell ist das Ziel bei der Verwendung von Präprozessor-Direktiven, Adaptierungen eines Programms an möglichst wenigen Stellen des Programms mit möglichst globaler Wirkung vornehmen zu können.

Präprozessor-Direktiven werden ausschließlich vom Präprozessor interpretiert und aus der von ihm an den eigentlichen Compiler weitergeleiteten Ausgabedatei entfernt. Sie müssen grundsätzlich mit einem "**#**" beginnen, das das erste nicht-leere Zeichen der Zeile sein muss. Präprozessor-Befehle können sich auch über mehrere Zeilen erstrecken, vorausgesetzt, das *letzte* Zeichen in der Zeile unmittelbar vor dem Zeilenvorschub ist ein "****". Sie können Kommentare in "**/* */**" oder nach "**//**" enthalten. Die folgenden Präprozessor-Direktiven sind definiert:

`#define`	**`#endif`**	**`#ifdef`**	**`#line`**
`#elif`	**`#error`**	**`#ifndef`**	**`#pragma`**
`#else`	**`#if`**	**`#include`**	**`#undef`**

Zusätzlich zur Bearbeitung der eigentlichen Präprozessor-Direktiven *expandiert* der Präprozessor *manifeste Konstanten* und *Makros* auch in jenen Teilen des Programms, die *keine* Präprozessor-Direktiven sind. Manifeste Konstanten und Makros werden grundsätzlich mit der **`#define`**-Direktive definiert (es gibt einige vordefinierte Konstanten, die beispielsweise die Type des Compilers oder der verwendeten CPU beschreiben); mit **`#define`** definierte Namen, die für eine Konstante stehen, sind manifeste Konstanten; solche, die für einen Ausdruck oder Befehl stehen, sind Makros. Makros können auch eine beliebige Anzahl von Argumenten haben. In jedem Fall werden alle mit **`#define`** definierten Namen in der Übersetzungseinheit vom Präprozessor durch den ihnen entsprechenden Wert, Ausdruck oder Befehl ersetzt (wobei die Ersetzung *wörtlich* vorgenommen wird); die formalen Argumente von funktionsartigen Makros werden dabei durch die aktuellen Argumente des Makro-Aufrufs ersetzt.

Der Präprozessor schreibt seine Ausgabedaten grundsätzlich in eine Datei, die dann, wenn der Präprozessor aus dem Kontext des Compilers heraus aufgerufen wurde, automatisch wieder gelöscht wird, sobald sie vom Compiler bearbeitet wurde. Manche Compiler erlauben einen separa-

ten Aufruf des Präprozessors; alle erlauben die Ausgabe der vom Präprozessor bearbeiteten Übersetzungseinheit in eine permanente Datei, die mit jedem Texteditor inspiziert werden kann.

7.1.2. Die Rolle des Präprozessors in C++

C++-spezifisch

Viele in Standard-C unerlässliche Funktionen des Präprozessors wurden in C++ durch Funktionen des Compilers ersetzt oder zumindest dupliziert:

Standard-C:	**C++:**
manifeste Konstanten (mit **`#define`** definiert)	Konstanten (mit dem Schlüsselwort **`const`** deklariert)
Makros	**`inline`**-Funktionen

Die Vorteile der C++-Lösung —

- Möglichkeit einer strikten Typenprüfung;
- Verfügbarkeit von Konstanten oder Enumeratoren mit ihren symbolischen Namen innerhalb eines symbolischen Debuggers;
- keine Probleme mit den Nebenwirkungen von Makro-Aufrufen —

lassen ihre Anwendung anstelle der Standard-C-Präprozessor-Lösungen als angezeigt erscheinen. Aus Kompatibilitätsgründen bleibt jedoch die volle Funktionalität des Standard-C-Präpro-zessors auch in C++ erhalten.

7.2. Präprozessor-Direktiven

7.2.1. Die **#define**-Direktive

Die **#define**-Direktive erlaubt die Definition eines symbolischen Namens für eine Konstante, einen Ausdruck oder einen Befehl (generell, für ein *Token*):

#define *Name Token-String*

Alle auf diese Definition folgenden Vorkommen von *Name* werden durch *Token-String* ersetzt, jedoch nur dann, wenn *Name* tatsächlich ein *Token* bildet. Dies ist *nicht* der Fall

- wenn *Name Teil* eines anderen Namens ist;
- wenn *Name* innerhalb eines Strings vorkommt;
- wenn *Name* innerhalb eines Kommentars steht.

Es ist auch zulässig, *keinen Token-String* in einer **#define**-Direktive anzugeben; in diesem Fall werden alle Vorkommen von *Name* im folgenden Teil der Übersetzungseinheit entfernt. Trotzdem gilt *Name* aber als definiert (und wird von den Direktiven **#if defined** oder **#ifdef** als existent betrachtet).

Unmittelbar anschließend an *Name* (*ohne* Leerzeichen) können in Klammern und durch Kommas getrennt *formale Argumente* angeführt werden, die beliebig oft in *Token-String* vorkommen können. Diese formalen Argumente werden bei der Expansion eines Makros durch die aktuellen Argumente des Makro-Aufrufs ersetzt.

Die Funktion der verschiedenen Formen der **#define**-Direktive wird anhand des folgenden Beispiels demonstriert:

Demo-Programm 7_02_01.cc

```
#define PROGRAMM_VERSION   "2.57"              // String
#define PI       3.141592                      // Manifeste Konstante
#define TWOPI    2*PI                          // Manifeste Konstante verwendet

#define max(a,b) (((a) > (b)) ? (a) : (b))     // Makro-Definition
#define min(a,b) (((a) < (b)) ? (a) : (b))     // Makro-Definition

char *Sign_On = "Willkommen zu Version " PROGRAMM_VERSION " von GNU C++!";

int main ()
{
    int int_1 = 5;
    int int_2 = 7;

    int i = max (int_1, int_2);
    int j = min (int_1, int_2);

    double r = 17;
    double A = r * r * PI;
    double U = r * TWOPI;

    double x = max (PI, TWOPI);
}
```

Aus diesem Quellprogramm macht der Präprozessor:

Demo-Programm 7_02_01.i

```
# 1 "7_02_01.cc"

        ( 9 Leerzeilen entfernt)

char *Sign_On= "Willkommen zu Version "    "2.57"  " von GNU C++!";
```

```
int main ()
{
    int i_1 = 5;
    int i_2 = 7;

    int i = ((( i_1 ) > (  i_2 )) ? ( i_1 ) : (  i_2 )) ;
    int j = ((( i_1 ) < (  i_2 )) ? ( i_1 ) : (  i_2 )) ;

    double r = 17;
    double A = r * r *    3.141592  ;
    double U = r * 2*   3.141592   ;

    double x = (((    3.141592   ) > (  2*   3.141592    )) ? (    3.141592
) : (  2*   3.141592    )) ;
}
```

Beachten Sie bitte, dass bei Verwendung manifester Konstanten wie `PROGRAMM_VERSION`, `PI` oder `TWOPI` diese tatsächlich durch den Rest der auf ihren Namen in der **`#define`**-Direktive folgenden String, einschließlich Leerzeichen oder Tabulatoren, jedoch ausschließlich der Kommentare, ersetzt werden. Mit **`#define`** definierte Konstante können auch in der Definition anderer Konstanten (`TWOPI`) und als aktuelle Argumente (`max (PI, TWOPI)`) eines Makroaufrufs verwendet werden. Dabei ist die Reihenfolge, in der die Definitionen erfolgten, gleichgültig: Wenn `PI` *vor* `TWOPI` definiert wird, wird zunächst jedes Vorkommen des *Tokens* `PI` durch `3.141592` ersetzt, also auch in der Definition von `TWOPI`, die dann also "`2* 3.141592`" lautet. Im umgekehrten Fall wird überall, wo `TWOPI` steht, "`2*PI`" eingesetzt; anschließend wird das *Token* `PI` überall, also auch in "`2*PI`", durch `3.141592` ersetzt. Das Ergebnis ist also in beiden Fällen das gleiche. (Wenn ein Name *vor* der **`#define`**-Direktive vorkommt, mit der er definiert wird, wird er dort jedoch nicht ersetzt.)

Funktionsartige Makros (wie `max(a, b)`) werden überall dort im Programmcode expandiert, wo ihrem Namen eine Parameterliste in Klammern folgt. Jedes formale Argument, dem nicht einer der Präprozessor-Operatoren "`#`" oder "`##`" (siehe Seite 220ff) vorangeht, und der nicht von "`##`" gefolgt wird, wird im Makro durch das korrespondierende aktuelle Argument ersetzt.

GNU-C/C++-spezifisch

Eine weitere Definition desselben Namens mit **`#define`** bewirkt eine Fehlermeldung, ausgenommen, beide Definitionen sind identisch. Manche Compiler, so auch GNU C++, lassen eine nochmalige Definition auch zu (wenn auch mit einer Warnung), wenn die beiden Definitionen *sinngemäß* gleich sind, also beispielsweise:

```
#define max(a,b) (((a) > (b)) ? (a) : (b))
#define max(c,d) (((c) > (d)) ? (c) : (d))
```

Beachten Sie bitte die Klammern um die Ausdrücke in einer Makro-Definition! Diese sind erforderlich, um eine korrekte Interpretation von Ausdrücken zu erzwingen, die als aktuelle Argumente an das Makro übergeben werden (siehe Seite 74).

7.2.2. Operatoren für **`#define`**

Zwei der drei präprozessorspezifischen Operatoren können nur im Zusammenhang mit einer **`#define`**-Direktive bei der Definition eines Makros verwendet werden:

7.2.2.1. Der "Stringmacher"-Operator "`#`"

Dieser Operator kann im "Körper" eines Makros einem formalen Argument vorangestellt werden. Bei der Expansion des Makros wird das aktuelle Argument an dieser Stelle des Makros in einen String umgewandelt (also mit "`"`" "eingeklammert"), wobei Leerzeichen vor dem ersten und nach dem letzten *Token* des Strings ignoriert werden. Wenn im aktuellen Argument die Zeichen "`"`" oder "`\`" vorkommen, wird ihnen ein "`\`" vorangestellt. Das folgende Beispiel zeigt dieses Verhalten:

Demo-Programm 7_02_02.cc

```
#include <iostream.h>

#define melde(x) cout << ">" #x "<\n"

int main ()
{
    melde ( Das ist eine normale Ausgabe. );
    melde ( Das ist "in Anführungszeichen");
    melde ( '"' wird als '\"' dargestellt.);
}
```

Der Präprozessor bindet die Datei `iostream.h` (und weitere von dieser inkludierte) ein; am Ende seiner ziemlich umfangreichen Ausgabedatei steht:

```
int main ()
{
    cout << ">" "Das ist eine normale Ausgabe." "<\n" ;
    cout << ">" "Das ist \"in Anf\201hrungszeichen\"" "<\n" ;
    cout << ">" "'\"' wird als '\\\"' dargestellt." "<\n" ;
}
```

Bei Ausführung des Programms ist die Ausgabe:

```
>Das ist eine normale Ausgabe.<
>Das ist "in Anführungszeichen"<
>'"' wird als '\"' dargestellt.<
```

Eine sinnvollere Anwendung ist ein Makro, das den Namen und den Wert einer Variablen oder eines Ausdrucks (von fundamentalem Typ) ausgibt:

Demo-Programm 7_02_03.cc

```
#include <iostream.h>

#define ZeigeWert(x) cout << #x " = " << (x) << endl;

int main ()
{
    int i = 13;
    double x = 3.141592;

    ZeigeWert (i);
    ZeigeWert (x);
    ZeigeWert (i * x);
}
```

Der Präprozessor macht aus `main()`:

```
int main ()
{
    int i = 13;
    double x = 3.141592;

    cout << "i" " = " << ( i ) << endl;  ;
    cout << "x" " = " << ( x ) << endl;  ;
    cout << "i * x" " = " << ( i * x ) << endl;  ;
}
```

Bei seiner Ausführung gibt das Programm aus:

```
i = 13
x = 3.14159
i * x = 40.8407
```

Beachten Sie bitte, dass in der Definition von `ZeigeWert` dem formalen Argument einmal "`#`" vorangestellt ist und einmal nicht.

7.2.2.2. Der "Tokenverbindungs"-Operator "##"

Dieser Operator erlaubt das Verbinden zweier getrennter *Tokens* zu einem einzigen (wobei mindestens eines der *Tokens* sinnvollerweise ein Argument des Makroaufrufs sein sollte). Die sich dabei ergebenden *Tokens* können auch ihrerseits die Namen von Makros oder manifesten Konstanten sein, die entsprechend umgesetzt werden. Da "**##**" zwei benachbarte *Tokens* verbinden muss, kann es weder das erste noch das letzte *Token* in der Definition des Makros sein. Leerzeichen vor oder nach "**##**" sind zulässig.

Demo-Programm 7_02_04.cc

```
#include <iostream.h>

#define Makro(x) cout << "Wert Nr. " #x " = " << i ## x << endl;

#define KONST_1     111111
#define KONST_2     222222
#define KONST_3     333333

#define Konst(x)    KONST_##x

int main ()
{
    int i1 = 123;
    int i2 = 234;
    int i3 = 345;

    Makro(1);                           // gibt i1 aus
    Makro(2);                           // gibt i2 aus
    Makro(3);                           // gibt i3 aus

    cout << Konst(1) << endl;           // 111111
    cout << Konst(2) << endl;           // 222222
    cout << Konst(3) << endl;           // 333333
}
```

Der Präprozessor bindet wieder einige *Include*-Dateien ein; am Ende seiner Ausgabedatei steht:

```
int main ()
{
    int i1 = 123;
    int i2 = 234;
    int i3 = 345;

    cout << "Wert Nr. " "1" " = " << i1  << endl ;
    cout << "Wert Nr. " "2" " = " << i2  << endl ;
    cout << "Wert Nr. " "3" " = " << i3  << endl ;

    cout <<                111111    << endl;
    cout <<                222222    << endl;
    cout <<                333333    << endl;
}
```

Wenn das Programm fertig übersetzt und ausgeführt wird, ist seine Ausgabe:

```
Wert Nr. 1 = 123
Wert Nr. 2 = 234
Wert Nr. 3 = 345
111111
222222
333333
```

7.2.3. Die **#undef**-Direktive

Die **#undef**-Direktive hebt die Definition eines mit **#define** definierten Namens wieder auf. Dies ist zum Beispiel dann erforderlich, wenn ein Makro oder eine Konstante anderswo (beispielsweise in einer der *Include*-Dateien des Compilers) definiert wurde, aber seine Definition geändert werden soll.

Das folgende Beispiel geht von der Annahme aus, dass mit dem GNU C++-Compiler ein von einer externen Quelle stammendes Programm übersetzt werden soll, das etliche der in der *Include*-Datei `ctype.h` definierte Funktionen benötigt, sodass diese Datei mit **#include** eingebunden werden sollte. In `ctype.h` wird aber auch das Makro `toupper(c)` definiert, wobei aber die Definition in GNU C++ *nicht* die Nebenwirkungen der "klassischen" Definition dieses Makros aufweist:

```
#define toupper(c)  ({int _c=(c); islower(_c) ? (_c-'a'+'A') : _c; })
```

Wenn aber gerade diese Nebenwirkungen für dieses Programm erwünscht sind, muss `toupper(c)` in seiner "klassischen" Form definiert werden:

```
#define _toupper(c) ( (c)-'a'+'A' )
#define toupper(c)  ( (islower(c)) ? _toupper(c) : (c) )
```

Eine derartige Neudefinition ist aber nur dann möglich, wenn die vorhergehende Definition mit **#undef** wieder aufgehoben wurde. Das könnte etwa so aussehen:

Demo-Programm 7_02_05.cc

```
#include <ctype.h>                          // GNU-C++-Definition

char func1 (char ch)
{
    return toupper(ch);
}

#undef toupper                              // vorige Definition ungültig

// hier darf toupper(c) nicht aufgerufen werden, weil dieser Name
// hier nicht existiert

#define _toupper(c) ( (c)-'a'+'A' )     // "klassische" Definition
#define toupper(c) ( (islower(c)) ? _toupper(c) : (c) )

char func2 (char ch)
{
    return toupper(ch);
}
```

Nach Bearbeitung durch den Präprozessor erhalten wir:

```
char func1 (char ch)
{
    return  ({int _c=( ch );  ((_ctype_+1)[ _c ]&02 ) ? (_c-'a'+'A') : _c; }) ;
}
```

(einige Leerzeilen gelöscht)

```
char func2 (char ch)
{
    return ( ( ((_ctype_+1)[  ch  ]&02 ) ) ? ( (  ch  )-'a'+'A' )  : ( ch ) ) ;
}
```

(`islower(c)` ist übrigens auch ein Makro, wie an seiner Umsetzung in "`(_ctype_+1)[ _c ]&02`" ersehen werden kann. Üblicherweise verwendet man in C/C++ für dieses und verwandte Makros eine Tabelle `_ctype_` mit einem Byte pro Zeichen des ASCII-Zeichensatzes, in dem verschiedene Bits die verschiedenen Eigenschaften des betreffenden Zeichens repräsentieren.)

Beachten Sie bitte die Syntax bei der Aufhebung der Definition eines funktionsartigen Makros mit **#undef**: in diesem Fall ist *nur* der Name des Makros anzugeben und *keine Argumente*!

#undef kann auch verwendet werden, um Namen ungültig zu machen, die Elemente der Sprache selbst sind. Im folgenden Beispiel wird die Definition der Type **long** durch **double** ersetzt:

Demo-Programm 7_02_06.cc

```
#include <iostream.h>

#undef long
#define long double

int main()
{
    long pi = 3.141592;
    cout << pi << endl;
}
```

Tatsächlich bekommt der eigentliche Compiler die Zeile "`long pi = 3.141592;`" nie zu "sehen"; der Präprozessor liefert ihm nämlich:

```
int main()
{
    double  pi = 3.141592;
    cout << pi << endl;
}
```

Dementsprechend schreibt das Programm auch eine Gleitkommazahl aus.

Beachten Sie bitte, dass eine derartige "Umdefinition" einer Type nichts mit einem **typedef**-Befehl zu tun hat: Eine Definition

```
typedef long double;
```

wäre zwar zulässig, würde aber an der Gültigkeit und Bedeutung der Type **long** überhaupt nichts ändern. Hingegen werden im obigen Demo-Programm alle Vorkommen von "**long**" durch "**double**" ersetzt, *bevor* der eigentliche Compiler das Programm übersetzt.

In der Praxis kann die Aufhebung der Definition eines Namens verwendet werden, um implementierungsspezifische Schlüsselworte zu entfernen oder ändern (beispielsweise die Speichermodell-Bezeichnungen **_near** und **_far** in 16-Bit-8086-Programmen, die auf eine 32-Bit-Plattform portiert werden sollen). Da der Präprozessor die von ihm bearbeiteten *Tokens* in keiner Weise interpretiert (außer bei der Expansion von Makros), haben auch reservierte Schlüsselworte keine besondere Bedeutung für ihn und werden genauso behandelt wie irgendein anderer Name.

7.2.4. Vordefinierte Makros

Einige Makros sind standardmäßig in jeden C- oder C++-Compiler eingebaut; sie können ohne vorherige explizite Definition verwendet werden:

`__cplusplus` Wird von einem C++-Compiler definiert, wenn er ein C++-Programm übersetzt.

`__DATE__` Übersetzungsdatum der aktuellen Übersetzungseinheit als String-Konstante, z.B. "`Apr 23 1999`".

`__FILE__` Name der Quelldatei als String-Konstante.

`__LINE__` Aktuelle Zeilennummer in der Quelldatei als **int**-Konstante.

`__STDC__` Mit dem Wert `1` definiert, wenn der Compiler den ANSI-C-Standard voll erfüllt.

`__TIME__` Übersetzungszeit der aktuellen Übersetzungseinheit als String-Konstante ((fast) immer in Normalzeit, korrigiert für Sommerzeit nach den in den USA üblichen Regeln).

Weitere vordefinierte Makros geben Type und Version des Compilers, Prozessor-Typ und Betriebssystem, für welche das Programm übersetzt wurde, und ähnliche Informationen an.

Für den GNU C++-Compiler Version 2.57 sind beispielsweise die folgenden Namen vordefiniert:

GNU-C/C++-spezifisch

Name:	Wert:	Name:	Wert:	Name:	Wert:
__GNUC__	2	__GNUG__	2	__cplusplus	1
__GNUC_MINOR__	5	unix	1	i386	1
GO32	1	MSDOS	1	__unix__	1
__i386__	1	__GO32__	1	__MSDOS__	1
__unix	1	__i386	1	__GO32	1
__MSDOS	1				

7.2.5. Die **#include**-Direktive

Die Direktive "**#include**" bewirkt, dass der Präprozessor den Inhalt der mit **#include** angegebenen Datei an der Stelle der **#include**-Direktive einbindet. Für den eigentlichen Compiler erscheint daher der Inhalt der *Include*-Datei (und allenfalls weiterer von dieser ihrerseits mit **#include** eingebundener Dateien) als integraler Bestandteil des Quellprogramms und damit der aktuellen Übersetzungseinheit. Die übliche Anwendung von *Include*-Dateien ist das Einbinden der Definitionen von Konstanten oder Makros, von Definitionen von Funktionen (Prototypen!) und Klassen oder von externen Variablen und Objekten. Dies ist in den folgenden Fällen besonders zweckmäßig:

- Wenn die Definitionen von Bibliotheksfunktionen benötigt werden, die mit dem Compiler ausgeliefert werden (z.B. "`#include <iostream.h>`");
- Wenn ein Programm aus mehreren Übersetzungseinheiten besteht, und diese Übersetzungseinheiten Zugriff auf gemeinsame Konstanten oder Makros, auf globale Daten oder auf die Definition benutzerdefinierter Datentypen benötigen.

GNU-C/C++-spezifisch

Im ersten Fall erspart sich der Programmierer sowohl die Mühe als auch das Risiko von Fehlern, eine Vielzahl von Definitionen in sein Programm aufzunehmen (`iostream.h` und die von ihr inkludierten Dateien umfassen beim GNU C/C++-Compiler zusammen 967 Zeilen an Quellcode). Im zweiten Fall werden vitale Definitionen an *einer* Stelle verwaltet. Wenn beispielsweise im Zuge der Programmentwicklung die Definition einer Klasse geändert werden soll, so braucht diese Änderung nur in *einer* Datei (nämlich in der mit **#include** eingebundenen) vorgenommen zu werden; bei einer neuerlichen Übersetzung aller Übersetzungseinheiten wird die Änderung in *allen* Funktionen des Programms korrekt vorgenommen. (Ein vom Autor betreutes Projekt umfasst zwei Programme mit insgesamt 74 Quelldateien zu insgesamt derzeit mehr als 61.000 Zeilen, deren gemeinsame Definitionen in insgesamt 29 *Include*-Dateien verwaltet werden. Eine andere Vorgangsweise als ein Einbinden der gemeinsamen Informationen mit **#include** wäre hier nicht mehr praktikabel.)

Die Wirkungsweise von **#include** beschränkt die Anwendung dieser Direktive jedoch nicht auf die genannten Fälle: Gelegentlich ist es zweckmäßig, Teile des Programmcodes mit **#include** einzubinden, beispielsweise, wenn mehrere Programme auf gemeinsame Code-Module zugreifen müssen, die Erstellung einer Funktionsbibliothek jedoch nicht sinnvoll oder möglich ist.

Es existieren zwei Varianten der **#include**-Direktive:

#include "*Dateiname*" und
#include <*Dateiname*>

Diese beiden Varianten unterscheiden sich durch die Strategie, mit der der Präprozessor nach der einzubindenden Datei sucht:

- Bei der Form "**#include** "*Dateiname*"" sucht der Präprozessor (in der angegebenen Reihenfolge)
  - im aktuellen Verzeichnis;
  - in den Verzeichnissen, die mit einer Umgebungsvariablen (beim GNU C++-Compiler "`CPLUS_INCLUDE_PATH=`") definiert wurden, in der Reihenfolge ihrer Definition;
  - in den Verzeichnissen, die mit dem Parameter `-I`*`Verzeichnis`* beim Aufruf des Compilers (oder Präprozessors) spezifiziert wurden.

➡ Bei der Form "**#include** <*Dateiname*>" sucht der Präprozessor

- in den Verzeichnissen, die mit einer Umgebungsvariablen (beim GNU C++-Compiler "CPLUS_INCLUDE_PATH=") definiert wurden, in der Reihenfolge ihrer Definition;
- in den Verzeichnissen, die mit dem Parameter -I*Verzeichnis* beim Aufruf des Compilers (oder Präprozessors) spezifiziert wurden.

(also *nicht* im aktuellen Verzeichnis).

Für Dateien, die mit **#include** eingebunden werden sollen, hat sich die Dateinamenerweiterung ".h" (für *Header*) eingebürgert. Es ist diese Konvention aber keinesfalls zwingend.

Gewisse Vorsicht ist bei der Reihenfolge geboten, mit der *Include*-Dateien aufgelistet werden. Die Inhalte aller dieser Dateien werden entsprechend der Reihenfolge der **#include**-Direktiven aneinandergereiht, bevor der eigentliche Compiler die Übersetzungseinheit übergeben erhält. Wenn der Inhalt einer der *Include*-Dateien beispielsweise eine in einer anderen *Include*-Datei enthaltene Definition voraussetzt, dann muss die Datei mit dieser Definition *vor* jener eingebunden werden, die die Definition benötigt.

Beispielsweise könnte eine benutzerdefinierte *Include*-Datei `mydefs.h` etwa den Funktions-Prototyp enthalten:

```
int DateiLesen (FILE *fp, size_t Anzahl);
```

Die Typen `FILE` und `size_t` werden in den Standard-*Include*-Dateien `stdio.h` bzw. `stddef.h` definiert. Wenn daher `mydefs.h` nicht *nach* diesen beiden Dateien eingebunden wird, ist an der Stelle der Definition von `DateiLesen` die Type `FILE` und/oder die Type `size_t` noch nicht definiert, was in einem Fehler resultiert. Die korrekte Reihenfolge der **#include**-Direktiven ist daher:

```
#include <stddef.h>
#include <stdio.h>
#include "mydefs.h"
```

Ungeachtet der durch die Bezeichnung "*Header*-Datei" implizierten Position von **#include**-Direktiven können diese *überall* in einer Übersetzungseinheit stehen, nicht nur an ihrem Anfang.

7.2.6. Bedingte Übersetzung

Die folgenden Präprozessor-Direktiven erlauben eine *bedingte Übersetzung*: Teile eines Programms werden in Abhängigkeit vom Definitions-Status eines Namens (definiert oder nicht) oder von dem ihm zugeordneten Wert bearbeitet oder übersprungen. Ähnlich wie bei der Umsetzung von Namen, die mit **#define** definiert wurden, erfolgt die Ausblendung von Teilen des Programmcodes bereits im Präprozessor; für den eigentlichen Compiler existieren die ausgeblendeten Teile nicht.

Zwei Direktiven — **#if** und **#elif** ("*else if*") — prüfen den *Wert* eines *Ausdrucks* (der zum Zeitpunkt der Bearbeitung des Programms durch den Präprozessor definiert sein muss); wenn dieser Wert von Null verschieden ist, wird der Teil des Quellcodes bis zur nächstgelegenen **#else**-, **#elif**- oder **#endif**-Direktive an den Compiler weitergeleitet, anderenfalls ausgeblendet. Der *Definitions-Status* eines Namens kann mit den Direktiven **#ifdef** und **#ifndef** sowie mit **#if** und dem Präprozessor-Operator **defined** getestet werden; wenn die Bedingung erfüllt ist (also bei **#ifdef** *Name* oder **#if defined** *Name* der Name *Name* definiert ist, bzw. bei **#ifndef** *Name* oder **#if ! defined** *Name nicht* definiert ist), wird wiederum der Teil des Programms bis zur nächsten **#else**-, **#elif**- oder **#endif**-Direktive bearbeitet.

Jedem **#if**, **#ifdef** oder **#ifndef** muss genau *ein* **#endif** entsprechen; dazwischen dürfen beliebig viele **#elif**-Direktiven und allenfalls — als letzte — maximal eine **#else**-Direktive liegen. Innerhalb eines durch **#if**, **#ifdef** oder **#ifndef** einerseits und **#endif** andererseits "aufgespannten" Blocks dürfen weitere **#if** — **#endif**-Blöcke liegen; jede **#elif**- oder **#else**-Direktive in diesen verschachtelten Blöcken bezieht sich auf das nächstgelegene vorhergehende **#if**.

Die in einer **#if**- oder **#elif**-Direktive getesteten Ausdrücke müssen konstant sein. Da Makros auch in Zeilen expandiert werden, die mit **#if** oder **#elif** beginnen, können auch Makros als Bedingung verwendet werden. Es gelten die folgenden Einschränkungen für die mit **#if** oder **#elif** verwendeten Ausdrücke:

- Sie müssen ganzzahlig sein und dürfen nur ganzzahlige Konstanten, Zeichenkonstanten und den Operator **defined** enthalten.
- Die Verwendung von *Type Casts* sowie des Operators **sizeof** ist unzulässig.
- Numerische Werte werden intern als **long** oder **unsigned long** dargestellt.

Das folgende Demo-Programm gibt unterschiedliche Meldungen aus, je nachdem, ob und mit welchem Wert der Name TEST definiert wurde. Eine Definition kann beim Aufruf des Compilers mit dem Befehlszeilen-Parameter "-DTEST", eine Wertezuweisung mit "-DTEST=*Wert*" erfolgen. (Im letzteren Fall muss der Aufruf von GNU C++ statt über das Stapelprogramm GPP.BAT direkt erfolgen, weil die Befehlszeileninterpretation in COMMAND.COM das "=" als Separator (wie ein Leerzeichen) interpretieren würde.)

Demo-Programm 7_02_07.cc

```
#include <iostream.h>

int main ()
{
#if ! defined (TEST)                              // TEST nicht definiert
    cout << "Versuchen Sie, das Programm mit\n\"gcc 7_02_07.cc -Wall -lgpp "
        "-g -o 7_02_07 -DTEST\"\nzu übersetzen!\n";
#else                                             // TEST definiert
#if TEST < 0
    cout << "Das Programm wurde mit einem Wert < 0 für TEST übersetzt.\n";
#elif TEST > 1
    cout << "Das Programm wurde mit TEST > 1 übersetzt.\n";
#elif TEST == 0
    cout << "Das Programm wurde am " __DATE__ " um " __TIME__ " Uhr "
        "übersetzt.\n";
#else                       // definiert, aber ohne Wertzuweisung (TEST == 1)
    cout << "Versuchen Sie jetzt, das Programm mit\n\"gcc 7_02_07.cc -Wall "
        "-lgpp -g -o 7_02_07 -DTEST=0\"\nund dann mit anderen Werten für "
        "TEST zu übersetzen!\n";
#endif                                            // Ende von "#if TEST < 1"
#endif                                            // Ende von "#if ! defined TEST"
#if 0
    cout << "Diese Ausgabe werden Sie nie zu sehen bekommen!\n";
#else
    cout << "\nSie werden diese Zeile immer erhalten!\n";
#endif
}
```

Das Beispiel demonstriert auch, wie Teile eines Programms permanent deaktiviert werden können. Im Gegensatz zum "Auskommentieren" von Programmteilen funktioniert **#if** 0 auch dann, wenn beliebig viele Kommentare (in "**/* */**"- oder "**//**"-Syntax) im deaktivierten Code enthalten sind.

Je nachdem, wie der Compiler bei der Übersetzung des Demo-Programms aufgerufen wurde, erhalten wir als Ausgabe:

- Übersetzt mit "gpp 7_02_07":

```
Versuchen Sie, das Programm mit
"gcc 7_02_07.cc -Wall -lgpp -g -o 7_02_07 -DTEST"
zu übersetzen!

Sie werden diese Zeile immer erhalten!
```

- Übersetzt mit "gcc ... -DTEST":

```
Versuchen Sie jetzt, das Programm mit
"gcc 7_02_07.cc -Wall -lgpp -g -o 7_02_07 -DTEST=0"
und dann mit anderen Werten für TEST zu übersetzen!

Sie werden diese Zeile immer erhalten!
```

➡ Übersetzt mit "`gcc ... -DTEST=0`":

```
Das Programm wurde am Jun 11 1994 um 14:56:12 Uhr übersetzt.

Sie werden diese Zeile immer erhalten!
```

➡ Übersetzt mit "`gcc ... -DTEST=-10`":

```
Das Programm wurde mit einem Wert < 0 für TEST übersetzt.

Sie werden diese Zeile immer erhalten!
```

➡ Übersetzt mit "`gcc ... -DTEST=10`":

```
Das Programm wurde mit TEST > 1 übersetzt.

Sie werden diese Zeile immer erhalten!
```

Beachten Sie bitte, dass dann, wenn ein Name beim Aufruf des Compilers mit "`-D`*`Name`*" definiert, ihm aber explizit kein Wert zugewiesen wurde, der Name *`Name`* vom Präprozessor durch "`1`" ersetzt wird. Hingegen entspricht eine mit **`#define`** erstellte leere Definition einem leeren String.

Eine häufige Anwendung der bedingten Übersetzung findet sich in den mit einem C- oder C++-Compiler ausgelieferten *Include*-Dateien: Aus Gründen der Effizienz ist es zweckmäßig, die Deklarationen gewisser zusammengehöriger Funktionen und Klassen in separaten *Include*-Dateien zusammenzufassen. (Jeder Name, der in einer Übersetzungseinheit deklariert wird, benötigt für seine Verwaltung bei der Übersetzung Platz, selbst dann, wenn er ansonsten gar nicht verwendet wird. Da die Ressourcen eines Compilers begrenzt sind, empfiehlt es sich, möglichst nur jene Funktionen zu deklarieren, die auch wirklich benötigt werden.) Gewisse Definitionen (z.B. mit **`typedef`**) müssen dabei jedoch in *mehreren Include*-Dateien vorgesehen werden, weil sie von den in diesen *Include*-Dateien enthaltenen Deklarationen vorausgesetzt werden. Andererseits darf ein Name aber nur *einmal* pro Übersetzungseinheit mit **`typedef`** definiert werden; wenn zufällig *mehrere Include*-Dateien in eine Übersetzungseinheit eingebunden würden, die die gleiche Type definieren, hätte dies einen Fehler zur Folge.

Abhilfe ist durch die Definition von Hilfs-Namen möglich, die in jeder *Include*-Datei auf ihren Definitions-Status hin getestet werden und das Einbinden des **`typedef`**-Befehls steuern. Das folgende Beispiel stammt vom Microsoft-Visual C++-Compiler, wo diese Technik wesentlich häufiger und konsequenter verwendet wird als bei GNU C++:

```
#ifndef _SIZE_T_DEFINED
typedef unsigned int size_t;
#define _SIZE_T_DEFINED
#endif
```

Diese Sequenz kommt außer in der Datei `stddef.h` (wo sie auf jeden Fall stehen sollte) noch in insgesamt 9 weiteren *Include*-Dateien dieses Compilers vor. Einzelne oder alle 10 Dateien können in beliebiger Reihenfolge mit **`#include`** eingebunden werden. Bei der ersten Datei, die `size_t` definiert, ist `_SIZE_T_DEFINED` noch undefiniert; der **`typedef`**-Befehl wird dort ausgeführt und `_SIZE_T_DEFINED` definiert. Folglich wird in allen nachfolgenden *Include*-Dateien der **`typedef`**-Befehl übersprungen; `size_t` wird daher genau einmal definiert.

7.2.7. Sonstige Direktiven

7.2.7.1. Die Direktive `#line`

Die vordefinierten Makros `__FILE__` und `__LINE__` repräsentieren den Namen der aktuellen Datei und die Nummer der aktuellen Zeile in dieser Datei. Diese Information kann mit der Direktive **`#line`** gezielt verändert werden; die Zeilennummern werden ausgehend von der mit **`#line`** angegebenen Nummer weitergezählt. Für diese Direktive (die im allgemeinen nur von Codegeneratoren verwendet wird, bei denen sich Fehlermeldungen auf eine Zeile im ursprüngli-

chen und nicht im maschinell erstellten Code beziehen sollen) sind die folgenden beiden Formen möglich:

#line *GanzeZahl* und

#line *GanzeZahl* "*Dateiname*"

Das folgende Beispiel illustriert diese Direktive:

Demo-Programm 7_02_08.cc

```
#include <iostream.h>

#define MELDE cout << "Zeile " << __LINE__ << " in Datei " << __FILE__ << endl;

int main()
{
    MELDE
    MELDE

#line 1000 "TestProg"
    MELDE
    MELDE

#line 2000
    MELDE
    MELDE
}
```

Der Präprozessor setzt die Funktion `main()` um in:

```
int main()
{
    cout << "Zeile " << 11 << " in Datei " << "7_02_08.cc" << endl;
    cout << "Zeile " << 12 << " in Datei " << "7_02_08.cc" << endl;

# 999 "TestProg"
    cout << "Zeile " << 1000 << " in Datei " << "TestProg" << endl;
    cout << "Zeile " << 1001 << " in Datei " << "TestProg" << endl;

# 1999 "TestProg"
    cout << "Zeile " << 2000 << " in Datei " << "TestProg" << endl;
    cout << "Zeile " << 2001 << " in Datei " << "TestProg" << endl;
}
```

Dementsprechend ist die Ausgabe des Programms:

```
Zeile 11 in Datei 7_02_08.cc
Zeile 12 in Datei 7_02_08.cc
Zeile 1000 in Datei TestProg
Zeile 1001 in Datei TestProg
Zeile 2000 in Datei TestProg
Zeile 2001 in Datei TestProg
```

7.2.7.2. Die Direktive **#error**

Mit der Direktive **#error**, der ein beliebiger String folgen kann, wird die Übersetzung eines Programms abgebrochen, wobei der mit der Direktive angegebene String ausgegeben wird. Typisch wird diese Direktive dazu verwendet, um nach Prüfung eines vordefinierten Makros die Übersetzung bei Bedarf abzubrechen:

Demo-Programm 7_02_09.cc

```
#if defined __unix__
#error Tu nix mit UNIX!
#endif
```

Mit dem GNU C++-Compiler übersetzt, bewirkt dieses Programm tatsächlich einen Abbruch und die Ausgabe des Kernspruchs:

```
7_02_09.cc:5: #error Tu nix mit UNIX!
```

Hingegen wird das selbe Programm mit einem Microsoft-Compiler, für den der Name `__unix__` nicht definiert ist, fehlerfrei übersetzt.

7.2.7.3. Die **#pragma**-Direktiven

#pragma-Direktiven haben grundsätzlich die Form

#pragma *Token-String*

Sie dienen dazu, um implementierungsspezifische Anpassungen der Funktionen des Compilers vorzunehmen. Im Gegensatz zu Befehlszeilen-Parametern des Compilers, die immer die gesamte Übersetzungseinheit beeinflussen, können sie auch lokale Veränderungen des Verhaltens des Compilers in Teilen einer Übersetzungseinheit erlauben.

In GNU C++ sind relativ wenige **#pragma**-Direktiven definiert; von Bedeutung sind nur **#pragma interface** und **#pragma implementation**. Eine detaillierte Diskussion der Bedeutung dieser Direktiven ist in der GNU C++ Compiler Info (Programm `INFO`, Datei `gcc.inf`) unter dem Titel "*Declarations and Definitions in One Header*" enthalten.

Andere Compiler erlauben eine große Zahl von unterschiedlichen **#pragma**-Direktiven. Informationen über Syntax und Funktion dieser Direktiven müssen der jeweiligen Dokumentation entnommen werden.

8. Literaturhinweise

Quellen:

Bjarne STROUSTRUP:
The C++ Programming Language, Second Edition
Addison-Wesley Publishing Company, 1991
669 p.

MICROSOFT CORPORATION:
Microsoft Visual C++ — C++ Language Reference
Microsoft Corporation, 1993
468p.

D.J. DELORIE et al.:
GNU C++ Compiler On-Line Info
D.J. Delorie & Free Software Foundation, 1993

MICROSOFT CORPORATION:
Microsoft Visual C++ — Run Time Library Reference
Microsoft Corporation, 1993
688p.

MICROSOFT CORPORATION:
Microsoft Visual C++ — `iostream` Class Library Reference
Microsoft Corporation, 1993
145p.

MICROSOFT CORPORATION:
Microsoft Quick C — Run Time Library Reference
Microsoft Corporation, 1987
687p.

Herbert SCHILDT:
C — The Complete Reference
Osborne McGraw-Hill,1987
773p.

Charles PETZOLD:
Programming Windows 3.1, Third Edition
Microsoft Press, 1992
983p.

Weiterführende Literatur (Algorithmen und Programmiertechniken):

Donald E. KNUTH:
The Art of Computer Programming — Vol. 1: Fundamental Algorithms
Addison Wesley, 1973
634p.

Donald E. KNUTH:
The Art of Computer Programming — Vol. 2: Seminumerical Algorithms
Addison Wesley, 1981
688p.

Donald E. KNUTH:
The Art of Computer Programming — Vol. 3: Sorting and Searching
Addison Wesley, 1973
723p.

Jack J. PURDUM, Timothy C. LESLIE, Alan L. STEGEMOLLER:
C Programmer's Library
Que Corporation, 1984
366p.

W. FEIBEL:
Using Quick C
Osborne McGraw-Hill, 1988
606p.

Mitchell WAITE, Stephen PRATA, Donald MARTIN:
C Primer Plus
Howard W. Sams & Co., Inc., 1985
531p.

Ed TILEY:
Tricks of the Windows 3.1 Masters
Sams, 1992
952p.

David J. KRUGLINSKI:
Inside Visual C++
Microsoft Press, 1993
598p.

Index

ISBN: 3-901578-05-6