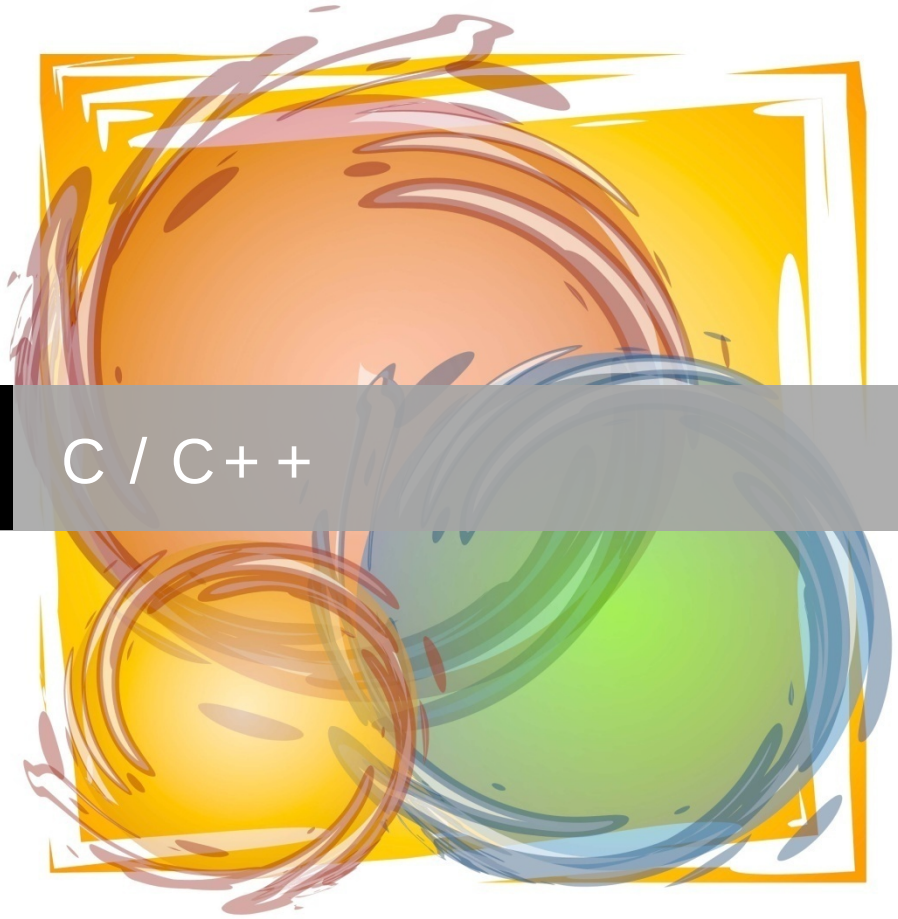
Heiko Gorski

C / C++ für Einsteiger



C / C++

VORWORT ZUR 17. AUSGABE

An diesem Skript haben, über die Jahre verteilt, viele Menschen mitgewirkt – Freunde, Bekannte, Kollegen und natürlich Kursteilnehmer. Alle haben mich eifrig mit Verbesserungsvorschlägen – und nach mühseliger Detektivarbeit auch mit Fehlerkorrekturen versorgt. Ihnen allen gilt mein Dank für die wertvolle Unterstützung.

Ein ganz besonderer Dank gebührt meiner Frau, die es seit Jahren erträgt, dass ich jeden Urlaub und viele Wochenenden dazu nutze, dieses Skript zu überarbeiten und zu erweitern, so dass es von seiner „Erstausgabe“ mit knapp 20 Seiten auf den heutigen Umfang wachsen konnte.

Dipl. Inform Heiko Gorski, Januar 2012

INHALTSVERZEICHNIS

1. EINLEITUNG

C++ ist im Gegensatz zu den klassischen, prozeduralen Varianten ANSI[1]-C und dem noch älteren K&R[2]-C eine objektorientierte Programmiersprache. Mit dem Begriff C werden im Folgenden immer beide Varianten (ANSI und K&R) bezeichnet, sofern nicht ausdrücklich anders angegeben.

1.1. EFFEKTIVER PROGRAMMIEREN MIT C++

Üblicherweise besteht hinsichtlich der objektorientierten Programmiersprachen der Irrglauben, dass diese grundsätzlich leichter zu erlernen, effektiver einzusetzen und produktiver seien.

In solcher Verallgemeinerung ist diese Behauptung schlicht falsch. Selbstverständlich hat C++ gegenüber C eine ganze Reihe von Vorteilen (insbesondere eine erheblich geringere Fehleranfälligkeit), die Produktivität des einzelnen Programmierers wird aber keinesfalls kurzfristig verbessert, denn vorher müssen die benötigten Objekte korrekt und möglichst vollständig beschrieben und implementiert werden. Allein nur durch eine vollständige Abbildung lassen sich Objekte einmalig Programmieren und anschließend ohne großen Änderungsaufwand immer wieder verwenden. So muss zum Beispiel ein Entwickler, Welcher eine neue Klasse „Datum“ schreibt, sich darüber Gedanken machen, in welchen Zusammenhängen dieses „Datum“ benutzt werden kann. Implementiert der Entwickler die Klasse nur teilweise, so wird er bei jeder neuen Anforderung gezwungen sein, die Klasse zu erweitern oder zu ändern. Die Objektorientierung legt dem Entwickler somit ein umfassenderes und vollständigeres Arbeiten nahe. Glücklicherweise wird der Entwickler in C++ in dieser Hinsicht auch wesentlich besser bei dieser Arbeit unterstützt. Trotzdem erfordert diese Art des Vorgehens zu Beginn mehr Zeit als der prozedurale Ansatz.

Der Effekt „Zeitersparnis“ trifft also nur sehr bedingt zu – ansonsten hätten sich die „reinen“ objektorientierten Sprachen auch schon Mitte der 80er Jahre durchsetzen müssen. SMALLTALK, eine der ersten objektorientierten Sprachen, war schon damals sehr gut entwickelt und auf Workstations verfügbar. Trotz dieser Verfügbarkeit (auch auf Großrechnern) wurde SMALLTALK nicht „die“ Programmiersprache, was sicherlich auch an den erheblich höheren Hardwareanforderungen lag, so dass die objektorientierten Sprachen erst in der ersten Hälfte der 90er Jahre ihren Siegeszug begonnen haben.

Zumindest scheint es so zu sein, denn es ist zu beachten, dass C++ (neben Java der bedeutendste objektorientierte Vertreter auf dem PC-

1 American National Standardization Institute

2 Kernighan & Ritchie

Sektor) keine „reine“ objektorientierte Sprache ist, sondern in ihrem Ursprungeine prozedurale Programmiersprache mit objektorientierten Elementen. Dies wird besonders deutlich, wenn man sich die Entwicklungsgeschichte von C++ betrachtet.

1.2. DIE ENTSTEHUNGSGESCHICHTE VON C UND C++

K&R-C wurde Anfang der siebziger Jahre von den beiden Programmierern Dennis M. Ritchie und Brian W. Kernighan auf Grundlage der Programmiersprachen ALGOL, BCPL und B entwickelt. Anlass für die Entwicklung war der Wunsch, eine handliche Sprache für die Umsetzung des Betriebssystems UNIX zu schreiben. Die erzeugten Programme sollten möglichst klein, schnell und effizient sein, ohne dabei die Unhandlichkeit von Assemblersprachen zu aufzuweisen.

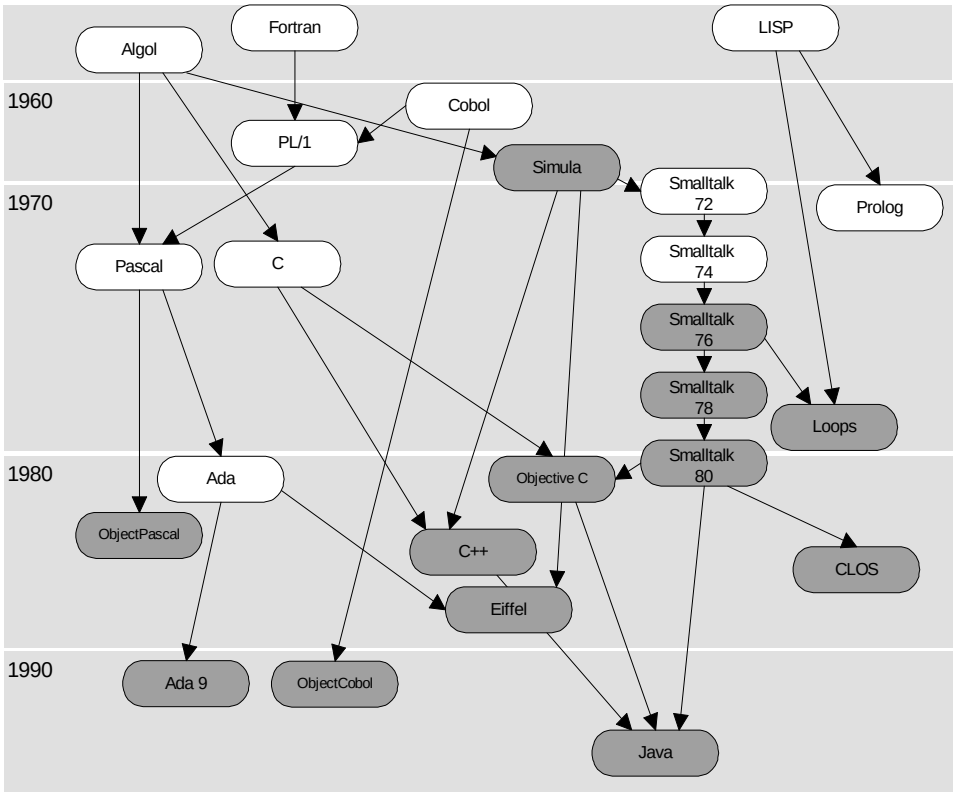


Abbildung 1-1 – Stammbaum der OOP-Sprachen

Das Ergebnis war fast zwangsläufig eine Sprache, die im Quellcode sowohl (bestimmte) Befehle in Assembler beinhalten, wie auch Elemente einer Hochsprache verarbeiten kann. Vorteilhaft ist dabei die Tatsache, dass die Programme trotzdem recht gut lesbar und unabhängig von der Hardware sind. Nachteilig wirkt sich an manchen Stellen die geringe Abstraktion von der Rechnerarchitektur aus. So ist es z.B. unmöglich, in C zu programmieren, ohne sich intensiv mit Adressen und Zeigern (Pointern) auseinander zu setzen. In C finden sich aber auch deutliche Einflüsse der damals sehr aktuellen und verbreiteten Sprachen PL1 und Algol68. Diese Einflüsse lassen sich z.B. bei den Ein- und Ausgabe-

funktionen nachvollziehen, die wesentlich flexibler und mächtiger sind, als die entsprechenden Instruktionen in der zur Entstehungszeit von C häufig verwendeten Sprache PASCAL.

Der Begriff C++ findet sich erstmals 1986 bei Bjarne Stroustrup, der den eigentlichen Compiler mit einem zusätzlichen Präprozessor versah, der die neuen C++ Sprachelemente (d.h. insbesondere die Klassen) in reinen, prozeduralen (ANSI-C) Code umwandelte und anschließend erst übersetzte. Stroustrup erzielte damit den Vorteil, dass man nicht nur in C++ geschriebene Programme übersetzen konnte, sondern weiterhin auch Quellen in ANSI-C. Hinzu kam die Fähigkeit des Compilers, auch gemischten Code korrekt übersetzen zu können – die Programmierer konnten also ihre Arbeitsweise langsam umstellen und die objektorientierten Teile dort einsetzen, wo sie Vorteile versprachen. Genau dieser Ansatz hat C++ zum großen Durchbruch verholfen und so findet man in der Praxis auch nur wenige Programme, die rein objektorientiert sind, sondern fast ausschließlich mehr oder weniger ausgeprägte Mischformen. Einige Programmierer nutzen kaum mehr als die C++ Kommentare, die große Mehrheit hingegen versucht einen Großteil der Daten zu eigenen Objektklassen zusammenzufassen. Trotzdem finden sich fast immer prozedurale Teile, wenn diese einfacher zu realisieren sind[3].

Nachdem die C++ Teile ebenfalls in den ANSI-Standard übernommen wurden, sind sie nun integrale Bestandteile des Compilers.

1.3. PORTABILITÄT

Bereits das K&R-C war in wesentlichen Teilen standardisiert. Dieser Standard wies jedoch noch große Lücken auf, so dass sich bereits nach kurzer Zeit mehrere, leicht unterschiedliche Dialekte der Sprache gebildet hatten. Die Aufspaltung in Dialekte drohte einen der Hauptvorteile von C, die relativ gute Portabilität von Programmen, zunichte zu machen. Deshalb wurde der Sprachstandard Anfang der achtziger Jahre vom ANSI überarbeitet, erweitert und an die neuesten Konventionen und Entwicklungen angepasst. Auch heute noch ist das ANSI-C eine sehr leistungsfähige Programmiersprache. Ihre breite Verfügbarkeit auf nahezu allen Hardwareplattformen (von PCs über Workstations bis hin zu Großrechnern) hat zu ihrer schnellen Verbreitung beigetragen. Da der eigentliche Kern (Kernel) nur sehr wenige Befehle enthält, sind auch Compiler relativ einfach zu entwickeln. Die meisten Befehle, die in C benutzt werden sind Unterprogramme, die aus Befehlen des Kerns

3 Gelegentlich sind diese prozeduralen Teile natürlich nur deshalb leichter zu realisieren, weil sie dem Entwickler vertrauter sind als die neueren, objektorientierten Ansätze.

bestehen. D.h. es genügt, die Kernbefehle zu implementieren und man kann dann alle anderen Befehle mit dem Compiler erzeugen.

Gleiches gilt für C++, welches sich zur wichtigsten Programmiersprache der 90er Jahre entwickelt hatte. Moderne Herausforderer (wie z.B. Java) basieren zweifelsfrei und deutlich auf der Syntax von C und C++ und fügen üblicherweise nur wenige neue Konzepte hinzu.

1.4. VON DER SCHWIERIGKEIT SOFTWARE ZU PRODUZIEREN

C ist eine prozedurale Programmiersprache, d.h. ein Programm, welches seine Aufgabe durch schrittweise Veränderung von Daten bewältigt. Die Struktur des zu schreibenden Programms orientiert sich daher an der Reihenfolge der Arbeitsschritte, die notwendig sind, um aus einer Menge von Eingabedaten die gewünschten Ausgabedaten zu erzeugen.

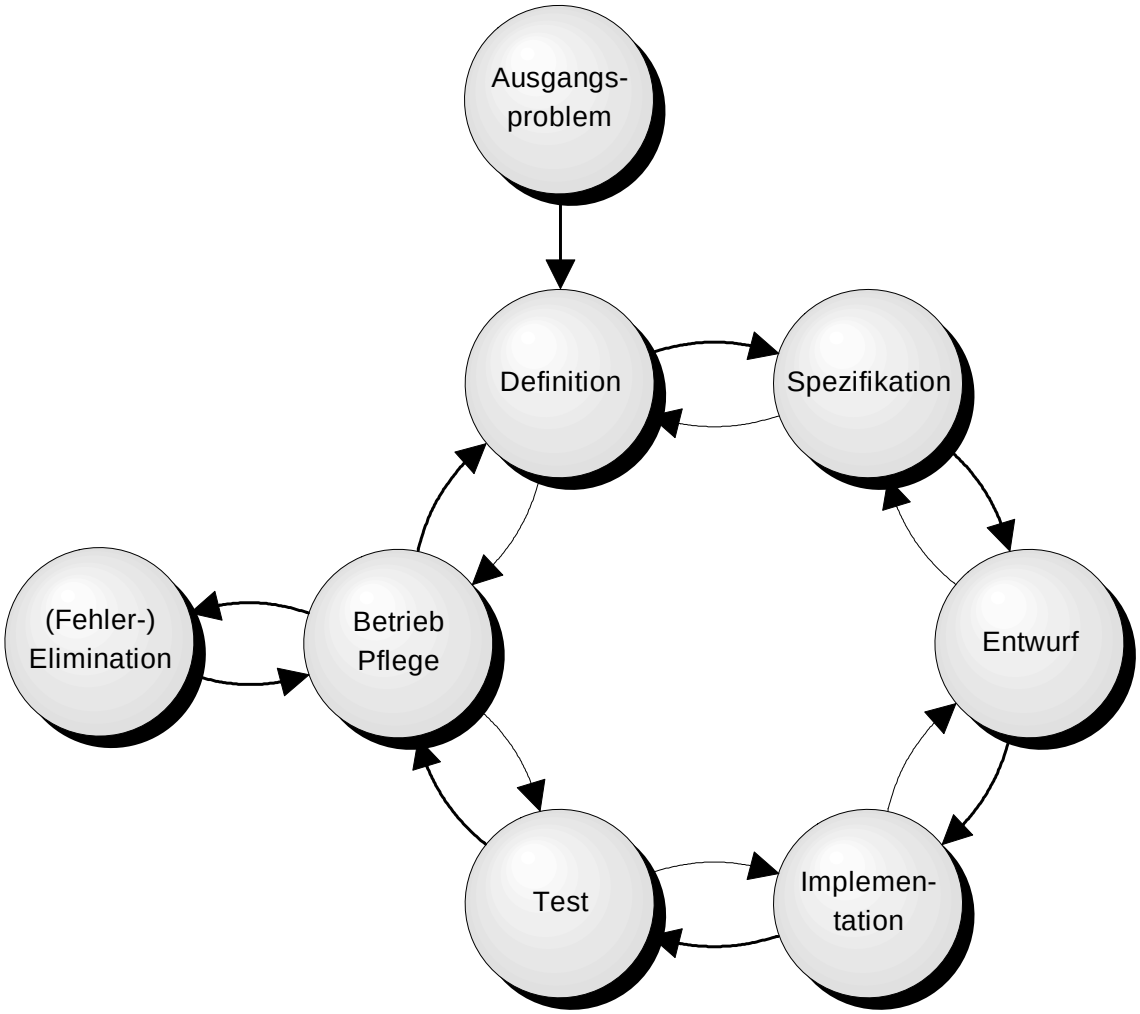


Abbildung 1-2 – Entwicklungszyklus

Die Hauptaufgabe des Programmierers ist es, die notwendigen Arbeitsschritte (d.h. die Programmbefehle bzw. Instruktionen oder Anweisungen) festzulegen und ihre Reihenfolge logisch aufeinander

aufbauen zu lassen. Dabei steht der Programmierer vor einer ganzen Reihe von zu lösenden Problemen.

C++ ist eine objektorientierte Programmiersprache, d.h. anders als bei der prozeduralen Vorgehensweise stehen nicht die Arbeitsabläufe im Vordergrund sondern die Daten, die miteinander in Beziehung gesetzt werden. Der Unterschied mag zunächst unerheblich erscheinen, erfordert jedoch gänzlich andere Vorgehensweisen bei der Implementation. Das generelle Vorgehen bei der Softwareentwicklung bleibt davon jedoch unbenommen:

Ist das zu lösende Problem erkannt und beschrieben worden (Definitionsphase), so muss zunächst der Umfang und die genaue Vorgehensweise bei dieser Aufgabe genau durchdacht und analysiert werden. Es muss zu Beginn festgestellt werden, welche der notwendigen Eingabedaten vorhanden sind bzw. ermittelt werden müssen und welche Ergebnisdaten während des Programmlaufs zu ermitteln sind (Spezifikationsphase).

Oftmals lassen sich nicht alle Daten von vornherein zur Verfügung stellen, dann muss das Programm entsprechend flexibel sein. Während des Programmablaufes sind ggf. notwendige Entscheidungen vom Anwender abzufordern usw. Daten müssen auf ihre Gültigkeit hin geprüft werden, (es muss während des Programmlaufes festgestellt werden, ob die berechneten bzw. eingegebenen Daten überhaupt Sinn machen (Plausibilitätskontrolle). Aus diesen Notwendigkeiten heraus entsteht der erste Entwurf des Systems (Entwurfsphase). Dieser Teil der Entwurfsphase wird auch als Datenmodellierung (oder Datenanalyse) bezeichnet.

Die von einer Programmiersprache zur Verfügung gestellten Instruktionen sind fast immer sehr einfach und elementar, d.h. sie sind noch nicht auf einen bestimmten Aufgabenbereich angepasst. Für den Programmierer bedeutet dies, dass er auch für relativ einfache Aufgaben eine zumeist stattliche Anzahl von elementaren Anweisungen erteilen muss. Hinzu kommt, dass ein Computer nicht „Mitdenken“ kann, sondern alle erteilten Anweisungen abarbeitet, egal ob sie nun offensichtlich falsch sind oder nicht. Im zweiten Teil der Entwurfsphase werden die einzelnen Arbeitsabschnitte analysiert und daraus eine grobe Ablaufstruktur abgeleitet, die man später bei der Programmierung als Grundgerüst heranziehen kann. Diese Phase wird sehr häufig als Funktionsanalyse bezeichnet. Werden anschließend Daten- und Funktionsmodell zusammengebracht, entdeckt man zumeist die ersten Entwurfsfehler. Entsprechend der üblichen Vorgehensweise müssen nun Daten- und Funktionsmodelle solange korrigiert und erneut zusammengeführt werden, bis der Entwurf geeignet zu sein scheint.

Die allgemeine Vorgehensweise bei der Softwareentwicklung ist hier äußerst verkürzt dargestellt. Im Kurs wird sie auch nicht angewendet werden, da die Aufgaben sehr einfach sind und die Lösungsprogramme klein sein werden. Dennoch ist es wichtig, dieses grundsätzliche Vorgehensschema zumindest zu verstehen, um später den Aufbau von komplexen Programmen besser bewältigen zu können. Empfehlenswert ist eine solche strukturierte Vorgehensweise allemal.

1.5. VERWENDETE TEXTFORMATIERUNGEN

In den nachfolgenden Kapiteln werden eine Reihe von speziellen Textformatierungen und Symbolen verwendet, um bestimmte Sachverhalte (Programmtexte, Hinweise, Syntax etc.) zu kennzeichnen. Um diese einzelnen Kennzeichnungen leichter zu verstehen, sind alle verwendeten Markierungsarten nachfolgend aufgezeichnet:

Tipps, Tricks und Merksätze, welche die Programmierung ein wenig leichter machen.

Programmtext oder Beispiel

Programmtext oder Beispiel.
Quelltext auf Diskette verfügbar

Programmtext oder Beispiel (der Blitz deutet an, dass der Quelltext typische Programmfehler enthält, die zu Abstürzen führen können)

```
Programmsyntax und / oder Syntaxhinweis
```

2. DER AUFBAU VON C/C++ PROGRAMMEN

Der grundsätzliche Programmaufbau in C/C++ folgt einem relativ einfachen Schema, welches in allen C/C++ Teilen (also auch externen Modulen) gleich ist. Zu Beginn eines Programmes finden sich eine Reihe von Anweisungen für den Präprozessor, erkennbar am Zeichen „#“. Dem Präprozessor ist ein eigenes Kapitel gewidmet, die genaue Syntax der Befehle und ihre Bedeutung finden Sie dort aufgeführt. Nach den Anweisungen für den Präprozessor folgen die globalen Vereinbarungen von Daten und Unterprogrammen. Die entsprechenden Erklärungen dazu finden Sie in den Kapiteln über die Datentypen und Unterprogrammtechniken. An dieser Stelle soll nur die Bedeutung und der Aufbau des Hauptprogramms erklärt werden, welches in C/C++ immer den Namen *main* trägt. Im Gegensatz zu anderen Programmiersprachen hat das Hauptprogramm in C/C++ keinen festen Platz (in PASCAL z.B. muss es an letzter Stelle stehen), da es durch den eindeutigen Namen ausreichend gekennzeichnet ist. Die Verarbeitung von Programmanweisungen beginnt immer mit der ersten Anweisungszeile von *main*. Das Hauptprogramm hat typischerweise folgendes Aussehen:

```
void main (void)
{
}
```

Das Schlüsselwort *void* vor dem Namen des Hauptprogramms besagt, dass keine Daten an das aufrufende Programm zurückgegeben werden (meistens das DOS, wo zurückgegebene Werte als Errorlevel interpretiert werden). Das zweite *void*, in runden Klammern, bedeutet, dass zudem keine Daten von vom aufrufenden Programm bzw. vom DOS übernommen werden. Wie die Datenkommunikation mit der DOS-Ebene genau funktioniert ist Thema des Kapitels 21 und wird daher an dieser Stelle nicht erläutert.

Der Anweisungsteil wird mit geschweiften Klammern gekennzeichnet, die öffnende Klammer leitet den Anweisungsteil ein, die schließende Klammer beendet ihn.

2.1. KOMMENTARE

Beachten Sie bitte auch den Quelltext-Kommentar, der durch die Zeichenkombination „/*“ eingeleitet und durch „*/“ beendet wird. Ein solcher Kommentar kann über mehrere Zeilen reichen. Quelltext-Kommentare können aber in den meisten C/C++ Compilern nicht verschachtelt werden!

Alternativ kann bei vielen Compilern auch die Zeichenkombination „//“ verwendet werden (Restzeilenkommentar aus C++), die lediglich den Rest der Zeile als Kommentar markiert. Mit dem ANSI-C Standard 99

steht der Restzeilenkommentar auch im Standard-C offiziell zur Verfügung.

```
void main (void)
{
   /* Kommentar, der über mehrere Zeilen
      gehen kann */
   // C++ - Kommentar bis zum Zeilenende
}
```

```
void main (void)
{
   /* mehrzeilige Kommentare dürfen bei den meisten Compilern
      /* nicht */
      verschachtelt werden */
}
```

2.2. DER ÜBERSETZUNGSVORGANG

C/C++ ist eine Compilersprache, d.h. der vom Programmierer erstellte Programmcode wird in Maschinensprache übersetzt. Die Übersetzung in Maschinensprache, eine für den Menschen nicht verständliche Folge von Befehlscodes, unterscheidet Compilersprachen von Interpretersprachen (viele BASIC-Dialekte, DBase, FoxPro, Comal usw.) und P-Code-Compilern (Java, Pascal, Clipper u.a.).

Interpreter lesen den erstellten Quelltext Zeichen für Zeichen und führen einen Befehl aus, sobald sie ihn vollständig gelesen und eindeutig erkannt haben. Der Vorteil liegt in der größeren Interaktivität, man kann einige Zeilen schreiben, das Programm sofort ausführen und bei einem Fehler hält das Programm exakt an der fehlerhaften Stelle an. Diese ändert man, startet das Programm erneut usw. Nachteilig ist die geringe Ausführungsgeschwindigkeit, die Interpretersprachen für die meisten kommerziellen Anwendungen (insbesondere solche mit aufwendigen Oberflächen) ungeeignet macht.

Compilersprachen hingegen lassen eine Interaktivität im Grunde nicht zu, das ausführbare Maschinenprogramm hat keinerlei Verbindung zum ursprünglichen Quelltext mehr, lässt ein zurückgreifen auf eine ggf. zugrundeliegende, fehlerhafte Programmzeile nicht zu. Auf die Fehlerquelle muss anhand der gewählten Funktion und des Absturzmomentes geschlossen werden. Diesem Nachteil steht eine sehr hohe Ausführungsgeschwindigkeit gegenüber, die für komplexe Programme unverzichtbar ist. Sogenannte Sourcelevel-Debugger schlagen eine Brücke zwischen Geschwindigkeit und Interaktivität. Der Compiler wird angewiesen, den Quellcode in Bezug zum ausführbaren Code zu setzen und ein spezielles Zwischenformat zu erzeugen. Nun ist die Anzeige des aktuell abgearbeiteten Befehls im Quellcode möglich und

Variableninhalte können online abgefragt werden. Dieses Zwischenformat eignet sich allerdings nicht als auszulieferndes Programm, da hier bereits erhebliche Abstriche hinsichtlich der Geschwindigkeit hinzunehmen sind.

Wie C/C++ Programme übersetzt:

Der Quelltext wird eingelesen und an den Präprozessor weitergeleitet. Dieser führt Textersetzungen durch - Die Textersetzungs-Makros und symbolischen Konstanten [gekennzeichnet durch #define] werden aufgelöst und Header-Dateien [gekennzeichnet durch #include] zugespielt.

Der Compiler nimmt den vom Präprozessor bearbeiteten Text und übersetzt ihn in ein spezielles Zwischenformat, den sogenannten Object-Code. Dabei wird der Programmtext auf syntaktische Korrektheit geprüft (sind alle Befehle richtig geschrieben, alle Variablen vereinbart, werden alle Funktionen mit korrekten Parametern aufgerufen usw.).

Im letzten Schritt wird der Linker aufgerufen, der - sofern das Programm aus mehreren Teilen besteht - die Programm-Module zusammenfügt und die Bibliotheksreferenzen auflöst und schließlich das ausführbare Programm erstellt.

Die Bibliotheken sind eines der Erfolgsrezepte von C/C++, sie erlauben es, den Sprachumfang an die Bedürfnisse des erstellten Programms anzupassen, so dass kein unnötiger Programmballast in das ausführbare Programm gelangt.

Je nach Hersteller oder vom Entwickler getätigten Einstellungen können während der Übersetzungsvorgangs zusätzliche Dateien erzeugt werden. Außerdem werden bei den meisten Compilern einige Verwaltungsdateien automatisch erzeugt, die nur für die Oberflächensteuerung (z.B. Position der geöffneten Fenster) benötigt werden.

Andere Dateien werden nur erzeugt, wenn die entsprechenden Schalter am Compiler gesetzt sind, sie dienen dem Entwickler als zusätzliche Informationsquellen bei der Fehlersuche (z.B. sogenannte MAP-Dateien, die Auskunft über die gelinkten Funktionen und ihre Programmadressen geben).

Die Abbildung zeigt den typischen, minimalen Umfang und Verlauf eines Übersetzungsvorgangs durch einen C/C++ Compiler.

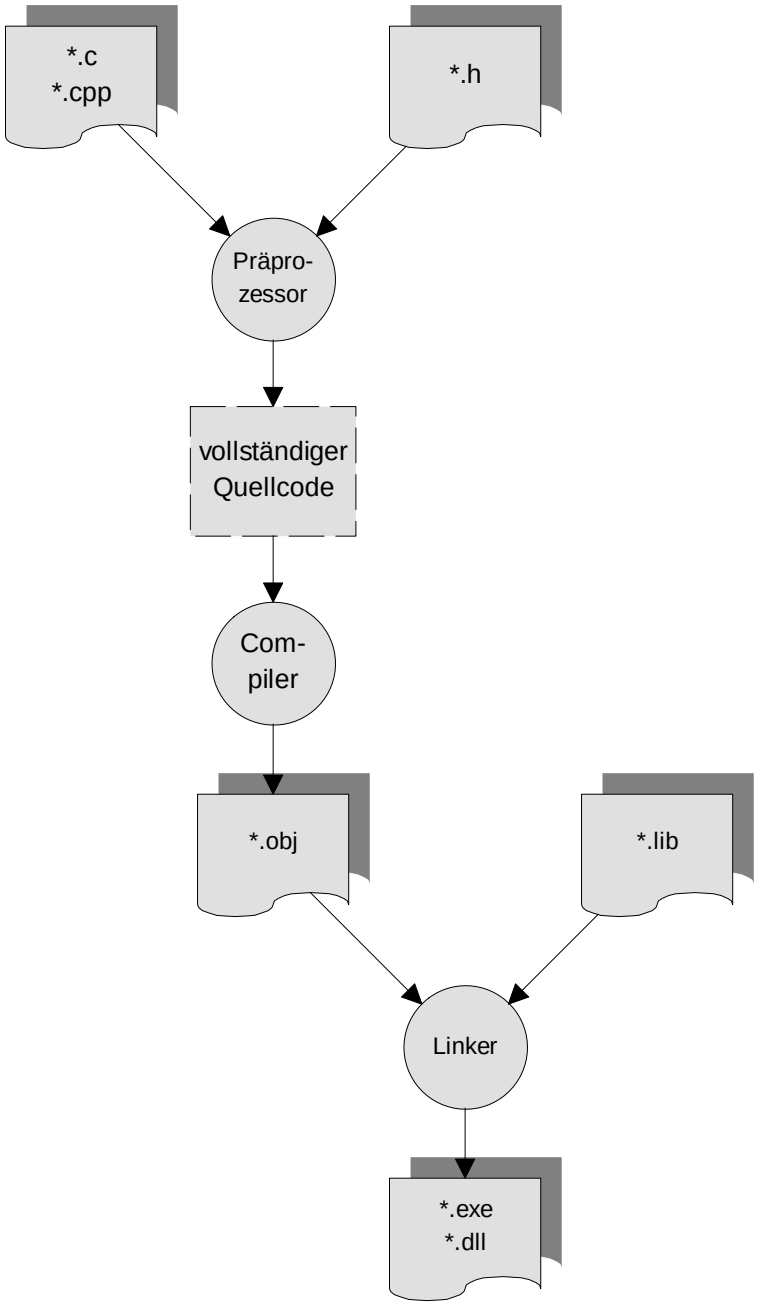


Abbildung 2-3 - Entwicklungszyklus

3. DATEN

Alle veränderlichen Werte, Texte und Unterprogramme, die in einem Programm Verwendung finden sollen, müssen zu Beginn des Programms mit dem Rechner vereinbart werden. Während der Datenmodellierung notiert der Entwickler alle benötigten Daten und ihre Bedeutung. Anschließend werden an alle Daten (die sogenannten Variablen) eindeutige und möglichst passende, sprechende Namen vergeben.

3.1 NAMEN UND NAMENSKONVENTIONEN IN C UND C++

Es gilt heute als absolute Notwendigkeit, die Variablen dergestalt zu benennen, dass bereits aus dem Namen der gespeicherte Inhalt ersichtlich wird (Tabelle 3-1).

Variablenbenennung	
Gute Benennung	Schlechte Benennung
Lohnsteuersumme	LStSm
Verkaufspreis	VkP
Anteil	a

Tabelle 3-1: Variablenbenennung

Ein gültiger Name darf – nach der Namenskonvention von C/C++ – nur aus Buchstaben, Ziffern und dem Unterstrich bestehen. Das Leerzeichen (Space bzw. Blank) und Sonderzeichen, wie z.B. der Bindestrich oder deutsche Umlaute, dürfen nicht verwendet werden. Zudem muss ein Name mit einem Buchstaben beginnen.

Prinzipiell darf ein Name zwar mit einem Unterstrich beginnen, allerdings gilt dies in den Kreisen von C/C++ Anwendungsentwicklern als unfein. Hintergrund ist, dass der Compiler selbst bei den einzelnen Übersetzungsdurchgängen den Namen von Daten und Funktionen einen oder mehrere Unterstriche voranstellt. Die Anzahl der Unterstriche gibt zugleich Hinweise auf die Art des Symbols (Variable, Funktion etc.). Beginnt man einen Namen mit einem Unterstrich, so werden die vom Compiler erzeugten Informationsdateien (z.B. Map-Dateien) wesentlich schlechter lesbar.

Die erlaubte Länge eines Namens ist abhängig vom verwendeten Compiler. Schon die ersten Compiler konnten Namen verarbeiten, die ca. 30 Zeichen lang waren, hatten jedoch die Einschränkung, dass nur die ersten 6 Zeichen signifikant waren.

Die signifikante Länge gibt die Anzahl der Zeichen an, die zur Unterscheidung der Variablen herangezogen werden. D.h. zwei Variablennamen, die sich erst an der siebten Stelle unterschieden, wurden vom Compiler als identisch angesehen.

Heute kann man von einer signifikanten Länge von mindestens 20 Zeichen ausgehen. Der ANSI-Standard schreibt sogar mindestens 31 signifikante Stellen vor, nicht jedoch dass Groß- und Kleinschreibung unterschieden werden muss (obwohl nahezu alle Compiler dies tun). Es empfiehlt sich daher, um sicher zu gehen, die signifikante Länge im Handbuch des Compilers nachzuschlagen.

Eine weitere Einschränkung bei der Vergabe von Namen besteht darin, dass die folgenden, für die Befehle des Kernel reservierten Namen (Schlüsselworte) nicht verwendet werden dürfen (Tabelle 3-2 und Tabelle 3-3).

Reservierte Worte in C und C++				
auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			

Tabelle 3-2: Reservierte Worte in C und C++

Zusätzlich reservierte Worte in C++				
catch	class	delete	friend	inline
new	operator	overload	private	Protected
public	template	this	throw	try
virtual				

Tabelle 3-3: Zusätzlich reservierte Worte in C++

Namen von Funktionen aus den Bibliotheken können verwendet werden – natürlich nur, solange die zugehörige Bibliothek nicht im Programm verwendet wird. Abhängig vom Compiler können eventuell noch einige (herstellerspezifische) Namen hinzukommen. Alle anderen Namen sind im Prinzip frei verwendbar. Da jedoch sehr viele Befehle und einige spezielle Variablen in den Bibliotheken liegen und erst bei Bedarf eingebunden werden, ist zudem darauf zu achten, dass auch hier möglichst keine Überschneidungen auftreten.

Zwar betrifft die Einschränkung nur aufgeführten Schreibweisen (Kleinschreibung), aber auch hier gilt: die Verwendung von gleichen Namen in einem Programm, die sich nur in der Schreibweise voneinander unterscheiden gilt generell als „unschicklich“.

Fast alle C/C++ Compiler unterscheiden zwischen Groß- und Kleinschreibung, zwei Variablen mit den Namen *Zaehler* und *zaehler* sind daher für den Compiler verschieden. Achten Sie daher konsequent auf die Schreibweise Ihrer Variablen oder entwickeln sie einen eigenen

Präfix, den Sie allen Funktionen voranstellen (Daten sind meist unkritisch und lokal änderbar).

Ein anderer Weg eigene Namen abzugrenzen ist die Verwendung eines Typpräfix, wie z.B. in der sogenannten „ungarischen Notation“. Die ungarische Notation ist nicht genormt, so dass es viele verschiedene, auf der Notation beruhende Schemata zur Benennung gibt. Ein Beispiel eines solchen Schemas ist in Tabelle 3-4 aufgeführt.

Präfix	Bedeutung	C/C++ Typen
a	Array, Feld	*datentyp* []
b	Boolean, logischer Wert	int, bool
by	Ganzzahliger 8-Bit Wert (Byte, vorzeichenlos)	unsigned char
c	Zeichen (Buchstabe, Ziffer etc.)	char
cx, cy	Koordinaten (Längen, Wert)	short, int
dw	Ganzzahliger 32-Bit Wert (Double Word, vorzeichenlos)	unsigned long
f	Fließkommazahl	float, double
fn	Funktion, Unterprogramm	*function*()
i	Ganzzahliger Wert (Integer)	int
l	Ganzzahliger Wert (Long Integer)	long
n	Aufzählbar numerischer Wert	short, int, long
r	Zusammengesetzter Datentyp (record, struct)	struct
sz, s	Zeichenkette (Zero Terminated String)	char []
tag, type	Typdeklaration für zusammengesetzte Datentypen	struct
w	Ganzzahliger 16-Bit Wert (Word, vorzeichenlos)	unsigned int
x, y	Koordinaten	short, int

Tabelle 3-4: Beispiel für ungarische Notation

Benutzen sie bei eigenen Funktionsnamen einen eindeutigen Präfix nach der ungarischen Notation (z.B. fn für Funktion: fnFunkionsname) oder einen Firmenpräfix (pm für Paul Müller: pmFunktionsname). Die Wahrscheinlichkeit, dass es dann noch zu Überschneidungen bei der Benennung von Funktionen kommt ist äußerst gering. Sie erhöhen dadurch außerdem die Lesbarkeit Ihrer Programme, da Sie jederzeit erkennen können, welche Variablen oder Unterprogramme aus Ihrer Feder stammen.

ACHTUNG: Microsoft Windows® verwendet eine ähnliche Konvention für Windows® Konstanten. Diese bestehen immer aus einer Folge von Großbuchstaben und einem Unterstrich (WM_ / CBN_ etc.) verwenden Sie daher Kleinbuchstaben.

Weitere Beispiele für fehlerhafte und korrekte Vergabe von Namen bei der Deklaration (Variablenvereinbarung) sind Tabelle 3-5 zu entnehmen.

3.2. DATENTYPEN

Variablen können von unterschiedlichen Typen sein, so gibt es Texte, Zahlen, logische Werte und andere. Da ein Rechner nur Sequenzen bestimmter Länge von Nullen und Einsen speichern kann, muss ihm mitgeteilt werden, wie er diese Nullen und Einsen zu interpretieren hat. Dies erreicht man durch Angabe des Variablentyps, der sich bei der Datenmodellierung zumeist automatisch ergibt. Die Verwendung von Datentypen hilft zudem Fehler zu vermeiden, da bestimmte Manipulationen nur mit den zugehörigen Datentypen erlaubt sind, so hat es z.B. im allgemeinen keinen Sinn zwei Vornamen miteinander zu multiplizieren. Grundsätzlich kennt C/C++ eine ganze Reihe von einfachen und komplexen Datentypen (siehe Tabelle 3-6 und Tabelle 3-7).

Deklarationsbeispiele	
Variablenname	Kommentar
Beispielvariable	Korrekt
Beispiel_1	Korrekt, könnte jedoch zu einem Fehler führen, wenn die signifikante Länge nur wenige Zeichen beträgt
Test_1	Korrekt
Test__2	Korrekt
_Test3	Korrekt, gilt jedoch unter C/C++ Programmierern als „unfein“
Test-3	Fehlerhaft, Sonderzeichen sind in Variablennamen nicht erlaubt
Zähler	Fehlerhaft, Umlaute sind in Variablennamen nicht erlaubt, sie gelten als Sonderzeichen
8Tung	Fehlerhaft, das erste Zeichen muss ein Buchstabe oder Unterstrich sein
Nr.Zwei	Fehlerhaft, Sonderzeichen sind in Variablennamen nicht erlaubt
Var Vier	Fehlerhaft, Leerzeichen sind in Variablennamen nicht erlaubt
register	Fehlerhaft, da reserviertes Wort
Register	Fehlerhaft, einige Compiler akzeptieren reservierte Worte in allen Schreibweisen. Außerdem würde die Verwendung von Variablen, die ähnlich wie reservierte Worte benannt sind die Lesbarkeit und die Verständlichkeit von Programmtexten mindern, da leicht Verwechslungen auftreten können

Tabelle 3-5: Deklarationsbeispiele

Um mit dem Rechner eine Variable zu vereinbaren, muss in C/C++ zunächst der Variablentyp und dann der Variablenname angegeben werden. Abschließend muss ein Semikolon angegeben werden, um die Zeile formal (syntaktisch) abzuschließen. Um die Schreibarbeit zu vermindern, kann nach einem Variablentyp eine Liste von Variablennamen folgen, die durch Kommata voneinander getrennt sind. Dadurch wird für alle angegebenen Variablen der gleiche Typ deklariert:

```
Variablentyp Variablenname;
Variablentyp Variablenname, Variablenname, ...;
```

Die Kommata sind notwendig, um zu erkennen, ob der Programmierer u.U. einen fehlerhaften Variablennamen (einen mit Leerzeichen) angegeben hat. Der syntaktische Abschluss weist den Compiler an, die Zeichen zwischen dem letzten syntaktischen Abschluss und dem Semikolon als Anweisung zu betrachten und auf formale Korrektheit zu überprüfen. Der ausdrückliche Abschluss von Anweisungen in C/C++ durch das Semikolon ermöglicht es dem Programmierer, Anweisungen zu schreiben, die sich über mehrere Zeilen erstrecken können.

Einfache Datentypen	
einfacher Typ	auch bezeichnet als
Zahl ohne Nachkommastellen	Integer, ganze Zahl
Zahl mit Nachkommastellen	Real, Float, gebrochene Zahl
Buchstaben	Char, Character, Zeichen
Logische Werte	Boolean, Wahrheitswert

Tabelle 3-6: Einfache Variablentypen

Komplexe Datentypen	
komplexer Typ	auch bezeichnet als
Texte	String, Zeichenkette
Felder	Array, Vektor, Liste
Zusammengesetzte Daten	Record, Struct, Struktur, Datensatz
Variante, zusammengesetzte Daten	Variantes Record, Union
Verkettete Listen	Pointer, Zeiger, Baum
Dateien	File
Zusätzlich in C++	
Klassen	Class, Datentyp, Typ, Objekte

Tabelle 3-7: Komplexe Variablentypen

3.3. EINFACHE DATENTYPEN

Alle diese Datentypen (siehe Tabelle 3-8 und Tabelle 3-9) haben ihre eigenen Beschränkungen, sowie Vor- und Nachteile, die im Folgenden vorgestellt werden sollen. Die komplexen Datentypen werden dabei zunächst ausgeklammert und später eingeführt, da diese sich immer aus einfachen Datentypen zusammensetzen.

Fließkomma Datentypen (Float)						
Typ	Bytes	Bits	Komma	Vor-zeichen	Werte-bereich	Präzision
float	4	32	Ja	Ja	$\pm 10^{\pm 38}$	11-12 Stellen
double	8	64	Ja	Ja	$\pm 10^{\pm 308}$	15-16 Stellen
long double	10	80	Ja	Ja	$\pm 10^{\pm 4932}$	19-20 Stellen
long float	identisch mit double					

Tabelle 3-8: Fließkomma-Datentypen

Wie aus den Tabellen ersichtlich, sind einige Deklarationen miteinander identisch. So muss z.B. der Typ *signed* nicht angegeben werden, da es sich um die Voreinstellung des Compilers handelt. Findet der Compiler den Typ *unsigned* nicht vor dem Variablentyp, so geht er automatisch von einer Variablen mit Vorzeichen aus. Ebenso verhält es sich mit dem Typ *int*, es braucht nach der Deklarationen einer Variablen als Typ *short* oder Typ *long* nicht angegeben werden, da es sich um die Voreinstellung handelt.

Integer Datentypen					
Typ	Bytes	Bits	Negativ	Minimalwert	Maximalwert
int	2/4	16/32	Ja	- 32768 - 2147483648	+ 32767 + 2147483647
unsigned int	2/4	16/32	Nein	0 0	+ 65535 + 4294967295
signed int	Identisch mit int				
short	2	16	Ja	- 32768	+ 32767
unsigned short	2	16	Nein	0	+ 65535
signed short	Identisch mit short				
short int	Identisch mit short				
unsigned short int	Identisch mit unsigned short				
signed short int	Identisch mit short				
long	4	32	Ja	- 2147483648	+ 2147483647
unsigned long	4	32	Nein	0	+ 4294967295
signed long	Identisch mit long				
long int	Identisch mit long				
unsigned long int	Identisch mit unsigned long				
signed long int	Identisch mit long				
char	1	8	Ja	- 128	+ 127
unsigned char	1	8	Nein	0	+ 255

Tabelle 3-9: Integer-Datentypen

Achtung: Wenn in einem Programm Fließkommazahlen verwendet werden sollen, so ist zu Programmbeginn die Headerdatei FLOAT.H anzugeben, die benötigt wird, um die Fließkommavarianten der Ein- und Ausgabefunktionen einzubinden.

Besonderheiten sind in C/C++ beim Typ *char* zu beachten, da dieser identisch ist mit dem Typ *signed short*. Im Prinzip ist es daher sogar möglich Buchstaben zu multiplizieren oder durcheinander zu dividieren (einmal abgesehen von der Frage, ob dies sinnvoll ist). Die meisten Compiler lassen das Verwenden von *char*-Variablen in Berechnungen einfach zu, ohne den Programmierer zu warnen.

Eine weitere Besonderheit ist, dass es in C/C++ (anders als in Java, Pascal oder Modula) keinen eigenen Typ zur Speicherung von Wahrheitswerten (z.B. logische Konstanten, Boolsche Algebra) gibt. Stattdessen kann ein beliebiger Integertyp verwendet werden. Der Wert Null (*NULL*) wird als „logisch falsch“ (*FALSE*) interpretiert, der Wert Eins als „logisch wahr“ (*TRUE*).

Unter Berücksichtigung der bisher dargestellten Regeln, sieht eine Variablendeklaration in C/C++ wie folgt aus:

```
#include <float.h>

int      iBeispiel, iErste_Zahl;
double   fFliesskommazahl;
char     cBuchstabe;

void main (void)
{
    /* Kommentar */
}
```

Nach der Deklaration enthalten die meisten neugeschaffenen Variablen noch keine Werte, oder besser gesagt keine definierten Werte. Globale Variablen werden durch die Variableninitialisierung mit Nullen oder leeren Zeichenketten vorbelegt, lokale Variablen werden nicht automatisch initialisiert. Denn die zufällig im Speicher befindlichen Nullen und Einsen werden als Wert der Variablen interpretiert. Beginnt man nun damit Rechenoperationen auszuführen, so wäre ein vernünftiges Ergebnis bloßer Zufall. In anderen Programmiersprachen umgeht man dieses Problem, indem man zunächst eine Reihe von Programmanweisungen erstellt, um den Variablen Startwerte zuzuweisen. Diese Vorgehensweise ist umständlich und anfällig für Fehler, da man bei nachträglichen Programmänderungen leicht die eine oder andere Variable vergisst. Daher gibt es in C/C++ die Möglichkeit Variablen schon bei der Deklaration mit einem Wert vorzubelegen. Dies erfolgt bei den einfachen Datentypen, indem man vor dem Komma oder Semikolon ein Gleichheitszeichen und einen gültigen Wert einfügt:

```
#include <math.h>

int    iErste_Zahl         = 7;
long   lZweimal_Erste_Zahl = iErste_Zahl * 2;
double fFliesskommazahl    = 17.899;
double fHalbe_Fliesskomma  = fFliesskommazahl / 2;
double fPi_Quadrat         = 3.14 * 3.14;
char   cBuchstabe          = 'A';

void main (void)
{
   // Kommentar
}
```

Wie man leicht erkennen kann, ist sogar die Verwendung von anderen Variablen und mathematischen Formeln bei der Initialisierung erlaubt. Voraussetzung ist jedoch, dass eine verwendete Variable vorher bereits deklariert worden ist und dass bei einer Berechnung das Ergebnis auch in der Variablen gespeichert werden kann. Die folgenden Beispiele zeigen Initialisierungen, bei denen dieses nicht der Fall ist:

```
// FEHLERHAFTE DEKLARATIONEN
int  iBeta     = 7;
long lDelta    = 200000;
long lAlpha    = 7;
int  iFehler_1 = iGamma;    // iGamma ist noch unbekannt
int  iFehler_2 = lDelta;    // Wert zu groß für iFehler_2
long lFehler_3 = lAlpha/2;  // Division ergibt
Fließkommawert!
int  iGamma    = 5;

void main (void)
{
}
```

3.4. VARIABLENDEKLARATION ALS STATEMENT IN C++

Im Gegensatz zu ANSI-C, wo Variablendeklarationen nur am Anfang eines Blocks erlaubt sind, kann man in C++ zu jeder Zeit neue Variablen einführen. Dadurch wird allerdings die Lesbarkeit des Programms erheblich eingeschränkt. Daher sollte man es, wie in ANSI-C üblich, dabei belassen, Variablen am Anfang des Funktionsblockes zu definieren.

4. OPERATOREN

Für die Verknüpfung von Daten stellen C und C++ eine Fülle von Operatoren zur Verfügung, die eine zum Teil sehr Hardware- bzw. assemblernahe Programmierung ermöglichen. Da, wie in den folgenden Beispielen noch zu sehen sein wird, die Ausnutzung von Seiteneffekten möglich ist, muss die Auswertungsreihenfolge der einzelnen Operatoren (Priorität) und die Auswertungsrichtung bekannt sein (wird, aus Sicht des Operators, zuerst der linke oder rechte Ausdruck ausgewertet?).

Entscheidend für die Reihenfolge der Auswertung ist die Priorität der einzelnen Operatoren.

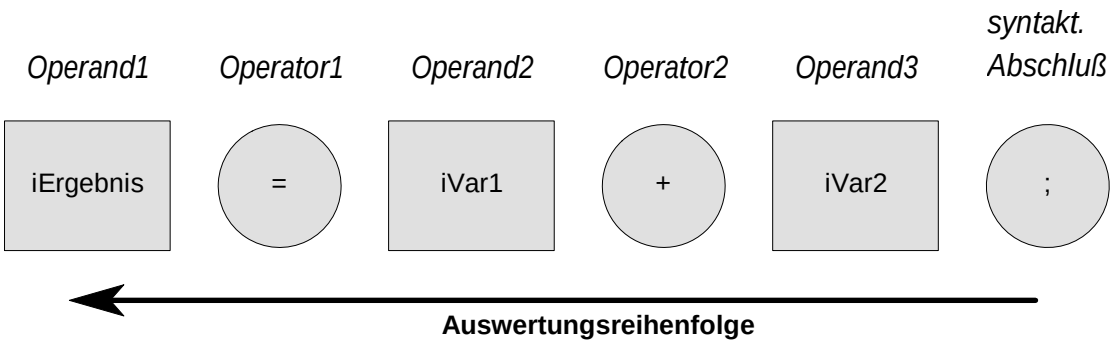


Abbildung 4-1 – Auswertungsreihenfolge

Die Operatorliste (Tabelle 4-1) zeigt alle in C/C++ verfügbaren Operatoren, ihre Priorität sowie die Auswertungsreihenfolge der mit ihnen verknüpften Argumente.

Operatoren					
Prio	Operator	Aus-wertung von	Bedeutung	Over-load	Verfüg-bar in
1	( )	links	Vorrangklammer/Parameter	Ja	C/C++
1	[ ]	links	Arrayindex	Ja	C/C++
1	.	links	Komponentenzugriff	Nein	C/C++
1	::	links	Klassendelimiter (Scope)	Nein	C++
1	->	links	Komponentenzugriff über Pointer	Ja	C/C++
2	!	rechts	logische Verneinung (NOT)	Ja	C/C++
2	~	rechts	Bitkomplement	Ja	C/C++
2	++Name	rechts	Präinkrement (vor Auswertung)	Ja	C/C++
2	Name++	rechts	Postinkrement (nach Auswertung)	Ja	C/C++
2	--Name	rechts	Prädekrement (vor Auswertung)	Ja	C/C++
2	Name--	rechts	Postdekrement (nach	Ja	C/C++

Operatoren					
Prio	Operator	Aus-wertung von	Bedeutung	Over-load	Verfüg-bar in
			Auswertung)		
2	+	rechts	unäres Plus (Vorzeichen)	Ja	C/C++
2	-	rechts	unäres Minus (Vorzeichen)	Ja	C/C++
2	(Typ)	rechts	Casting (Typwandlung)	Ja	C/C++
2	*Name	rechts	Verweisoperator	Ja	C/C++
2	&Name	rechts	Adressoperator	Ja	C/C++
2	sizeof(Name)	rechts	Größe des belegten Speicherplatzes	Ja	C/C++
2	sizeof(Typ)	rechts	Größe des belegten Speicherplatzes	Ja	C/C++
2	new Typ	rechts	dynamische Speicherreservierung	Ja	C++
2	delete Name	rechts	dynamische Speicherfreigabe	Ja	C++
2	delete [] Name	rechts	dynamische Speicherfreigabe für Arrays	Ja	C++
3	.*	links	Dereferenzierung eines Elementes einer Klasse	Nein	C++
3	->*	links	Dereferenzierung eines Elementes einer Klasse über Pointer (Zeiger)	Ja	C++
4	*	links	Multiplikation	Ja	C/C++
4	/	links	Division	Ja	C/C++
4	%	links	Modulo (Rest einer ganzzahligen Division)	Ja	C/C++
5	+	links	Addition	Ja	C/C++
5	-	links	Subtraktion	Ja	C/C++
6	<<	links	Bit-Linksverschiebung (Leftshift)	Ja	C/C++
6	>>	links	Bit-Rechtsverschiebung (Rightshift)	Ja	C/C++
7	<	links	kleiner als	Ja	C/C++
7	<=	links	kleiner als oder gleich	Ja	C/C++
7	>	links	größer als	Ja	C/C++
7	>=	links	größer als oder gleich	Ja	C/C++
8	==	links	Test auf Gleichheit (Vergleich)	Ja	C/C++
8	!=	links	Test auf Ungleichheit	Ja	C/C++
9	&	links	Und-Verknüpfung von Bits (AND)	Ja	C/C++
10	^	links	Exklusive Oder-Verknüpfung von Bits (XOR)	Ja	C/C++
11	|	links	Oder-Verknüpfung von Bits (OR)	Ja	C/C++
12	&&	links	Logisches Und (AND)	Ja	C/C++
13	||	links	Logisches Oder (OR)	Ja	C/C++
14	? :	rechts	Bedingte Bewertung	Nein	C/C++

Operatoren					
Prio	Operator	Auswertung von	Bedeutung	Overload	Verfügbar in
15	=	rechts	Wertzuweisung	Ja	C/C++
15	+=	rechts	Addition und Zuweisung	Ja	C/C++
15	-=	rechts	Subtraktion und Zuweisung	Ja	C/C++
15	*=	rechts	Multiplikation und Zuweisung	Ja	C/C++
15	/=	rechts	Division und Zuweisung	Ja	C/C++
15	%=	rechts	Ganzzahlige Division (Modulo) und Zuweisung	Ja	C/C++
15	<<=	rechts	Leftshift und Zuweisung	Ja	C/C++
15	>>=	rechts	Rightshift und Zuweisung	Ja	C/C++
15	&=	rechts	Bitverknüpfung AND und Zuweisung	Ja	C/C++
15	|=	rechts	Bitverknüpfung OR und Zuweisung	Ja	C/C++
15	^=	rechts	Bitverknüpfung XOR und Zuweisung	Ja	C/C++
16	,	links	Separator	Ja	C/C++

Tabelle 4-1: Operatorliste

Aufgrund der Vielzahl der Operatoren und der Vielfalt ihrer Einsatzgebiete können die einzelnen Operatoren nur nach und nach eingeführt und erklärt werden. Nachstehend erfolgt eine kurze Einführung in die einfachen, mathematischen Verknüpfungen.

4.1. DIE MATHEMATISCHEN OPERATOREN

C/C++ behandelt eine Vielzahl von Sonderzeichen als Operatoren, darunter fallen einige Zeichen, wie z.B. die Vorrangklammern, die vielfach nicht als Operatoren angesehen werden. Die runden Klammern ( ) dienen zur Festlegung der Berechnungsreihenfolge, wenn diese nicht dem Standardschema Multiplikation/Division vor Addition/Subtraktion (Punktrechnung von Strichrechnung) folgt. So ist z.B. das Ergebnis der beiden folgenden Zeilen unterschiedlich:

```
#include <math.h>

int    iA = 1;
int    iB = 2;
int    iC = 3;
double fE = 0.0;

void main (void)
{
   fE =  iA + iB  * iC;   // Ergibt 7
   fE = (iA + iB) * iC;   // Ergibt 9
}
```

Wie im nächsten Beispiel ersichtlich, schließen die runden Klammern zudem die Parameter für Unterprogramme ein. Damit das Ergebnis eines Unterprogrammes in einer Berechnung verwendet werden kann, muss es mit höherer Priorität ausgeführt werden, als die Berechnung selbst.

```
#include <math.h>

int    iA = 1;
int    iB = 2;
int    iC = 3;
int    iD = 9;
double fE = 0.0;

void main (void)
{
   fE = (iA + iB) * iC / sqrt (iD);   // Ergibt 3
}
```

Etwas geringere Priorität als die Vorrangklammern, aber noch höhere als die Multiplikation oder Division, genießen die Inkrement und Dekrement Verknüpfungen. Eine sehr häufige Anweisung in jeder Programmiersprache ist die Erhöhung oder Verminderung eines Variableninhaltes um den Wert Eins. Die typische Anweisung in den gängigen Sprachen, wie z.B. PASCAL, lautet in etwa wie folgt:

```
#include <math.h>

int iZahl = 1;

void main (void)
{
   iZahl = iZahl + 1;    // iZahl erhält den Wert 2
   iZahl = iZahl - 1;    // iZahl erhält wieder Wert 1
}
```

Die Übersetzung durch den Compiler ergibt zumeist eine Additionsanweisung im Assemblercode (Objectcode), obwohl jeder Prozessor für die obigen Beispiele gesonderte, wesentlich schnellere und optimierte Befehle besitzt, den Befehl für Inkrement bzw. Dekrement. Dieser kann in C/C++ gezielt angesprochen werden:

```
#include <math.h>

int iZahl = 1;

void main (void)
{
   iZahl++;    // iZahl erhält den Wert 2
   iZahl--;    // iZahl erhält wieder den Wert 1
   ++iZahl;    // iZahl erhält den Wert 2
   --iZahl;    // iZahl erhält wieder den Wert 1
}
```

Beide Anweisungen gibt es in einer Prä- und Postversion. Beim Präinkrement oder Prädekrement wird zunächst der Wert im Speicher um den Wert Eins verändert und anschließend die betreffende Variable für eine Berechnung aus dem Speicher geholt. Bei den Post-Varianten ist es genau umgekehrt, zunächst wird der Wert, z.B. für eine Weiterverwendung in einer Berechnung, aus dem Speicher geholt und erst dann der Wert im Speicher (nicht in der Berechnung) um den Wert Eins geändert.

Der Unterschied zwischen beiden Varianten wird erst deutlich, wenn das Inkrement- oder Dekrement innerhalb einer Berechnung oder einer anderen Anweisung verwendet wird:

```
#include <math.h>

int    iZahl1    = 3;
int    iZahl2    = 2;
double fErgebnis = 0.0;

void main (void)
{
   fErgebnis = iZahl1 * iZahl2++; // Postinkrement
                                  // Ergebnis erhält Wert 6
                                  // iZahl2 ist danach 3
   iZahl2    = 2;                // iZahl2 wieder 2 setzen
   fErgebnis = iZahl1 * ++iZahl2; // Präinkrement
                                  // Ergebnis erhält Wert 9
                                  // iZahl2 ist danach 3
}
```

Das Ergebnis der ersten Berechnung ist der Wert Sechs, da beide Werte (bevor der Inkrementbefehl greift) aus dem Speicher in die Berechnung eingefügt werden. Die Tatsache, dass der Multiplikationsoperator eine geringere Priorität besitzt als der Inkrementoperator ist hier ohne Bedeutung, denn Inkrement und Dekrement arbeiten unär (wie ein Vorzeichenminus). Im zweiten Beispiel ist das Ergebnis Neun, da zuerst Zahl2 von Zwei auf Drei erhöht wird und erst dann die Werte von Zahl1 und Zahl2 zwecks Multiplikation aus dem Speicher geholt werden.

Die Division und Multiplikation bedürfen sicherlich, wie auch die Addition und Subtraktion, keiner genaueren Beschreibung, anders hingegen der Operator „%“ (Modulo). Der Modulo-Operator steht für die ganzzahlige Division mit Rest, wobei dieser Rest das Ergebnis der Operation ist:

```
#include <math.h>

double fErgebnis = 0.0;

void main (void)
{
   fErgebnis = 17 % 4;     // Modulo: 17 durch 4 ist 4 Rest 1
                           // Ergebnis ist 1
}
```

Das Ergebnis der obigen Beispielrechnung ist der Wert Eins, denn Siebzehn dividiert durch Vier ergibt Vier mit dem Rest Eins.

In allen bisher gezeigten Beispielen findet sich der Zuweisungsoperator „=“, der einer Variablen einen Wert, hier das Ergebnis der Berechnung, zuweist. Eine Zuweisung kann nur an eine Variable erfolgen, niemals an eine Konstante oder eine Funktion – daher kann auf der linken Seite des Gleichheitszeichens immer nur ein Variablenname stehen, während rechts eine beliebige Kombination aus Variablen, Konstanten und Funktionen stehen darf, die miteinander verknüpft sind. Man beachte bitte dass es sich hier um Zuweisungen handelt und nicht, wie man auf den ersten Blick vermuten könnte um Gleichungen.

Eine weitere Besonderheit von C/C++ sind die kombinierten Zuweisungsoperatoren mit niedriger Prioritätsstufe, die zunächst ziemlich sinnlos erscheinen (+=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=). Wie schon beim Inkrement und Dekrement ist der Hauptgedanke auch hier das Vermeiden unnötiger Speicherzugriffe. Normalerweise wird jedesmal, wenn ein Variablenname erscheint dessen Wert oder Adresse im Speicher ermittelt. Der Vorgang der Ermittlung ist, aus Sicht des Prozessors, verhältnismäßig langsam. Wenn sich also Taktzyklen einsparen lassen, indem man weniger häufig auf den Speicher zugreift, so ist dies ein Geschwindigkeitsgewinn im Programmablauf. Eine typische Programmzeile, wie man sie in fast allen Programmiersprachen sehr häufig antrifft ist etwa die folgende:

```
#include <math.h>

int iZahl1 = 3;
int iZahl2 = 2;

void main (void)
{
   iZahl1 = iZahl1 + iZahl2;    // typische Zuweisung
}
```

Hier wird der Inhalt von Zahl1 mit dem Inhalt von Zahl2 addiert und das Ergebnis nach Zahl1 zurückgeschrieben. Zur Verarbeitung der Anweisung ermittelt der Rechner die Adresse von Zahl2 und holt sich dann an dieser Adresse den Wert von Zahl2 ab. Anschließend ermittelt der Rechner die Adresse von Zahl1 und holt sich dann analog zum vorherigen Vorgehen an dieser Adresse den Wert von Zahl1 ab. Wenn beide Werte bekannt sind, werden die Zahlen addiert. Um das Ergebnis zu speichern wird nun erneut die Adresse von Zahl1 ermittelt und das Rechenergebnis dorthin gespeichert. Es ist leicht zu erkennen, dass die Adresse von Zahl1 zweimal ermittelt wird, was eigentlich unnötig ist, denn diese ist ja bereits vom vorherigen Schritt bekannt. Dieser Tatsache kann man in C/C++ Rechnung tragen und die obige Anweisung auch wie folgt geben:

```
#include <math.h>

int iZahl1 = 3;
int iZahl2 = 2;

void main (void)
{
   iZahl1 += iZahl2;   // optimierter Zugriff
}
```

Das Ergebnis der Anweisung „+=„ ist identisch mit dem vorangehenden Beispiel, bis auf die Tatsache, dass die Zugriffe optimiert erfolgen, d.h. die Adresse von Zahl1 nur einmal ermittelt wird. Diese Form der Optimierung ist vor allem in Schleifen interessant, wenn hunderte oder Tausende von Operationen so optimiert zugreifen. Anhand der Liste der Operatoren (Tabelle 4-1) ist zu erkennen, dass diese Art der Optimierung mit einer ganzen Reihe von Verknüpfungen möglich ist.

4.2. DIE SEITENEFFEKTE

Der Erfolg von C/C++ wäre undenkbar gewesen, wenn die Verwendung von Seiteneffekten nicht eine der elegante (leider aber auch gegenüber Fehlern sehr anfällige) Methode der Optimierung wäre. In C/C++ hat jeder Ausdruck ein Ergebnis. Als einfachstes und verständlichstes Beispiel sei hier die Zuweisung angeführt.

Wie eine Addition hat auch die Zuweisung selbst ein Ergebnis – den zugewiesenen Wert (im Beispiel die Drei). Diesen Wert kann man, wenn man möchte, wiederum in einem Ausdruck (z.B. einer weiteren Zuweisung) verwenden:

```
#include <math.h>

int iZahl1 = 0;
int iZahl2 = 0;
int iZahl3 = 0;

void main (void)
{
   iZahl1  = 3;
   iZahl3  = iZahl1 = 3;
   iZahl2 += iZahl1 = 3 * iZahl3;
}
```

Zum Beispiel kann man eine solche doppelte Zuweisung durchführen, oder etwas kompliziertere Ausdrücke, wie die zweite Anweisung formulieren. Empfehlenswert ist ein solches Vorgehen nicht unbedingt, da hierbei die Ergebnisse stark von der Auswertungsreihenfolge und der Priorität abhängen können. Es ist daher besser auf die Verwendung von Seiteneffekten zu verzichten.

4.3. MATHEMATIK UND ERGEBNISTYPEN

Durch die Größen- und Genauigkeitsbeschränkungen der meisten Datentypen ist darauf zu achten, dass die Verknüpfung zweite Operanden dazu führen kann, dass die angegebene Ergebnisvariable zu klein ist:

```
#include <math.h>

short nZ1 = 0;
short nZ2 = 1000;
short nZ3 = 1000;

void main (void)
{
   nZ1 = nZ2 * nZ3;   // Überlauf !!!
}
```

Diese Anweisung versucht einen Wert von einer Million auf einer Variablen vom Typ *short* abzuspeichern, die maximal den Wert 32767 annehmen kann. Das Resultat ist ein Überlauf und der in *nZ1* gespeicherte Wert ist falsch, da er irgendwo im Bereich zwischen 32767 und -32768 liegt.

Auch die folgende Anweisung erzeugt einen Überlauf:

```
#include <math.h>

short nZ1 = 32767;

void main (void)
{
   nZ1 += 1;   // Überlauf !!!
}
```

Das Ergebnis dieser Zuweisung ist der Wert -32768. Aufgrund der internen Organisation eines Rechners ist das höchste Bit immer das Vorzeichenbit (sofern die Zahl mit einem Vorzeichen behaftet ist) und negative Werte sind das binäre Komplement zu ihren positiven Werten abzüglich Eins:

```
  0111 1111 1111 1111    Wert  32767
+ 0000 0000 0000 0001    Wert      1
= 1000 0000 0000 0000    Wert -32768
```

Addiert man nun die 32767 und die Eins, so erhält man durch Überträge im dualen Zahlensystem genau den Wert -32768. Um solche Fehler zu vermeiden, muss man abschätzen, wie groß ein berechneter Wert werden kann, bzw. gleich einen größeren Zahlentyp vorsehen. Folgende Regeln sollten bei der Verknüpfung beachtet werden - verwenden Sie den kleineren Typ nur, wenn Sie absolut sicher sind, dass der Wertebereich und die notwendige Präzision ausreichen (Tabelle 4-2 und Tabelle 4-3).

Ergebnistypen bei Integer-Operationen		
Typ Operand 1	Typ Operand 2	Ergebnistyp
short	short	short oder int
short	int	int oder long
int	int	int oder long
int	long	long
long	long	long

Tabelle 4-2: Integer Ergebnistypen

Ergebnistypen bei Fließkomma-Operationen		
Typ Operand 1	Typ Operand 2	Ergebnistyp
short	float	float
int	float	float
long	float	float
float	float	float oder double
float	double	double
double	double	double oder long double
double	long double	long double
long double	long double	long double

Tabelle 4-3: Fließkomma Ergebnistypen

4.4. BITMANIPULATION

Für die binäre Manipulation von Daten stellt C/C++ eine ganze Reihe von Operatoren (Tabelle 4-4) zu Verfügung. Häufig werden diese Operatoren für die Bearbeitung von Zustandsflags oder in der Bildbearbeitung (Bitplanes) eingesetzt.

Operatoren der Bitmanipulation		
Deutsche Begrifflichkeit	Englischer Begriff	Operator
„Und“-Verknüpfung von Bits	AND	&
„Oder“-Verknüpfung von Bits	OR	|
„Entweder-oder“-Verknüpfung von Bits	XOR	^
Bitverschiebung nach links	LEFT SHIFT	< <
Bitverschiebung nach rechts	RIGHT SHIFT	> >
Komplementbildung / Umkehrung	Complement	~

Tabelle 4-4: Bit-Operatoren

Die binären Verknüpfungen „AND“, „OR“ , „XOR“ , „LEFT SHIFT“ sowie „RIGHT SHIFT“ benötigen jeweils zwei Operanden, die Bildung des Komplements hingegen ist, wie auch das Vorzeichenminus, unär. Bei den „SHIFT“-Befehlen gibt der zweite Operand jeweils an, um wie viele Stellen die Bits im ersten Operanden verschoben werden soll. Aufgrund der binären Stellenwertigkeit entspricht eine Verschiebung nach links („LEFT SHIFT“) einer Multiplikation mit Zwei. Eine Verschiebung nach rechts („RIGHT SHIFT“) ist dementsprechend mit einer ganzzahligen Division durch Zwei (mit Abrundung) gleichzusetzen. Normalerweise werden bei der Verschiebung Nullen an den Rändern nachgezogen, es ist jedoch zu beachten, dass bei einigen Systemen der Wert des Carry-Flag (Überlaufbit) zur Auffüllung verwendet wird. Ist dieses der Fall, so entspricht dies einer wiederholten Verschiebung einer Bit-Rotation (bitte beachten Sie hier das C/C++ Handbuch Ihres Compilers).

Die Komplementbildung (alle Bits der Variablen werden gekippt und somit in ihr Gegenteil umgewandelt) entspricht - aufgrund der internen Zahlendarstellung - einem Vorzeichenwechsel und einer anschließenden Subtraktion von Eins. Für eine Variable *nX* bedeutet dies:

```
#include <math.h>

short nX = 100;

void main (void)
{
   ~nX;   // ist gleichzusetzen mit (-1 * nX) -1
}
```

In den folgenden Beispielen, ist jeweils auf der rechten Seite der Tabelle die Binärdarstellung des Bytes zu sehen:

Binärdarstellung von Zahlen	
Variablendefinition	Binärdarstellung
Char A = 1	0000 0001
Char B = 3	0000 0011
Char C = 8	0000 1000
Char D = 15	0000 1111
Char E = 85	0101 0101

Tabelle 4-5: Beispiele für die Binärdarstellung von Zahlen

Binäroperation: Bitverschiebung nach rechts				
	Wert vor der Operation		Wert nach der Operation	
Ausdruck	Binär	Dezimal	Binär	Dezimal
A = A >> 1	0000 1000	8	0000 0100	4
B = B >> 2	0000 1000	8	0000 0010	2
C = C >> 3	0000 1000	8	0000 0001	1
D = D >> 4	0000 1000	8	0000 0000	0

Tabelle 4-6: Bitverschiebung rechts

Binäroperation: Bitverschiebung nach links				
	Wert vor der Operation		Wert nach der Operation	
Ausdruck	Binär	Dezimal	Binär	Dezimal
E = E << 1	0000 0011	3	0000 0110	6
F = F << 2	0000 0011	3	0000 1100	12
G = G << 3	0000 0011	3	0001 1000	24
H = H << 4	0000 0011	3	0011 0000	48
I = I << 5	0000 0011	3	0110 0000	96
J = J << 6	0000 0011	3	1100 0000	-64

Tabelle 4-7: Bitverschiebung links

Binäroperation: Bitkomplement				
	Wert vor der Operation		Wert nach der Operation	
Ausdruck	Binär	Dezimal	Binär	Dezimal
K = ~K	0001 1000	12	1110 0111	-13
L = ~L	1111 1101	-3	0000 0010	2
M = ~M	0000 0000	0	1111 1111	-1
N = ~N	0111 1111	127	1000 0000	-128

Tabelle 4-8: Bit-Komplement

Binäroperation: AND			
Ausdruck	Binär		Dezimal
O = O & P		0000 1111	15
	&	0101 0101	85
	=	0000 0101	5

Tabelle 4-9: Bit-UND-Verknüpfung

Binäroperation: OR			
Ausdruck	Binär		Dezimal
Q = Q | R		0000 1111	15
	|	0101 0101	85
	=	0101 1111	95

Tabelle 4-10: Bit-ODER-Verknüpfung

Binäroperation: XOR			
Ausdruck	Binär		Dezimal
Q = Q ^ R		0000 1111	15
	^	0101 0101	85
	=	0101 1010	90

Tabelle 4-11: Bit-EXKLUSIV-ODER-Verknüpfung

Man beachte den Unterschied zwischen dem logischen „Und“ (*&&*) und der Bitmanipulation „Und“ (*&*). Ein Vergleich der oben aufgeführten Variablen zeigt, dass bei einer Verwechslung der Operatoren sehr leicht schwerwiegenden Fehler entstehen können:

```
A && C --> 1   (TRUE)   // entspricht 1 && 8
A &  C --> 0   (FALSE)  // entspricht 1 &  8
```

Wie leicht zu erkennen ist, können die beiden Operatoren nicht gleichbedeutend verwendet werden - auch wenn aufgrund der ähnlichen Schreibweise diese Vermutung sehr naheliegend ist.

4.5. REGELN ZUR KLAMMERUNG VON AUSDRÜCKEN

Die letztlich sehr unübersichtlichen Vorrangregeln für Operatoren, die man schwerlich komplett im Gedächtnis behalten kann und die leicht zu vermeidbaren Fehlern führen können (wie im folgenden Beispiel), legen es nah, Ausdrücke auf jeden Fall zu klammern.

```
Variable1 && Variable2 == Variable3 || Variable4
```

Im ersten Ansatz legt das Beispiel nahe, dass zunächst Variable1 mit Variable2 verknüpft wird (AND), anschließend Variable3 mit Variable4

(OR) und die Ergebnisse der beiden Operationen miteinander verglichen werden. - Mitnichten, denn gemäß der C/C++ Operatorpräzedenzen (= Prioritäten) wird zuerst der Vergleichsoperator ausgeführt, wobei die Inhalte von Variable2 und Variable3 miteinander verglichen werden. Anschließend wird die „AND“-Verknüpfung mit dem Ergebnis der Vergleichs durchgeführt und zuletzt der „OR“-Operator.

In C/C++ gibt es nur zwei Regeln, die man bei logischen Ausdrücken und komplexen Berechnungen beachten muss:

*Regel 1: Bei einfachen Rechnungen (Ausdrücke, die nur mit den mathematischen Operatoren +, -, * und / gebildet werden) gilt Punktrechnung vor Strichrechnung (wie man es aus der Schule kennt).*

Regel 2: Tauchen außer den in Regel 1 genannten Operatoren noch andere auf, dann sollte man eindeutig klammern.

Probleme mit der Anzahl der Klammerungsebenen sind kaum zu befürchten, die ANSI-Kommission hat das Minimum an zu unterstützenden Ebenen auf 32 festgelegt.

4.6. FLUSSDIAGRAMMSYMBOL FÜR ANWEISUNGEN

Zur Vorbereitung einer Programmimplementation werden meist graphische Mittel verwendet. Zu den gebräuchlichsten Verfahren zur optischen Darstellung von Abläufen gehören z.B. Flussdiagramme und Nassi-Shneiderman-Diagramme (Struktogramme). Beide Verfahren haben ihre Vor- und Nachteile. Der Vorteil von Flussdiagrammen ist ihre bessere Übersichtlichkeit, wobei sich Designfehler aber leider nicht ausschließen lassen.

Nassi-Shneiderman-Diagramme lassen Designfehler, wie z.B. vergessene Verbindungslinien (ein typischer Fehler in großen Flussdiagrammen), nicht zu.

Andererseits werden die Struktogramme sehr schnell zu groß für eine übersichtliche Darstellung und eine Möglichkeit mehrere Diagrammteile durch aneinanderlegen zu verbinden ist kaum machbar.

Im Zuge dieses Kurses werden an entsprechender Stelle die zugehörigen Flussdiagrammsymbole eingeführt. Die in den Übungsteilen zu schreibenden Programme sind jedoch sehr klein, so dass eine Arbeitsvorbereitung mit Flussdiagrammen nicht zwingend notwendig ist.

Das Start- und Endsymbol eines Programms ist jeweils ein abgerundetes Rechteck (s.u.), in welches der Programm- oder Unterprogrammname eingetragen wird. Das häufigste und allgemeinste Symbol ist das Anweisungsrechteck (s.u.). Es steht für Zuweisungen, Berechnungen und andere Befehle, die kein eigenes Symbol besitzen:

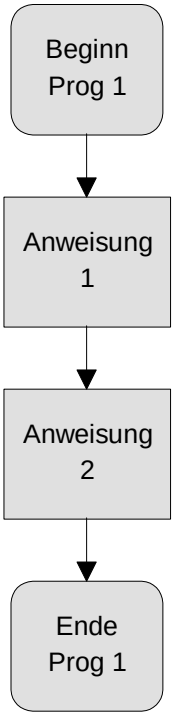


Abbildung 4-2 – Struktogramm für Anweisungen

5. STANDARD EIN- UND AUSGABE

In diesem Abschnitt wird die standardisierte Ein- und Ausgabe vorgestellt, die in C und C++ vorhanden ist. Die Standard-IO wird unter C++ vom sogenannten Stream-Konzept abgelöst, welches besser mit eigenen Klassen (Objektdefinitionen) nutzbar ist. Da aber beide Konzepte noch verwendet werden, ist es weiterhin notwendig, auch die Standard-IO programmieren zu können.

Der Kern von C/C++ selbst enthält keinerlei Ein- oder Ausgabefunktionen. Stattdessen befinden sich diese Unterprogramme in den Funktionsbibliotheken von C/C++. Der Vorteil besteht darin, dass aus den Bibliotheken nur diejenigen Teile übernommen werden, die im Programm auch verwendet werden. Daher sind Programme, die mit C/C++ geschrieben wurden im Allgemeinen sehr klein und schnell. Die Übernahme der Funktionen aus einer der Funktionsbibliotheken erfolgt erst nach dem Übersetzungsvorgang (Compilieren) beim sogenannten Binden (Linking). Um eine Funktion aus einer der Bibliotheken zu übernehmen muss sie zunächst dem Compiler bekannt gemacht werden. Dies geschieht meist durch Übernahme der Funktionsdeklaration aus einer Headerdatei (Headerfile).

Die Einbindung einer Headerdatei bedeutet nicht zwangsläufig, dass alle darin deklarierten Funktionen auch in das Programm übernommen werden. Vielmehr sind die Headerdateien nach häufig verwendeten Funktionsbereichen geordnet. So gibt es Headerdateien für die Ein- und Ausgabe, für Graphik, Textbearbeitung usw. Im Verlauf des Kurses wird sich die Nützlichkeit dieses Konzeptes zeigen, da jederzeit neue Headerdateien und neue Funktionsbibliotheken hinzugefügt werden können und so der Gesamtumfang der Funktionalität von C erhöht wird. Funktionen die in einer Headerdatei deklariert werden, aber im Programm keine Verwendung finden, werden beim Linking wieder entfernt.

Die folgenden Ein- und Ausgabefunktionen sind entweder in der Headerdatei <stdio.h> oder der Headerdatei <conio.h> deklariert. Um diese Funktionen nutzen zu können muss die Headerdatei am Anfang des Programmtextes (noch vor der Variablendeklaration und dem Programmtext) mittels einer sogenannten *include*-Direktive (oft auch *include*-Anweisung) in das Programm eingebunden werden. Eine mehrfache Einbindung von *include*-Dateien ist unkritisch, da die Headerdateien selbst dafür sorgen, dass jede von ihnen (pro Compilerlauf) nur ein einziges Mal übersetzt wird. Wie man dieses Verhalten in eigenen Headerdateien erreichen kann wird im Kapitel über den Präprozessor behandelt.

```
#include <stdio.h>
#include <conio.h>

```

```
void main (void)
{
   // Kommentar
}
```

Das „#“-Zeichen kennzeichnet die *include*-Anweisung als eine Instruktion, die vom Präprozessor auszuführen ist. Der Präprozessor bereitet den eigentlichen Übersetzungsvorgang, das Compilieren, vor. Die *include*-Anweisung besagt, dass an der entsprechenden Stelle der Inhalt der angegebenen Datei einzufügen ist. Die Headerdateien enthalten eine Reihe von C/C++ Instruktionen, die dann mit dem Programm zusammen übersetzt werden. Eine Beschreibung der Direktiven und der übrigen Anweisungen die vom Präprozessor verarbeitet werden, folgt in einem eigenen Kapitel.

5.1. FORMATIERTE AUSGABE MIT PRINTF

Eine der wichtigsten Funktionen in C ist die Funktion *printf*. Sie dient zur formatierten Ausgabe von Texten und Variablen. Besonders praktisch ist die Tatsache, dass in der *printf*-Funktion Ausgabetexte und Variablenwerte nahezu beliebig gemischt werden können.

```
Syntax:
   printf (Formatstring, Variablenliste);

Beispiel:
   printf  ("Hallo Nr. %d und %6.3f\n", Intvar,
             Floatvar);)
```

```
//=============================================================
// Programm PRINTF1.CPP
//=============================================================

#include <stdio.h>

int iZahl       = 7;
int iLaengenvar = 3;
int iErste_Zahl = 6;

void main (void)
{
   printf ("\nNoch\neine\nZeile\n");
   printf ("Resultat: Nur %d %%", iErste_Zahl);
   printf ("Man kann auch rechnen: %d %d %d\n", iZahl+4, 7+2,
           iZahl-3);
   printf ("Ausgabelänge aus Variable: Nur %*d %%",
           iLaengenvar, iErste_Zahl);
}
```

Die Beispiele zeigen neben einigen lesbaren Befehlen und den bereits bekannten Variablennamen eine Fülle von zunächst verwirrenden Sonderzeichen, die den Formatstring ziemlich sinnlos erscheinen lassen. Beim Formatstring ist zu beachten, dass er (wie alle Zeichenketten in C) in doppelte Hochkommata eingeschlossen ist (*"Textbeispiel"*). Im Gegensatz dazu stehen einzelne Zeichen (char) in einfachen Hochkommata (*'A'*).

Bei den Sonderzeichen im Formatstring handelt es sich um Platzhalter (*%*) und das Fluchtsymbol (\). Beide stehen jeweils in Verbindung mit einem oder mehreren nachfolgenden Zeichen und beiden kommt bei der Ausgabe jeweils eine besondere Aufgabe zu.

5.1.1. PLATZHALTER FÜR VARIABLENINHALTE

Das Prozentzeichen steht für eine Zahl, einen Buchstaben oder einen Text aus der Variablenliste. Dabei entspricht die Reihenfolge der Platzhalter genau der Reihenfolge der Variablen in der Liste. Im ersten Beispiel steht also das erste Prozentzeichen als Platzhalter für die Variable *Intvar* und das zweite Prozentzeichen für die Variable *Floatvar*. Damit der Rechner auch weiß, welche Art von Variable er an der Stelle einfügen soll, wird das Zeichen *„%“* um einen oder mehrere Buchstaben ergänzt, die den Typ der an dieser Stelle auszugebenden Variable angeben. Entsprechend steht also die Zeichenkombination *„%d“* für eine Variable vom Typ *int* und *„%f“* für eine Variable vom Typ *float*. Die Zahlen in der Platzhalterangabe *„%6.3f“* stellen eine optionale Formatangabe dar. Diese besagt in diesem Fall, dass die Fließkommazahl insgesamt sechsstellig auszugeben ist, wobei drei dieser Stellen auf die Nachkommastellen entfallen. Die nachstehenden Tabellen enthalten alle üblichen Platzhalter und Formatierungsschalter, die in einem C-Compiler vorkommen sollten. Es kann durchaus sein, dass einige Compiler mehr Schalter und Platzhalter anbieten, diese stellen dann jedoch eine Erweiterung des Standards dar.

Platzhalter in Formatstrings	
Symbol	Bedeutung
%d oder %i	Integerzahl einfügen
%u	Integerzahl ohne Vorzeichen einfügen (unsigned int)
%o	Integerzahl in oktaler Darstellung einfügen
%x	Integerzahl in hexadezimaler Darstellung einfügen (Hexadezimalzahlen „a-f“)
%X	Integerzahl in hexadezimaler Darstellung einfügen (Hexadezimalzahlen „A-F“)
%f	Fließkommazahl einfügen
%e	Fließkommazahl im Exponentialformat einfügen (Format [-]d.ddd e[+/-]ddd)
%E	Fließkommazahl im Exponentialformat einfügen (Format [-]d.ddd E[+/-]ddd)

Platzhalter in Formatstrings	
Symbol	Bedeutung
%g	Fließkommazahl einfügen im Format %f oder %e (jeweils die kürzere Form)
%G	Fließkommazahl einfügen im Format %f oder %E (jeweils die kürzere Form)
%c	Buchstabe aus char-Variable einfügen
%s	Zeichenkette aus String einfügen
%p	Speicheradresse einfügen
%%	Prozentzeichen ausgeben

Tabelle 5-1: Format-Platzhalter in der Standard-IO

Besondere Aufmerksamkeit in der Tabelle 5-1 verdient der „%%“-Platzhalter. Da das Prozentzeichen die Ausgabe eines Variablenwertes einleitet, muss ein besonderer Weg beschritten werden, wenn man das Prozentzeichen selbst auf dem Bildschirm ausgeben will. Um dieses zu erreichen ist das Prozentzeichen im Formatstring einfach doppelt anzugeben.

5.1.2. FORMATIERUNGSSCHALTER FÜR VARIABLENINHALTE

Die Formatierungsschalter können jeweils mit einer ganzen Reihe von Platzhaltern verwendet werden, sie geben z.B. an, ob es sich um eine Variable vom Typ *short* oder *long* handelt. Die Standardtypen (*int*, *char* und *float*) benötigen keine zusätzlichen Formatierungsschalter (Tabelle 5-2).

Die Schalter „+“ und „-“ haben eine besondere Bedeutung. Normalerweise erfolgen alle Ausgaben rechtsbündig, mit Hilfe des Schalters „-“ kann man den Rechner allerdings dazu veranlassen Zahlen und Texte linksbündig auszugeben. Der Schalter „+“ veranlasst den Rechner das Vorzeichen auch dann mit auszugeben, wenn die Zahl positiv ist. Bei negativen Werten erfolgt die Ausgabe des Vorzeichens ohnehin automatisch.

Formatierungsschalter in Formatstrings	
Schalter	Bedeutung
Zahl	Anzahl der (mindestens) auszugebenden Stellen (Integer)
Zahl.Zahl	Anzahl der (mindestens) auszugebenden Stellen (Float), Gesamtzahl der Zeichen und anteilige Nachkommastellen)
+	Ausgabe mit Vorzeichen (+ und -)
-	Linksbündige Ausgabe
0	Ausgabebreite mit führenden Nullen auffüllen
*	Lies die zu verwendende Breite aus der Variablenliste
Schalter	ergänzendes Formatkennzeichen für interne Variablenlänge
h	Wenn Typ %d, %i, %u, %o, %x, %X SHORT ist

Formatierungsschalter in Formatstrings	
Schalter	Bedeutung
l	Wenn Typ %d, %i, %u, %o, %x, %X, LONG ist oder wenn Typ %f, %e, %E, %g, %G DOUBLE ist
L	Wenn Typ %f, %e, %E, %g, %G LONG DOUBLE ist
F	Wenn Typ %p oder %s FAR-Zeiger (32 Bit) sind (dies ist kein ANSI-Schalter und ist daher nur auf IBM-PCs und kompatiblen verfügbar, bzw. auf Computern mit Intel-CPU)
N	Wenn Typ %p oder %s NEAR-Zeiger (16 Bit) sind (dies ist kein ANSI-Schalter und ist daher nur auf IBM-PCs und kompatiblen verfügbar, bzw. auf Computern mit Intel-CPU)

Tabelle 5-2: Formatierungsschalter in der Standard-IO

Neben den Schaltern können Ausgabelängen angegeben werden. Diese führen dazu, dass auch kleinere Zahlen mit entsprechender Breite ausgegeben werden, so dass man leicht übersichtliche Tabellen darstellen kann. Die angegebenen Breiten stellen immer minimale Sollwerte dar, die vom Rechner ignoriert werden, wenn die Einhaltung des angegebenen Formats zum Verlust von Vorkommastellen führen würde. Dies bedeutet, dass wenn eine Ausgabebreite von drei Stellen angegeben wird, der Wert aber größer als 999 ist, auf jeden Fall alle Vorkommastellen auf dem Bildschirm ausgegeben werden. Für Nachkommastellen gilt diese Regelung nicht, sie werden abgeschnitten, wenn mehr Nachkommastellen auftreten als angegeben wurden. Die Verwendung von Formatbreiten ist den Beispielen in Tabelle 5-3 zu entnehmen.

Formatierung der Ausgabebreite von Variablenwerten	
Format	Bedeutung
%10.2Lf	Fließkommazahl vom Typ long double mit mindestens zehn Stellen, davon zwei Nachkommastellen, rechtsbündig
%.2f	Fließkommazahl mit zwei Nachkommastellen, rechtsbündig
%6f	Fließkommazahl mit mindestens sechs Stellen, rechtsbündig
%4d	Integerzahl mit mindestens 4 Stellen, rechtsbündig
%04d	Integerzahl mit mindestens 4 Stellen und führenden Nullen
%-4d	Integerzahl mit mindestens 4 Stellen, linksbündig
%+6ld	long Integerzahl mit mindestens 6 Stellen, rechtsbündig und mit Vorzeichen
%20s	Zeichenkette mit mindestens 20 Zeichen Länge

Tabelle 5-3: formatierte Variablenausgabe in der Standard-IO

5.1.3. FLUCHTSYMBOLE ZUR AUSGABEGESTALTUNG

Das Fluchtsymbol „\“ (Backslash) leitet besondere Steuerungszeichen für die Ausgabe ein, wie z.B. einen Zeilenvorschub oder einen Tabulatorsprung (siehe Tabelle 5-4). Die anwendbaren Fluchtsymbole sind teilweise vom verwendeten Computersystem abhängig

Fluchtsymbole in Formatstrings und Variablen		
Symbol	Name	Bedeutung
\n	Carriage-Return / Linefeed	Zeilenvorschub/Wagenrücklauf
\r	Carriage-Return	Wagenrücklauf ohne Vorschub
\b	Backspace	Rückschritt
\a	Bell	Glocke (Soundausgabe)
\t	Tab	Tabulatorsprung (acht Zeichen)
\v	Vertical Tab	Vertikaltabulator (nur für Druck)
\f	Formfeed	Seitenvorschub (nur für Druck)
\\	Backslash	Ausgabe des Backslash-Zeichens
\''	Inch / Quote	Ausgabe des Anführungszeichens
\0	String-Terminator	String-Ende-Symbol (nur bei Stringvariablen)
\'	Apostroph	Hochkomma (nur für char-Variablen)

Tabelle 5-4: Fluchtsymbole in der Standard-IO

Besondere Aufmerksamkeit ist in der Tabelle 5-4 den letzten drei Fluchtsymbolen zu schenken. Wie beim Platzhalter „%%“ handelt es sich um Sonderfälle. Analog zur Ausgabe des Prozentzeichens erfolgt die Ausgabe des Zeichens „Backslash“ durch doppelte Angabe (\\). Auf die Doppelschreibweise muss unbedingt geachtet werden, wenn auf IBM-kompatiblen PCs Pfadangaben verarbeitet werden. Beim Einlesen der Pfadnamen von der Tastatur erfolgt die Übersetzung in die Doppelform für die Speicherung automatisch, nicht jedoch wenn der Backslash in einer Stringvariablen oder einer Stringkonstanten vorgegeben wird. Das besondere Fluchtsymbol für die Darstellung des Anführungszeichens ist notwendig, da die Hochkommata normalerweise den Formatstring begrenzen.

Das Zeichen „\0“ zeigt das Ende einer Zeichenkette an. Bei nahezu allen C/C++ Funktionen mit Zeichenketten wird dieses Zeichen automatisch gesetzt. Wenn Sie jedoch eigene Funktionen zur Manipulation von Zeichenketten schreiben, ist es nötig darauf zu achten, dass alle Zeichenketten auch ordnungsgemäß abgeschlossen sind. Im Verlauf des Kursus wird das Thema Zeichenketten ausführlich behandelt werden und an entsprechender Stelle wird auf die Verwendung des String-Ende Fluchtsymbols vertiefend eingegangen.

5.2. FORMATIERTE EINGABE MIT SCANF

Die Funktion *scanf* dient zum Einlesen von Variablen. Die Platzhalter sind dabei identisch mit denen der *printf*-Anweisung, allerdings darf der Formatstring mit den Platzhaltern (s.u.) keine Ausgabetexte enthalten. Zudem erfordert *scanf* die Angabe der Speicheradresse der Variablen (Variablenadresse), in welcher der eingelesene Wert abgelegt werden soll. Handelt es sich um eine Variable vom einfachen Datentyp (Integer-, char- oder Fließkommatypen), so wird dem Variablennamen einfach ein *„&“*-Zeichen vorangestellt (bei Zeichenketten ist dies nicht notwendig, da Strings komplexe Datentypen sind und ohnehin nur über Zeiger bzw. Adressen erreichbar sind).

```
Syntax:
   scanf (Formatstring, Variablenadressliste);

Beispiel:
   scanf ("%d%f", &Intvar, &floatvar);
```

```
//=============================================================
// Programm SCANF1.CPP
//=============================================================

#include <stdio.h>
#include <float.h>

int    iErste_Zahl  = 6;
double fZweite_Zahl = 1.0;

void main (void)
{
   printf ("Bitte 1. Zahl eingeben: ");
   scanf  ("%d", &iErste_Zahl);
   printf ("\n\n Erste Zahl = %5d", iErste_Zahl);
   printf ("\nBitte 2. Zahl eingeben: ");
   scanf  ("%lf", &fZweite_Zahl);
   printf ("\n\n Zweite Zahl = %8.2lf", fZweite_Zahl);
}
```

Prinzipiell kann man mit *scanf* auch mehrere Variable zugleich einlesen. Da es für den Anwender jedoch nicht ersichtlich ist, welche Variablen er eingeben soll und von welchem Typ diese sind, empfiehlt sich eine solche Vorgehensweise nicht.

Entscheidet man sich dennoch für das Einlesen mehrerer Werte in einer einzelnen *scanf*-Anweisung, so sind die einzelnen Werte bei der Eingabe durch Leerzeichen zu trennen:

```
//=============================================================
// Programm SCANF2.CPP
//=============================================================

#include <stdio.h>
#include <conio.h>

int iJahr  = 0;
int iMonat = 0;
int iTag   = 0;

void main (void)
{
   printf ("\n\nBitte Datum eingeben (TT MM JJJJ): ");
   scanf ("%d%d%d", &iTag, &iMonat, &iJahr);

   printf ("\n\nEingegeben wurde: %d:%d:%d", iTag, iMonat,
           iJahr);
}
```

Die Trennung durch Leerzeichen führt zwangsläufig zu einem Problem bei der Eingabe von Zeichenketten, da Texte üblicherweise Leerzeichen enthalten. Es empfiehlt sich daher zum Einlesen von Texten auf die dafür vorgesehenen Funktionen *gets* oder *fgets* auszuweichen, die im Kapitel über Zeichenketten ausführlicher behandelt werden.

5.3. EINLESEN EINES ZEICHENS MIT GETCHAR

Die Funktion *getchar* liest ein einzelnes Zeichen (bzw. dessen ASCII-Wert) ein und gibt es auf dem Bildschirm aus.

Überzählige Zeichen verbleiben im Tastaturpuffer.

```
Syntax:
   Integer_var = getchar ();

Beispiel:
   TestInt = getchar ();
   Charvar = getchar ();
```

Die Funktion wartet auf die Betätigung der Taste „Return“, bevor das Programm fortgesetzt wird. Gelegentlich können sich Probleme dadurch ergeben, dass bei einer vorherigen Eingabe das Returnzeichen von der Tastatur im Tastaturpuffer verbleibt. Es ist daher zumeist besser mit der Funktion *getch* zu arbeiten, die aber nicht auf allen Rechner- bzw. Betriebssystemplattformen verfügbar ist.

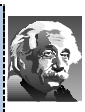
5.4. EINLESEN EINES ZEICHENS MIT GETCH

Die Funktion *getch* liest ein einzelnes Zeichen (bzw. dessen ASCII-Wert) vom Standard-Eingabegerät (Tastatur) ein, ohne es auf dem Bildschirm auszugeben.

```
Syntax:
   Integer_var = getch ();

Beispiel:
   TestInt = getch ();
   Charvar = getch ();
```

Im Gegensatz zur Funktion *getchar* wartet *getch* nicht auf die Returntaste, sondern setzt die Arbeit nach Betätigung einer beliebigen Taste unmittelbar fort.



Integrierte Entwicklungsumgebungen (IDE[4]) ermöglichen es, direkt aus dem Programmeditor heraus den Compiler, Linker und das zu testende Programm aufzurufen. Leider blenden fast alle IDE das Ausgabefenster von Konsolenprogrammen nach Beendigung des zu testenden Programms sofort wieder aus und schalten auf den Editor um. Um dies zu vermeiden, kann man am Ende des Programms einfach die folgende Anweisung einfügen:

getch ();

(gilt nur für Konsolenprogramme, nicht für die Erstellung von Windowsprogrammen)

Der Programmlauf bleibt dann bis zu einem Tastendruck stehen, so dass man in Ruhe die Ausgaben betrachten kann. Eine Zuweisung auf eine Variable ist nicht notwendig, da man den Tastendruck nicht auswerten möchte.

5.5. AUSGABE EINES ZEICHENS MIT PUTCHAR

Die Funktion *putchar* schreibt ein einzelnes Zeichen auf das Standard-Ausgabegerät (Bildschirm). Wird eine Zahl angegeben, so wird das Zeichen ausgegeben, das diesem ASCII-Wert entspricht. Für *putchar* gelten die gleichen Fluchtsymbole wie für die Funktion *printf*. Allerdings muss das Fluchtsymbol, da *putchar* nur einen char ausgibt, in einfachen Hochkommata angegeben werden.

```
Syntax :
   putchar (Integer_var);

Beispiel:
   putchar ('A');       // Ausgabe: "A"
   putchar (65);        // Ausgabe: "A"
   putchar (CharVar);   // Ausgabe: Char-Var.
   putchar ('A'+3);     // Ausgabe berechnete
```

[4] IDE = Integrated Development Environment

```
                          // Char-Variable
  putchar ('\n');         // Ausgabe Zeilen-
                          // vorschub
```

Die Funktion kann immer nur ein einzelnes Zeichen ausgeben, die Angabe von zwei Zeichen in den Hochkommata ist nur bei Fluchtsymbolen zur Textformatierung erlaubt.

5.6. STRUKTOGRAMMSYMBOL EIN- UND AUSGABE

Das Struktogrammsymbol für eine Ein- oder Ausgabe besteht aus einem Parallelogramm. In das Symbol wird die Bedeutung der Ein- bzw. Ausgabe eingetragen.

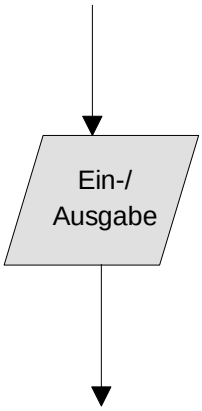


Abbildung 5-1 – Struktogrammsymbol Ein-/Ausgabe

Das Beispiel zeigt einen einfachen Berechnungsvorgang als Diagramm:

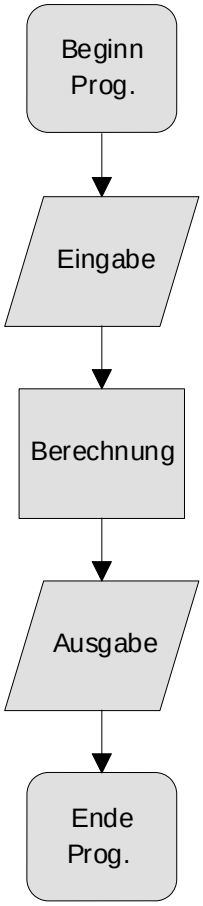


Abbildung 5-2 – Struktogramm: Berechnungsvorgang

6. EIN- UND AUSGABE MIT IO-STREAMS

In diesem Abschnitt wird das Stream-Konzept von C++ eingeführt. Die Stream-IO soll die Standard Ein- und Ausgabe ablösen, da es Erweiterungsmöglichkeiten für benutzerdefinierte Klassen bietet (siehe dazu Abschnitt über Klassen).

Mit der Einführung von C++ wurde zugleich der Schritt zu einer neuen Form der Ein-/Ausgabesteuerung unternommen. Statt des funktionalen Ansatzes, der in K&R- sowie im ANSI-C verfolgt wird, wurde ein objektorientiertes Konzept implementiert, welches eine Erweiterung der Ein-/Ausgabe um selbstdefinierte Klassen zulässt. Die prozeduralen Funktionen der Standard-IO von C lassen durch das Platzhalterkonzept in der Ausgabeformatierung (siehe Abschnitt über *printf*) eine Erweiterung der Liste von zu verarbeitenden Datentypen nicht zu. Eine solche Erweiterung wäre aber notwendig, damit auch auf benutzerdefinierte Klassen in der gleichen Art und Weise zugegriffen werden kann, wie auf die Standarddatentypen.

Genau dieser Mechanismus wird in C++ durch die Streams ermöglicht (wie man die Streams für eigene Klassen verwenden kann, ist im Abschnitt über Klassen detailliert erklärt). An dieser Stelle soll lediglich die Handhabung der Streams mit den Standarddatentypen erläutert werden.

Der Umstand, dass in C++ die Ausgabe über Streams erfolgt, heißt natürlich nicht, dass die „alten" ANSI-Funktionen nicht mehr existieren. Vielmehr kann in C++ eine Mischung beider Formen verwendet werden. Empfehlenswert ist dies natürlich nicht, es erleichtert jedoch den Übergang von C zu C++, insbesondere, wenn man zuvor relativ viel Programmieraufwand in Ausgabe-Formatierungsfunktionen gesteckt hat und diese Ergebnisse nach C++ übernehmen möchte.

ANSI-C++ Compiler mit STL-Unterstützung benutzen für die Stream-IO Headerdateinamen ohne die .h-Extension:

```
#include <iostream>
#include <iomanip>
```

Compiler ohne STL-Unterstützung verlangen hingegen die entsprechende Extension:

```
#include <iostream.h>
#include <iomanip.h>
```

Einige eher exotische Compiler verwenden anstelle der .h-Extension für C++-Headerdateien die Extension .h++, diese Form entspricht aber nicht der ANSI-Norm.

Das vorliegende Skript geht vom Defakto-Standard aus, d.h. einem STL-fähigen C++ Compiler.

Um die C++ Stream-Klassen nutzen zu können, muss die Headerdatei *iostream* in das Programm eingebunden werden. C++ erzeugt dann automatisch die drei Standard-IO-Streams *cout* (Standardausgabe, d.h. Bildschirm), *cin* (Standardeingabe, d.h. Tastatur) und *cerr* (Standardfehlerausgabe, ebenfalls Bildschirm). Die Streams *cout*, *cin* und *cerr* lösen somit die logischen Standard-Dateien *stdout*, *stdin* und *stderr* aus ANSI-C ab.

ANSI-C bietet neben den drei Standard-Dateien noch eine weitere, die direkt mit dem Drucker verbunden ist. Diese trägt den Namen stdprn.
Ein entsprechender IO-Stream ist unter dem Namen cprn *vorgesehen, in vielen Compilern aber nicht implementiert (z.B. Borland C++ Version 5).*

Die Ein- und Ausgabe über Operatoren ist für alle Standarddatentypen bereits vordefiniert und kann bei Bedarf auf eigene Datenklassen ausgedehnt werden (siehe Kapitel über Klassen). Vordefiniert sind dementsprechend:

Vordefinierte Datenklassen aus ANSI-C für Stream-IO		
signed short	unsigned short	float
signed int	unsigned int	double
signed long	unsigned long	long double
signed char	unsigned char	void *
signed char *	unsigned char *	

Tabelle 6-1: Vordefinierte Datenklassen aus ANSI-C für Stream-IO

6.1. AUSGABEN MIT STREAMS

Ausgaben erfolgen mit Hilfe der Klasse *ostream*. Die Objekte *cout* und *cerr* sind Instanzen dieser Klasse und stehen automatisch zur Verfügung, beide Objekte schreiben (standardmäßig) auf den Bildschirm.

6.1.1. AUSGABE MIT DEM OPERATOR <<

Wesentlichster Unterschied zwischen der Ausgabe in ANSI-C und C++ ist die Verwendung eines Operatorsymbols anstelle der Funktion *printf* (siehe *printf*). Der Operator << wird durch die *ostream*-Klasse überladen und hat hier nicht mehr die Bedeutung des *shift*-Operators. Er dient dazu, die Daten in Richtung des Ausgabestroms zu „schieben“ und kann hier gelesen werden als „nach *cout* wird geschoben…“

```
Syntax:
ostreamvar << var1 << var2 << ... << varN;
```

```
//=========================================================
// Programm OSTREAM1.CPP
//=========================================================

#include <iostream>

using namespace std;

void main (void)
{
   int iZahl = 7;

   cout << "Hallo Welt ";
   cout << 15;
   cout << "\n";
   cout << iZahl;
   cout << "\n";
}
```

Da der Operator von links nach rechts abgearbeitet wird und das Ergebnis der Stream-Ausgabe wiederum der angegebene Stream ist, kann man die gewünschten Ausgaben (mit unverändertem Ergebnis) einfach aufreihen:

```
//=========================================================
// Programm OSTREAM2.CPP
//=========================================================

#include <iostream>

using namespace std;

void main (void)
{
   int iZahl = 7;

   cout << "Hallo Welt " << 15 << "\n" << iZahl << "\n";
}
```

Wie bei ANSI-C kann auch in C++ mit *"\n"* ein Zeilenvorschub erzwungen werden, da man hier bei Bedarf die gleichen Fluchtsymbole verwenden kann (siehe auch Abschnitt über ANSI-C). Eine Alternative zu den Fluchtsymbolen (siehe Tabelle Fluchtsymbole) besteht in der Verwendung der sogenannten Manipulatoren (siehe Abschnitt über Manipulatoren).

6.1.2. AUSGABE MIT OSTREAM-METHODEN

Neben dem Operator << gibt es in der Klasse *ostream* die Möglichkeit, interne Methoden der Klasse direkt aufzurufen. Diese internen Funktionen werden normalerweise durch den Operator << automatisch korrekt aufgerufen (unterschieden anhand des auf den Operator folgenden Datentyps), so dass der Aufruf über Operator immer das Mittel der Wahl sein sollte. Gelegentlich gibt es jedoch Situationen, in denen die Kenntnis dieser internen Methoden von Vorteil ist. Die internen Methoden *put* und *write* werden hauptsächlich verwendet, wenn man eigene Datentypen in Standardausgabe einbinden muss.

6.1.2.1. AUSGABE MIT PUT

Die Methode *put* schreibt ein einzelnes Zeichen in den Ausgabestrom und ist damit identisch zur ANSI-C-Funktion *putchar*.

```
Syntax:
   ostream& put (char Zeichen);
   ostream& put (signed char Zeichen);
   ostream& put (unsigned char Zeichen);
```

```
#include <iostream>

using namespace std;

void main (void)
{
   cout.put ('A');     // ist identisch mit: cout << 'A'
}
```

Die Methode *put* hat den Vorteil, dass mit dieser auch Steuerzeichen an den Ausgabestrom gegeben werden können, die nicht als Fluchtsymbole definiert sind.

Es gibt aber auch einen deutlichen Unterschied zwischen der C und der C++ Ausgabe.

Die C++ Ausgabe interpretiert (im Gegensatz zum *printf*) einen Wert zunächst einmal immer als Zahl, nicht wie im nachstehenden Beispiel als char:

```
//============================================================
// Programm BACKSPAC.CPP
//============================================================

#include <stdio.h>

void main (void)
{
   printf ("12%c%c  ", 8, 8); // "12" schreiben und mit
                              // Cursor gleich wieder
```

```
                                    // zurückgehen und
                                    // überschreiben
}
```

Der einfache Ansatz, dies in C++ über den Ausgabeoperator zu lösen schlägt fehl, da die Acht als Wert interpretiert wird. Erst wenn die Acht ausdrücklich zu einem Character mit dem ASCII-Wert Acht gemacht wird (siehe Casting) oder man die *put*-Methode verwendet, funktioniert es.

```
//============================================================
// Programm BACKSPA2.CPP
//============================================================

#include <iostream>

using namespace std;

void main (void)
{
   cout << 12 << 8 << " \n";      // ergibt Ausgabe "128"
   cout << 12 << (char)8 << " ";  // ergibt Ausgabe "1"

   cout << 12;
   cout.put ((char)8);          // entspricht: cout << (char)8;
   cout << "\n";
}
```

6.1.2.2. AUSGABE MIT WRITE

Die Methode *write* dient zur Ausgabe von Zeichenketten unter Angabe einer Ausgabelänge.

```
Syntax:
  ostream& write (const char *Zeichenkette,
                  int Anzahlzeichen);
  ostream& write (const signed char *Zeichenkette,
                  int Anzahlzeichen);
  ostream& write (const unsigned char *Zeichenkette,
                  int Anzahlzeichen);
```

```
#include <iostream>

using namespace std;

void main (void)
{
   cout.write ("ABCDE", 3); // Ausgabe: ABC
   cout << "\n";
   cout.write ("ABCDE", 9); // Ausgabe: ABCDE§$&%
                            // also 4 Zeichen unvorhersehbarer
                            // Datenmüll
}
```

Besonders wichtig ist, dass die Funktion *write* die angegebene Ausgabelänge auf jeden Fall einhält. Ist die Zeichenkette länger, dann ist die Sache recht unproblematisch, denn es wird nur der Anfang der Zeichenkette (entsprechend der angegebenen Länge) ausgegeben.

Ist die Zeichenkette jedoch kürzer, so kümmert sich der C++-Compiler sich nicht um das Stringende-Symbol *'\0'* (siehe auch Abschnitt über Zeichenketten), sondern gibt immer die angegebene Länge an Zeichen aus – unerheblich was für Datenmüll auch folgen mag!

6.1.3. AUSGABEFORMATIERUNG UND MANIPULATOREN

Mit der Verwendung von << als Ausgabeoperator ist zwangsläufig auch ein neues Konzept der Formatierung verbunden, denn die in ANSI-C übliche Formatierungszeichenkette (siehe Abschnitt über Ausgabe mit *printf*) entfällt in C++. Die Formatierung erfolgt stattdessen durch Methoden und Operatoren der Klasse *ostream* und sogenannte Manipulatoren, die sich in die Ausgabekette einfügen lassen.

6.1.3.1. AUSGABENBREITE

Häufig ist es notwendig, die Ausgabebreite einer Zahl oder eines Textes zu bestimmen. In ANSI-C wird Ausgabebreite im Formatstring der *printf*-Funktion angegeben. Da dieses Vorgehen sich nur schlecht oder gar nicht auf eigene Datentypen ausweiten lässt, geht C++ hier einen anderen Weg.

Im objektorientierten Ansatz ist die Ausgabebreite eine Eigenschaft (engl. „Property“) des Ausgabeobjektes (hier *cout*), die mit der Methode *width* gesetzt werden kann. Die Methode *width* bestimmt aber lediglich die Feldbreite der nächsten (!) Ausgabeoperation. Wenn also mehrere Ausgaben über Operatoren aufgereiht werden, wird nur die erste gemäß der *width*-Angabe formatiert:

```
Syntax:
   int streamvar.width ();
   int streamvar.width (int Anzahlzeichen);
```

```
//=============================================================
// Programm MANIP1.CPP
//=============================================================

#include <iostream>

using namespace std;

void main (void)
{
```

```
   // so ist es richtig - nur Euro-Anteil wird formatiert
   cout << "Euro";
   cout.width (5);    // wirkt nur auf nächste Ausgabe!
   cout << 123 << "," << 99 << "\n";
   // Ausgabe: Euro··123,99           · ist hier Leerzeichen

   // Falsch: "Euro" wird formatiert, nicht Euro-Betrag
   cout.width (5);
   cout << "Euro" << 123 << "," << 99 << "\n";
   // Ausgabe: ·Euro123,99            · ist hier Leerzeichen

   // Euro-Anteil und Cent werden formatiert
   cout << "Euro";
   cout.width (5);   // wirkt nur auf nächste Ausgabe!
   cout << 123 << ",";
   cout.width (5);
   cout << 99 << "\n";
   // Ausgabe: Euro··123,···99         · ist hier Leerzeichen
}
```

Wie man leicht erkennen kann, hat *cout* nach der Ausgabe das auszugebende Format wieder „vergessen“ und gibt alle weiteren Daten so aus, als wäre nie eine Formatierung angegeben worden. Tatsächlich ist dieses Verhalten in C++ beabsichtigt. Die Formatierungsbreite wird nur kurz (d.h. bis zur nächsten Ausgabe) zwischengespeichert und anschließend wieder gelöscht.

Die Formatierung über die Methode *width* neigt deutlich zur Unübersichtlichkeit, da die Ausgaben (die sonst aufgereiht in einer Zeile stehen) auseinandergerissen werden müssen. Aus diesem Grund wurden sogenannte „Manipulatoren“ eingeführt, die das gleiche Verhalten innerhalb der Aufreihung mit dem Operator << erzwingen. Statt der Funktion *width* kann man somit einfach den Manipulator *setw* verwenden:

```
//============================================================
// Programm MANIP2.CPP
//============================================================

#include <iostream>
#include <iomanip>

using namespace std;

void main (void)
{
   // nur Euro-Anteil wird formatiert
   cout << "Euro" << setw(5) << 123 << "," << 99 << "\n";
   // Ausgabe:   Euro··123,99          · ist hier Leerzeichen

   // Falsch: "Euro" wird formatiert, nicht Euro-Betrag
   cout << setw(5) << "Euro" << 123 << "," << 99 << "\n";
```

```
   // Ausgabe:   ·Euro123,99           · ist hier Leerzeichen

   // Euro-Anteil und Cent werden formatiert
   cout << "Euro" <<setw(5) <<123 <<"," <<setw(5) << 99
        << "\n";
   // Ausgabe:   Euro··123,···99       · ist hier Leerzeichen
}
```

Da jetzt die Formatierung und der Wert unmittelbar aufeinander folgen, kommen Formatierungsfehler wie in der zweiten Beispielzeile (die häufig auftreten, wenn die Ausgabekette nachträglich verändert wird und man die vorhergehenden Zeilen nicht kontrolliert) nur noch sehr selten vor.

Werden *width* bzw. *setw* ohne Parameter aufgerufen, so liefern diese die gerade eingestellte Ausgabebreite zurück:

```
#include <iostream>

using namespace std;

void main (void)
{
   cout.width (5);
   cout << cout.width () << "\n";   // Ausgabe: 5
}
```

6.1.3.2. AUSGABEPRÄZISION

Eine etwas andere Steuerung als bei der Methode *width* ist für die Ausgabe einer Fließkommazahl notwendig. In ANSI-C wird Breite und Präzision (Anzahl der Nachkommstellen) im Formatstring der *printf*-Funktion angegeben (siehe *printf*).

Die Lösung in C++ geht einen etwas anderen Weg – der leider dazu führt, dass man die Anzahl der Nachkommastellen nicht mehr direkt beeinflussen kann.

Auch die Fließkommazahlen-Präzision ist als Eigenschaft des Ausgabeobjektes *cout* implementiert, die allerdings (im Gegensatz zur Methode *width*) dauerhaft eingestellt werden kann. D.h. die zuletzt eingestellte Präzision bleibt erhalten, bis sie erneut geändert wird. Gesetzt wird die Ausgabepräzision mit der Methode *precision*, welche die Gesamtpräzision als Summe der Vor- und Nachkommastellen angibt (Voreinstellung ist der Wert Sechs):

```
Syntax:
   int streamvar.precision ();
   int streamvar.precision (int Anzahlzeichen);
```

```
//============================================================
// Programm MANIP3.CPP
//============================================================

#include <iostream>

using namespace std;

void main (void)
{
   cout.precision (7);  // insgesamt 7 Stellen ausgeben!
   cout << 1.23456789 << "\n"; // Ausgabe : 1.234568
   cout << 12.3456789 << "\n"; // Ausgabe : 12.34568
   cout << 123.456789 << "\n"; // Ausgabe : 123.4568
   cout << 1234.56789 << "\n"; // Ausgabe : 1234.568

   cout << 12345.6789 << "\n"; // Ausgabe : 12345.68
   cout << 123456.789 << "\n"; // Ausgabe : 123456.8
   cout << 1234567.89 << "\n"; // Ausgabe : 1234568
   cout << 12345678.9 << "\n"; // Ausgabe : 1234568e+07
   cout << 123456789  << "\n"; // Ausgabe : 123456789
}
```

Man beachte, dass im Gegensatz zu ANSI-C der Dezimalpunkt nicht als Ausgabestelle für die Anzahl der Zeichen mitgezählt wird! Genau wie bei ANSI-C hingegen werden grundsätzlich alle Vorkommastellen mit ausgegeben, auch wenn dadurch das angegebene Ausgabeformat zerstört wird. Wenn nötig wird bei den Nachkommastellen automatisch (ab einschließlich 0.5) aufgerundet:

```
//============================================================
// Programm MANIP4.CPP
//============================================================

#include <iostream>

using namespace std;

void main (void)
{
   cout.precision (3);  // insgesamt 3 Stellen ausgeben!

   cout << 1.200 << "\n"; // Ausgabe : 1.2
   cout << 1.201 << "\n"; // Ausgabe : 1.2
   cout << 1.202 << "\n"; // Ausgabe : 1.2
   cout << 1.203 << "\n"; // Ausgabe : 1.2
   cout << 1.204 << "\n"; // Ausgabe : 1.2
   cout << 1.205 << "\n"; // Ausgabe : 1.21
   cout << 1.206 << "\n"; // Ausgabe : 1.21
   cout << 1.207 << "\n"; // Ausgabe : 1.21
   cout << 1.208 << "\n"; // Ausgabe : 1.21
   cout << 1.209 << "\n"; // Ausgabe : 1.21
}
```

Genau wie mit *setw* für die Methode *width* gibt es auch für die Präzision einen Manipulator (*setprecision*), der die Handhabung etwas erleichtern kann:

```
#include <iostream>
#include <iomanip>

using namespace std;

void main (void)
{
   cout << setprecision (7) << 1.2345 << "\n";
}
```

Werden *precision* oder *setprecision* ohne Parameter aufgerufen, so liefern diese die gerade eingestellte Präzision zurück. Um eine korrekte Ausgabe mit einer bestimmten Anzahl von Nachkommastellen zu erreichen, ist daher ein Griff in die Trickkiste nötig:

```
#include <iostream>
#include <iomanip>
#include <math.h>

using namespace std;

void main (void)
{
   double x = 123.456;

   cout << (long)x << "."  // nur ganzzahliger Anteil
                           // Nachkommastellen*100, runden
                           // und ganzzahligen Anteil bilden:
        << (long)(((x - (long)x) * 100) + 0.5) << "\n";
}
```

Etwas einfacher geht es, wenn man die beiden Berechnungen als Makrofunktionen (inline) definiert. Der genaue Mechanismus der Makroprogrammierung und der Makrofunktionen sind im Kapitel über die Präprozessoranweisungen enthalten und werden daher an dieser Stelle nicht näher erläutert:

```
//============================================================
// Programm MAKRO.CPP
//============================================================

#include <iostream>
#include <math.h>

using namespace std;
```

```
inline double vorkomma (double x)
   {return (floor(x));};
inline long nachkomma (double x, int y)
   {return((long)(((x-(long)x)*pow(10,y))+0.5));};

void main (void)
{
   double x = 123.456;
   cout << vorkomma (x) << "." << nachkomma (x,2) << "\n";
}
```

Die inline-Makrofunktion *vorkomma* ermittelt die Vorkommastellen durch Abschneiden der Nachkommastellen - diese Funktion ist recht simpel. Die inline-Makrofunktion *nachkomma* liefert die angegebene Anzahl an Nachkommastellen als ganzzahligen Wert, der korrekt gerundet wurde.

```
#include <iostream>

using namespace std;

void main (void)
{
   cout.precision (10);
   cout << cout.precision () << "\n";   // Ausgabe: 10
}
```

Die folgende Funktion hingegen macht davon Gebrauch, dass in C++ noch immer die „alten“ ANSI-C Funktionen enthalten sind:

```
#include <stdio.h>

char *outfloat (double x, int v, int n)
{
   char sOut [255] = "";
   sprintf (sOut, "%*.*lf", v, n, x);
   return (sOut);
}
```

Die Formatierungsanweisung wird hier zusammen mit der Variablen übergeben, im Stil von ANSI-C aufbereitet und anschließend als Zeichenkette zurückgegeben:

```
//=========================================================
// Programm OUTFLOAT.CPP
//=========================================================

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

```
using namespace std;

char *outfloat (double x, int v, int n)
{
   char sOut [255] = "";
   sprintf (sOut, "%*.*lf", v, n, x);
   return (sOut);
}

void main (void)
{
   double fx = 1234.123456;
   cout << outfloat (fx, 8, 2) << "\n";
   // Ausgabe: ···1234.12
}
```

6.1.3.3. ZEILENVORSCHUB UND STRINGENDE

Eine etwas vereinfachte Handhabung gibt es auch für die Ausgabe des Zeilenvorschubzeichens. Statt des bisher gezeigten *"\n"*, dessen Backslash-Zeichen auf den meisten deutschen Tastaturen nur umständlich zu erreichen ist und welches man allzu leicht einmal vergisst (oder mit "/n" verwechselt), kann der Manipulator *endl* verwendet werden.

Die Verwendung des Manipulators hat aber noch einen entscheidenden Vorteil – nicht in allen Betriebssystemen wird der Zeilenvorschub durch CR/LF dargestellt. Die Manipulator-Konstante *endl* kann daher relativ einfach in den Compilern umdefiniert werden, wodurch Quellcodes portabel bleiben:

```
#include <iostream>

using namespace std;

void main (void)
{
   cout << "Hallo Welt" << endl;
}
```

Das Stringendesymbol *ends* wird in C++ in gleicher Form behandelt, wie ein Zeilenvorschub. Wie bei *sprintf* - welches eine Variante der ANSI-C Ausgabefunktion *printf* ist (siehe Abschnitt über Zeichenketten), mit der auf Zeichenketten ausgegeben werden kann - existiert auch für die Streamausgabe eine Möglichkeit die Ergebnisse auf einen String umzuleiten. Der genaue Mechanismus ist im Kapitel über Zeichenketten beschrieben.

```
//=========================================================
// Programm STRING.CPP
//=========================================================

#include <iostream>
#include <strstream>

using namespace std;

void main (void)
{
   char mySz [256];                // Zeichenkette
   ostrstream myStream(mySz,256); // an mySz gebundener
                                   // Stream

   myStream << 126 << ends;      // String setzen, Abschluss
   cout << mySz << endl;         // String ausgeben
}
```

Wird im obigen Beispiel das Schieben von *ends* auf den Stream vergessen, so gibt das Ausgabeobjekt *cout* den Text soweit aus, bis es irgendwo im Speicher auf ein Stringendesymbol trifft.

6.1.3.4. FÜLLZEICHEN

Eine in ANSI-C häufig vermisste Eigenschaft ist in C++ neu hinzugekommen, die Angabe des Füllzeichens. Wird die Ausgabebreite mit *width* oder *setw* gesetzt, dann erzeugt der Rechner standardmäßig Leerzeichen um auf die gewünschte Breite aufzufüllen. In ANSI-C war lediglich ein auffüllen mit Nullen als Alternative vorgesehen (siehe Formatangaben bei *printf*). In C++ können nun beliebige Zeichen diese Rolle übernehmen, gesteuert über die Methode *fill*.

```
#include <iostream>
#include <iomanip>

using namespace std;

void main (void)
{
   cout.fill ('*');                  // Füllzeichen umsetzen
   cout << setw(5) << 3 << " " << setw(5) << 4 << endl;
}
```

Wie die Methode *precision* behält auch *fill* die zuletzt getätigte Einstellung im Speicher, um sie fortan zu verwenden. Es ist also ggf. notwendig, die Voreinstellung des Füllzeichens abzurufen und zwischen zu speichern, um nach einer Ausgabe den vorhergehenden Zustand des Streams wiederherstellen zu können. Dazu kann man die Methode *fill*

ohne Parameter aufrufen, sie liefert dann das eingestellte Füllzeichen zurück:

```
Syntax:
   int streamvar.fill ();
   int streamvar.fill (char Füllzeichen);
```

```
//============================================================
// Programm FILL.CPP
//============================================================

#include <iostream>
#include <iomanip>

using namespace std;

void main (void)
{
   char oldfill;

   oldfill = cout.fill ();  // altes Füllzeichen puffern
   cout.fill ('X');         // neues Füllzeichen setzen
   cout << setw(5) << 3 << " " << setw(5) << 4 << endl;
   cout.fill (oldfill);     // Füllzeichen wiederherstellen
}
```

Natürlich gibt es auch für die Methode *fill* einen Manipulator *setfill*, der das Umschalten innerhalb der Ausgabeverkettung erlaubt:

```
#include <iostream>
#include <iomanip>

using namespace std;

void main (void)
{
   cout << setfill('X') << setw(5) << 3 << endl;
   cout << setfill(' ') << setw(5) << 4 << endl;
}
```

6.1.3.5. PUFFER LEEREN

Eine gelegentlich nützliche Funktion ist das Leeren der IO-Puffer von Hand. Dazu dient bei der Stream-Ein-/Ausgabe der Manipulator *flush*. Dieser Manipulator wird bei Aufruf von *endl* und *ends* automatisch ausgeführt.

```
//============================================================
// Programm FLUSH.CPP
//============================================================

#include <iostream>
```

```
#include <iomanip>

using namespace std;

void main (void)
{
   cout << "hallo" << flush;
}
```

6.1.3.6. ZAHLENFORMATE

C++ stellt drei Manipulatoren für die Zahlenformatkonvertierung zur Verfügung: *dec* (Dezimalformat, Voreinstellung), *hex* (Hexadezimal-format) und *oct* (Oktalformat). Zu beachten ist, dass diese Einstellungen genau wie das Füllzeichen gespeichert werden und solange Bestand haben, bis sie durch einen anderen Zahlenformat-Manipulator wieder aufgehoben werden.

```
//============================================================
// Programm ZFORMAT.CPP
//============================================================

#include <iostream>
#include <iomanip>

using namespace std;

void main (void)
{
   cout << hex << 12 << " " << 13 << dec << endl;
   //Ausgabe: c d

   cout << setfill('0') << hex << setw(2) << 12
        << dec << setfill(' ') << endl;       // Ausgabe:0ct
}
```

6.1.3.7. AUSGABEFLAGS

Neben den bisher behandelten Methoden und Manipulatoren lässt sich die Ausgabe auch noch über eine Reihe von sogenannten „Flags“ steuern. Flags sind üblicherweise Bitfelder, deren Zustand (Bit x ist gesetzt oder gelöscht) eine Auskunft darüber geben, welches Verhalten gewünscht ist. Diese Flags sind Teil der übergeordneten Klasse *ios* (Vaterklasse von *ostream*, siehe dazu Abschnitte Klassen und Vererbung) und werden an *cout* vererbt. Sie gelten somit für alle Streams, egal ob diese sich auf Bildschirm, Zeichenketten oder Dateien beziehen.

Für die Ausgabesteuerung gibt es eine ganze Reihe von Flags, mit denen man das Verhalten des Ein- und Ausgabestroms steuern kann. Diese Flags zerfallen in mehrere Gruppen, deren Flags sich z.T. gegenseitig ausschließen:

Flaggruppen der Stream-IO	
Gruppe ios::adjustfield	
ios::left	Ausgabe erfolgt linksbündig, es wird nach dem Variableninhalt mit dem (mit „fill“ gesetzten) Füllzeichen aufgefüllt.
ios::right	Ausgabe erfolgt rechtsbündig, es wird vor dem Variableninhalt mit dem (mit „fill“ gesetzten) Füllzeichen aufgefüllt.
ios::internal	Ausgabe erfolgt intern, d.h. es wird zwischen dem Vorzeichen und dem Variableninhalt mit dem (mit „fill“ gesetzten) Füllzeichen aufgefüllt.
Gruppe ios::basefield (besser über die Manipulatoren hex, dec und oct setzbar)	
ios::dec	Ausgabe erfolgt im Dezimalsystem (Basis 10)
ios::hex	Ausgabe erfolgt im Hexadezimalsystem (Basis 16)
ios::oct	Ausgabe erfolgt im Oktalsystem (Basis 8)
Gruppe ios::floatfield	
ios::scientific	Ausgabe von Fließkommazahlen erfolgt im Exponential-format
ios::fixed	Ausgabe von Fließkommazahlen erfolgt im Standard-format

Tabelle 6-2: Flaggruppen der Stream-IO

Obwohl die Möglichkeit besteht, sollte man Gruppenflags nach Möglichkeit nicht „per Hand“ umschalten (durch eine binäre Oder-Operation), sondern immer mit der Methode *setf*, wie im folgenden Beispiel:

```
Syntax:
   long ostreamvar.flags  (void);
   long ostreamvar.setf   (ios::flagname);
   long ostreamvar.setf   (ios::flagname,
                           ios::flaggruppe);
   long ostreamvar.unsetf (long ios::flagname);
```

Der Manipulator *setf* gibt nach Ausführung automatisch den aktuellen Zustand des Flagfield als *long*-Variable zurück.

```
#include <iostream>

using namespace std;

void main (void)
{
   // hexadezimale Ausgabe einschalten. Die Angabe von
   // ios::adjustfield bewirkt, dass zugleich die Flags
   // ios::left und ios::right auf Null gesetzt werden
```

```
   cout.setf (ios::internal, ios::adjustfield);
   cout << 12 << endl;                          // Ausgabe: c
}
```

Neben den Gruppenflags gibt es eine Reihe von Einzelflags, die nach Wunsch ein- und ausgeschaltet werden können:

Einzelflags der Stream-IO	
ios::showbase	Bei der Ausgabe Zahlensystem anzeigen (bei Basis hex wird 0x vorangestellt, bei Basis oct eine Null)
ios::showpoint	Bei Fließkommazahlen auf jeden Fall Nachkommastellen anzeigen.
ios::uppercase	Bei Exponentialausgabe und Hexadezimalzahlen große Buchstaben verwenden
ios::showpos	Vorzeichen auch bei positiven Zahlen anzeigen
ios::unitbuf	flush für alle Streams nach der Datenübergabe aufrufen
ios::stdio	flush für stdout und stderr nach der Datenübergabe aufrufen

Tabelle 6-3: Einzelflags in IO-Streams

```
#include <iostream>

using namespace std;

void main (void)
{
   cout.setf (ios::showpos);
   cout << 12 << endl;          // Ausgabe : +12
}
```

Das Abschalten von Optionen gestaltet sich etwas schwieriger, da die Funktion *unsetf* nicht in allen Compilern definiert ist. Wenn es weder Funktion noch Manipulator zum Setzen oder Löschen einzelner Flags gibt, muss man mittels der Methode *flags* zunächst den Zustand des gesamten Bitfeldes ermitteln und mit den binären Operatoren dann das gewünschte Flag löschen oder setzen:

```
//============================================================
// Programm FLAGS.CPP
//============================================================

#include <iostream>

using namespace std;

void main (void)
{
   // Flag setzen
   cout.setf (ios::showpos);
```

```
   cout << 12 << endl;            // Ausgabe : +12

   // Flag löschen
   cout.flags (cout.flags() & (~ios::showpos));
   cout << 12 << endl;            // Ausgabe : 12
}
```

6.2. AUSGABE AUF DEN ERROR-STREAM

Die Verarbeitung von Ausgaben auf den Fehlerstrom erfolgt genauso wie auf den Standard-Ausgabestrom, wobei statt des Ziels *cout* der Stream *cerr* angegeben werden muss. Obwohl beide Streams auf den Bildschirm ausgeben, ist die Trennung der Ausgabeströme dennoch sinnvoll, da der Fehlerstrom auch auf eine Datei umgeleitet werden könnte, wo er dann als Fehlerprotokoll dient.

```
#include <iostream>

using namespace std;

void main (void)
{
   cerr << "Fehler\n";
}
```

Für *cerr* gelten natürlich die gleichen Methoden und Manipulatoren wie für den Ausgabestream *cout*.

6.3. EINGABEN MIT STREAMS

Eingaben erfolgen mit Hilfe der Klasse *istream*, die sich, genau wie die Klasse *ostream* von der Vaterklasse *ios* ableitet (s.o.). Das Objekt *cin* ist eine Instanz dieser Klasse und steht automatisch zur Verfügung. Das Objekt *cin* liest standardmäßig von der Tastatur ein.

6.3.1. EINGABE MIT DEM OPERATOR >>

Entsprechend dem << Zeichen für die Ausgabe, werden mit dem Operator >> Eingabedaten aus dem Eingabestrom in Variablen geschoben:

```
#include <iostream>

using namespace std;

void main (void)
{
   int i=0, j=0, k=0;

   cout << "Datumseingabe (mit Leerzeichen trennen):\n";
   cin  >> i >> j >> k;           // so besser nicht !!!
```

```
   cout << i << "." << j << "." << k << endl;
}
```

Eine Eingabe erfolgt jetzt durch Eingabe einer Zahl und Bestätigung mit der Returntaste. Werden mehrere Zielvariable aufgereiht, so werden die einzelnen Teile durch Leerzeichen getrennt oder nacheinander eingegeben (jeweils bestätigt mit der Returntaste). Hier stellen sich jedoch grundsätzlich die gleichen Fragen wie in ANSI-C:

- Wie soll der Anwender erkennen, wie viele Werte bzw. Zeichen einzugeben sind?
- Falls mehrere Eingaben erforderlich oder gewünscht sind, wie sind diese voneinander zu trennen?

Daher ist die folgende Darstellung für den Anwender in den meisten Fällen durchschaubarer:

```
//============================================================
// Programm EINGABE1.CPP
//============================================================

#include <iostream>

using namespace std;

void main (void)
{
   int i=0, j=0, k=0;

   cout << "DATUMSEINGABE\nTag  : ";
   cin  >> i;

   cout << "Monat: ";
   cin  >> j;

   cout << "Jahr : ";
   cin  >> k;               // so ist es besser!!!
   cout << i << "." << j << "." << k << endl;
}
```

Problematisch ist in jedem Fall die Eingabe von Zeichen, die nicht dem geforderten Datentyp entsprechen. Zeichen, die nicht verarbeitet werden können, da sie vom falschen Datentyp sind, werden in C++ schlicht ignoriert. Der Wert der eigentlich einzugebenden Variablen bleibt unbestimmt und damit zufällig (es sei denn, dass die Variable zuvor vorbelegt wurde).

Die Problematik falscher Eingaben ist bei C++ etwas besser gelöst als bei ANSI-C (siehe Abschnitt über *scanf*), wo Daten falschen Datentyps im Puffer verbleiben und mit größter Wahrscheinlichkeit auch die folgenden Eingabewerte fehlerhaft beeinflussen. In beiden Fällen (bei C und C++) ist es dennoch ratsam, Eingaben grundsätzlich nur auf Zeichenketten zu tätigen und diese dann auszuwerten. Ein entsprechendes Unterprogramm ist im Abschnitt über Zeichenketten beschrieben.

Objekte der Klasse *istream* überlesen grundsätzlich die sogenannten Whitespaces (d.h. Leerzeichen, Tabulatoren und Zeilenvorschubzeichen) und benutzen diese stattdessen als Trennsymbole für einzelne Variablen. Daher eignet sich *cin* nicht zum Einlesen von Zeichenketten, die üblicherweise ja Leerzeichen enthalten, um Worte innerhalb der Zeichenkette voneinander zu trennen (diese Einschränkung besteht analog zu den unten aufgeführten Beschränkungen der Funktion *scanf* in ANSI-C).

Im Gegensatz zu ANSI-C muss man in C++ den Namen der zu lesenden Variablen angeben und nicht deren Adresse (siehe Abschnitt über *scanf*). Dies führt zu einer wesentlich höheren Programmsicherheit, da Abstürze bei „vergessenen“ Zeigerreferenzen eine der Hauptursachen für Programmabbrüche in C sind. Möglich wird dies durch einen „echten“ Call-by-Reference-Mechanismus, der den Call-by-Pointer von ANSI-C ergänzt und weitgehend ablösen soll.

6.3.2. EINGABE MIT ISTREAM-METHODEN

Neben dem Operator >> gibt es auch in der Klasse *istream* die Möglichkeit, interne Methoden der Klasse aufzurufen. Diese internen Funktionen werden normalerweise durch den Operator >> automatisch, dem Kontext entsprechend, aufgerufen. Häufiger als bei der Ausgabe gibt es jedoch Situationen, in denen die gewünschte Eingabeform nur mit Hilfe der Methoden möglich ist.

6.3.2.1. EINGABE MIT GET

Die Methode *get* gibt es gleich in mehreren Varianten (die Methode ist also mehrfach „überladen“), jede mit einer leicht anderen Handhabung und unterschiedlichen Parameterleisten. Die Methode *get* akzeptiert, im Gegensatz zum Operator >> auch Leerzeichen als Eingaben.

```
Syntax:
   int istreamvar.get ();
```

Die einfachste Form der *get*-Methode liest das nächste Zeichen aus dem Eingabestrom und gibt dessen ASCII-Wert zurück. Da ASCII-Wert und *char* sich in ANSI-C und C++ nicht unterscheiden, kann man das Ergebnis entweder als Zahl oder als gelesenes Zeichen betrachten.

```
Syntax:
   istream& istreamvar.get (char& charvar);
```

Auch diese Form der *get*-Methode liest nur das nächste Zeichen aus dem Eingabestrom. Das Zeichen selbst wird jedoch nicht zurückgegeben, sondern auf die Variable geschrieben, die als Parameter übergeben wurde.

```
Syntax:
   istream& istreamvar.get (char* buf, int len,
                            char Delim = '\n');
```

Diese Form des *get* ist dazu geeignet Zeichenketten einzulesen. Dem Eingabestrom werden solange Zeichen entnommen und in der Puffervariablen (*buf*) abgelegt, bis die Methode entweder auf das angegebene Endzeichen (*Delim*) oder ein Dateiendezeichen stößt. Über die Funktion wird jedoch niemals mehr als *len-1* Zeichen in die Puffervariable übernommen. Die *get*-Methode sorgt zudem dafür, dass stets ein Stringende-Zeichen (*'\0'* bzw. *ends*) in die Puffervariable geschrieben wird. Der Parameter *Delim* (Delimiter) muss nicht angegeben werden, wenn es sich um die Return-Taste handeln soll, da dies die Voreinstellung ist (siehe auch Abschnitt über Default-Parameter).

```
#include <iostream>

using namespace std;

void main (void)
{
   char j [255];

   cin.get (j, 255);
   cout << "[" << j << "]" << endl;
}
```

6.3.2.2. EINGABE MIT GETLINE

Die Methode *getline* ist identisch mit der *get*-Variante, die in der Lage ist Zeichenketten zu lesen. Ein Unterschied zwischen *getline* und der angesprochenen *get*-Methode besteht nicht. Die Verwendung von *getline* ist trotzdem vorzuziehen, da sich die Bedeutung im Unterschied zum einfachen *get* leichter erschließt.

```
Syntax:
   istream& istreamvar.getline (char* buf, int len,
                                char Delim= '\n');
```

Die Methode *getline* dient, wie bereits erwähnt, dazu Zeichenketten einzulesen. Dem Eingabestrom werden solange Zeichen entnommen und in der Puffervariablen (*buf*) abgelegt, bis die Methode entweder auf das angegebene Endzeichen (*Delim*) oder ein Dateiende-Zeichen stößt. Es wird jedoch niemals mehr als *len-1* Zeichen in die Puffervariable übernommen. Die *getline*-Methode sorgt zudem dafür, dass stets ein Stringende-Zeichen (*'\0'* bzw. *ends*) in die Puffervariable geschrieben wird. Der Parameter *Delim* (Delimiter) muss nicht angegeben werden, wenn es sich um die Returntaste handeln soll, da dies ist die Voreinstellung ist.

```
//============================================================
// Programm EINGABE2.CPP
//============================================================

#include <iostream>

using namespace std;

void main (void)
{
   char j [255];

   cin.getline (j, 255);
   cout << "[" << j << "]" << endl;
}
```

6.3.2.3. EINGABE MIT PEEK

Die Methode *peek* dient dazu das nächste Zeichen zu betrachten, ohne es aus dem Datenstrom zu entnehmen, d.h. das Zeichen bleibt im Eingabepuffer (meist der Tastaturpuffer) und wird erst beim nächsten *get* oder *getline* entnommen.

```
Syntax:
   int istreamvar.peek ();
```

6.3.2.4. EINGABE MIT PUTBACK

Die Methode *putback* dient dazu ein bereits auf dem Stream gelesenes (oder ein beliebig anderes) Zeichen wieder in diesen zurückzustellen (oder hinzuzufügen).

```
Syntax:
   istream & istreamvar.putback(char);
```

```
#include <iostream>

using namespace std;

void main (void)
```

```
{
   char j [255];

   cin.putback ('A');  // der Eingabe wird A vorangestellt
   cin.getline (j, 255);
   cout << "[" << j << "]" << endl;
}
```

6.3.3. EINGABEFORMATIERUNG, MANIPULATOREN UND FLAGS

Der einzige relevante Manipulator für die Eingabe ist die Angabe, ob Whitespace-Zeichen (Leerzeichen, Tabulatoren usw.) bei der Eingabe ignoriert werden sollen. Dazu dient der Manipulator *ws*, dessen Einstellung erhalten bleibt, bis das entsprechende Flag wieder gelöscht wird (siehe oben). Wird das Flag *skipws* gelöscht, so wird auch jedes Leerzeichen als Eingabe aufgefasst, während sonst Leerzeichen nicht als Teil der Eingabe angenommen werden. Das Zulassen von Leerzeichen als Eingaben ist meist unsinnig, es ist immer besser stattdessen die Methode *getline* zu verwenden, um Zeichenketten mit Leerzeichen einlesen und nachbearbeiten zu können.

```
//============================================================
// Programm FLAGS2.CPP
//============================================================

#include <iostream>
#include <iomanip>

using namespace std;

void main (void)
{
   char j [255];

   // Überlesen von Blanks ausschalten
   cin.flags (cin.flags() & (~ios::skipws));

   cout << "String: ";
   cin  >> j >> ws;          // wieder einschalten
   cout << "[" << j << "]" << endl;
}
```

6.4. STRUKTOGRAMMSYMBOL EIN- UND AUSGABE

Das Struktogrammsymbol für eine Ein- oder Ausgabe besteht aus einem Parallelogramm.

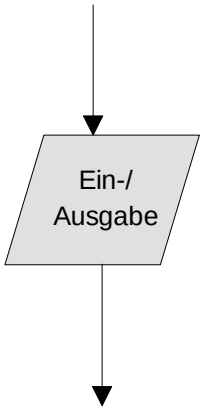


Abbildung 6-1 – Struktogrammsymbol Ein-/Ausgabe

In das Symbol wird die Bedeutung der Ein- bzw. Ausgabe eingetragen.

6.5. AUFGABENTEIL

Bei allen Aufgaben ist ein Schwierigkeitsgrad angegeben. Der Schwierigkeitsgrad bezieht sich nicht allein auf die Aufgabenstellung sondern ggf. auch auf den Umfang der Ausgabe.

6.5.1. AUFGABE 1 (SCHWIERIG)

Nachdem sich der Führer der "Revolutionären Bewegung", El Commandante Caramba Garcia erst kürzlich an die Regierung der tropischen Insel und Bananenrepublik Porto Banana geputscht hat, steht nun eine umfangreiche Neuorganisation der Güter, Ländereien, Privilegien und Aufgaben im Lande an.

Mit Blick auf den Inselstaat Santa Clementina, den um die Gunst der Touristen konkurrierenden Nachbarn, hat El Commandante zunächst verfügt, dass die Verwaltung durch den Einsatz von moderner Computertechnik auf Vordermann zu bringen ist.

Da El Commandante zudem für noch ausstehende Schmiergeldzahlungen dringend Geld benötigt, soll zunächst mit der Computerisierung des Steuersystems begonnen werden.

Die in Porto Banana gültige Währung ist der Nachos (Na). Jeder Nacho zu 100 Centimos (ctm).

Ihre erste Aufgabe ist es, ein Programm zu schreiben, welches aus der Eingabe des jährlichen Bruttoeinkommens die zu zahlenden Steuern berechnet.

In Porto Banana gilt ein gemischtes System aus Kopf- und Lohnsteuer.

Jeder Steuerzahler hat zunächst für jedes erwachsene Familienmitglied 50 Nachos Kopfsteuer pro Jahr zu zahlen, vom Rest sind für jedes Kind bleiben je 25 Nachos steuerfrei, vom dann noch verbleibenden Rest sind pauschal 20% Lohnsteuer zu zahlen.

Geben Sie die zu zahlenden Steuern sowohl in Nachos als auch in Euro an. Ein Nacho hat einen Wert von 0,23561 Euro.

Testen Sie das Programm mit einigen typischen Durchschnittsverdiensten auf Porto Banana:

Bauer, Ehefrau, 2 Großeltern, 5 Kinder, Jahresverdienst: 970 Na
Professor, Ehefrau, 2 Großeltern, 2 Kinder, Jahresverdienst: 1623 Na
Parteifunktionär, Ehefrau, 1 Kind, Jahresverdienst: 47000 Na
El Commandante, unverheiratet, offizieller Jahresverdienst: 100.000 Na

6.5.2. AUFGABE 2 (EINFACH)

Schreiben Sie ein Programm, welches den Verkaufspreis einer Ware in Porto Banana berechnet.

Errechnen Sie den Verkaufspreis aus dem Einkaufspreis, den Fixkosten (70 % des Einkaufspreises), der Mehrwertsteuer (15 % des bis dahin errechneten Verkaufspreises), sowie dem Gewinn (variabel in Prozent des Einkaufspreises).

Gewähren Sie bei Barzahlung 10 % Rabatt.

Geben Sie beide Endpreise (mit und ohne Rabatt), auf dem Bildschirm aus.

Testen Sie das Programm mit folgenden Einkaufspreisen:

100.00 Nachos
50.00 Nachos
76.98 Nachos

6.5.3. AUFGABE 2 (MITTEL)

Schreiben Sie ein Programm, welches die Unterhaltskosten eines Fahrzeugs in Porto Banana berechnet.

Lesen Sie bitte alle Daten über die Tastatur ein und geben Sie Eingaben und Ergebnisse nach der Berechnung formatiert auf dem Bildschirm aus.

Testen Sie das Programm bitte mit folgenden Werten:

Im letzten Jahr sind im Schnitt jeden Arbeitstag 20 KM beruflich gefahren worden und der Fahrzeughalter bekommt eine Kilometerpauschale 0.20 Nachos pro Doppelkilometer (d.h. für die Hälfte der gefahrenen KM).

Zudem zahlt der Halter pro Halbjahr 170.-- Nachos Steuern, jährlich 300.-- Nachos Versicherung und 378.63 Nachos Wartungs- und Reparaturkosten. Der Wagen verbraucht 8 Liter Benzin auf 100 KM.

Da wir die einzelnen Tankrechnungen nicht vorliegen haben, gehen Sie bitte von einem Durchschnittswert 1.29 Nachos je Liter aus..

Berechnen Sie die Gesamtaufwendungen im vergangenen Jahr und die Summe, die man über die Kilometerpauschale zurückbekommt.

6.5.3.1. ZUSATZFRAGE

Welche Werte müssten zusätzlich berücksichtigt werden, um die wirklichen Kosten und nicht nur die Unterhaltskosten zu berechnen?

7. BLÖCKE, LOGIK UND BEDINGTE VERZWEIGUNGEN

Blöcke sind zusammengehörige Sequenzen von Anweisungen, die wiederholt oder nur unter bestimmten Randbedingungen ausgeführt werden sollen. Die Anweisungen sind also unteilbar (d.h. sie müssen immer komplett oder gar nicht ausgeführt werden). Die bereits erwähnten Randbedingungen werden durch logische Aussagen beschrieben. Unter bedingten Verzweigungen sind dabei alle Anweisungen zu verstehen, die einen Programmfluss in Alternativen aufspalten, so dass gegebenenfalls Programmteile zusätzlich und/oder nicht ausgeführt werden. Eine bedingte Verzweigung enthält immer eine logische Bedingung und einen Anweisungsblock. Der Anweisungsblock kann jedoch so trivial und kurz sein, dass die Klammerung des Blocks entfallen kann (dies ist immer genau dann der Fall, wenn der Anweisungsblock aus genau einer Anweisung besteht).

7.1. BLÖCKE

Blöcke fassen syntaktisch mehrere Anweisungen zu einer einzigen Anweisung zusammen. Überall, wo nach einem Kontrollschlüsselwort eine Anweisung erwartet wird (bedingte Verzweigung, Schleifen), kann statt dessen auch ein Anweisungsblock eingefügt werden. Die öffnende, geschweifte Klammer *„{“* leitet einen Block ein, die schließende Klammer *„}“* beendet ihn. Blöcke können (theoretisch) beliebig ineinander verschachtelt werden, tatsächlich besitzen die meisten Compiler eine maximale Verschachtelungstiefe (Blockschachtelung), die dem jeweiligen Handbuch zu entnehmen ist.

Wie leicht zu erkennen ist, handelt es sich demnach auch beim Hauptprogramm (*main*) um einen Anweisungsblock, wobei das Hauptprogramm zusätzlich die Besonderheit aufweist, der erste Block zu sein, der abgearbeitet wird. Die immense Bedeutung von Blöcken für die Programmierung wird z.B. bei den bedingten Verzweigungen (s.u.) deutlich.

7.2. LOGISCHE VARIABLEN UND KONSTANTEN

Im Gegensatz zu den meisten Programmiersprachen kennt C keinen eigenen Datentyp für boolsche Variablen (logische Variablen). Stattdessen kann jeder skalare Datentyp (*int*, *short*, *long* oder *char*) zur Speicherung von logischen Werten verwendet werden. Dabei wird der Wert Null für den boolschen Wert *FALSE* verwendet und ein Wert ungleich Null für den boolschen Wert *TRUE*.

Aus diesem Umstand ergibt sich jedoch eine weitere Besonderheit von C/C++. So ist bei Umkehrung von Logikabfragen darauf zu achten, dass alle Zahlenwerte ungleich Null einer Abfrage auf „NOT FALSE“ entsprechen. „NOT TRUE“ hingegen ist auf einigen Systemen erfüllt, wenn der abzufragende Wert ungleich Eins ist (dies hängt aber stark vom

Maschinencode ab, den der Compiler erzeugt). Bedingte Verzweigungen fragen immer auf „NOT FALSE“ ab.

Die normalerweise in einer Programmiersprache vorhandenen symbolischen Konstanten *TRUE* und *FALSE* sind in C/C++ nicht vordefiniert.

Definieren Sie sich die logischen Konstanten TRUE und FALSE selbst. Fügen Sie dazu nach den #include-Anweisungen die folgenden beiden Zeilen ein:

```
#define FALSE 0
#define TRUE 1
```

Eine andere Möglichkeit besteht darin, dass Sie eine eigene Headerdate mit diesen Konstanten erzeugen (z.B. MeineDat.h) und diese dann mit einer #include-Zeile einbinden.

7.3. VERGLEICHE

Vergleiche werden benötigt, um anhand von Entscheidungen im Programm zu verzweigen. Dabei ist es gleichgültig, ob diese Entscheidungen nur durch die eingegebenen Daten oder durch direkte Benutzerentscheidungen bedingt sind - die Daten oder Eingaben müssen mit den im Programm vorgesehenen Lösungswegen verglichen werden. C/C++ stellt für Vergleiche die üblichen Operatoren zur Verfügung, die den nachstehenden Listen (Tabelle 7-1 und Tabelle 7-2) entnommen werden können:

Vergleichsoperatoren	
Operator	Bedeutung
==	Test auf Gleichheit
!=	Test auf Ungleichheit
>	Test auf Größer als
>=	Test auf Größer gleich
<	Test auf Kleiner als
<=	Test auf Kleiner gleich

Tabelle 7-1: Vergleichsoperatoren

Die letzten vier Operatoren bieten wenig Überraschung, es sind die üblichen Zeichen, die auch in der Mathematik Verwendung finden. Interessanter hingegen ist der Test auf Gleichheit der durch zwei Gleichheitszeichen dargestellt wird (==). Da das einfache Gleichheitszeichen (=) bereits durch die Zuweisung belegt ist und in C++ wegen der verwendbaren Seiteneffekte eine anweisungsabhängige Unterscheidung nicht getroffen werden kann, ist für den Vergleich ein

eigenes Zeichen notwendig (ein Beispiel für die Notwendigkeit ist weiter unten bei den Verzweigungen aufgeführt). Die meisten anderen Programmiersprachen gehen übrigens den umgekehrten Weg und verwenden eine besondere Zeichenkombination für die Zuweisung (wie z.B. in Pascal mit dem „:=“ Zeichen). In C++ hingegen hat man sich bewusst für diese Variante entschieden, da Zuweisungen in Programmen weit häufiger vorkommen als Vergleiche. Bei der Ungleichheit hat man das „!=“-Zeichen gewählt, da das „!“-Zeichen auch für das logische „NOT“ verwendet wird und die Zeichenkombination somit als „NOT EQUAL“ gelesen werden kann. Wenn man will, kann man sich bei der Zeichenkombination „!=“ auch eine Art durchgestrichenes Gleichheitszeichen vorstellen.

```
//============================================================
// Programm VERZW1.CPP
//============================================================

#include <iostream>

using namespace std;

#define TRUE  1
#define FALSE 0

int iZahl1   =  0;
int iZahl2   =  0;
int iZahl3   = -4;
int iZahl4   =  1;

void main (void)
{
   iZahl1 = iZahl2 > iZahl3 == iZahl4;
   cout << iZahl1;
}
```

Man beachte, dass die beiden Operatoren „==“ und „!=“ von geringerer Priorität (siehe Operatorentabelle) als die übrigen Vergleichsverknüpfungen sind. Vergleiche mit Zuweisungen müssen daher besonders gut durchdacht werden. Das Ergebnis der Zuweisung im folgenden Beispiel ist *TRUE* (Eins), da der Vergleich zwischen *iZahl2* und *iZahl3* zuerst erfolgt. Hätte der Vergleich auf Gleichheit höhere Priorität, wäre das Ergebnis *FALSE*. Eine vollständige Klammerung der Vergleiche ist daher oftmals der einzige Weg, die genaue Auswertung sicherzustellen und im Quellcode auch transparent zu machen.

Sparen Sie bei logischen Abfragen nicht mit Klammern, auch wenn diese letztlich vielleicht überflüssig sind. Die Lesbarkeit wird erhöht und Sie sind sicher, dass der Rechner auch wirklich den Vergleich ausführt, den Sie wollen.

Anstelle eines Vergleichs kann auch eine Variable direkt bewertet werden, dazu kann auch der logische „NOT“-Operator zu Hilfe genommen werden (*!*). Die folgenden Ausdrücke sind jeweils äquivalent zueinander:

```
integer_var == 0  // ist identisch mit !integer_var
integer_var != 0  // ist identisch mit integer_var
```

Zur Verknüpfung logischer Abfragen (z.B. in Bedingungen) sind in C/C++ eine Reihe von Operatoren vorgesehen, deren Priorität (siehe Operatortabelle) mit 11 und 12 eher gering ist. Im Einzelnen ist dies:

Logische Operatoren		
Bedeutung	Logischer Begriff	C/C++ Operator
Logisches „nicht“	NOT	!
Logisches „und“	AND	&&
Logisches „oder“	OR	||

Tabelle 7-2: Logische Operatoren

Für das logische, „Exklusiv-oder“ („entweder oder“, bzw. „XOR“) ist in C/C++ kein eigener Operator enthalten. Die „XOR“-Operation muss durch logische Verknüpfungen mit „AND“ und „OR“ ersetzt werden. In ungeklammerten Ausdrücken bindet der „AND“-Operator (*&&*) stärker als das logische „oder“ („OR“, ||). Die höhere Priorität ist also vergleichbar mit Vorrang einer Multiplikation gegenüber einer Addition:

```
A || B && C
```

Die Auswertung einer logischen Berechnung erfolgt ansonsten strikt von links nach rechts. Das C-Laufzeitsystem bricht die Auswertung jedoch ab, sowie das Resultat einer Bewertung zwingend feststeht. So wird z.B. der zweite Operand überhaupt nicht mehr betrachtet, wenn in einer „AND“-Verknüpfung bereits der erste Operand den Wert Null (*FALSE*) hat:

```
A && ((B || C) && (D || E))
```

Die Auswertung beginnt rechts, also bei *„B || C“*, da diese Variablen sich in der tiefsten Klammerungsebene befinden. Hat *„B || C“* den Wert Null (also *FALSE*), so wird der Ausdruck in der Klammer *„D || E“* nicht mehr ausgewertet, da das Ergebnis der logischen Formel ohnehin den Wert *FALSE* ergeben muss. Dies ist eine der Optimierungen, die in C/C++ automatisch durchgeführt werden. Obwohl syntaktisch korrekt, ist es daher sehr gefährlich, im nachgeordneten Teil einer logischen Auswertung noch Zuweisungen oder Inkrement bzw. Dekrementbefehle unterzubringen:

```
//-------------------------------------------------
// So bitte nicht
//-------------------------------------------------

#include <iostream>
#include <iomanip>

using namespace std;

int iA =  1;
int iB =  0;
int iC =  3;
int iD = -3;
int iF =  2;
int iG =  1;

void main (void)
{
    //-------------------------------------------------
    // Zur Syntax des IF-Befehls siehe unten...
    //-------------------------------------------------

    if (iA && ((iB || iC++) && ((iD = iE) || (iF += iG))))
    {
        cout << endl << "Bedingung ist erfüllt" << endl;
    }
}
```

Beachten Sie bitte, dass im Beispiel die Berechnungen zu *iD* und *iF* zusätzlich geklammert werden müssen, da die Priorität der Operatoren „+=“ und „=“ geringer ist, als die Priorität der logischen Verknüpfungen. Solche komplexen Konstruktionen sind insgesamt (in jeder Programmiersprache) sehr schwer lesbar.

*Komplexe logische Ausdrücke sind eine Quelle schwer auffindbarer Fehler. Im Beispiel würden alle mathematischen Anweisungen (*iC++, iD+=4 *und* iF=iG*) nicht ausgeführt werden, wenn* iE *den Wert* FALSE *hat.*

Fast alle Compiler optimieren Ausdrücke, die mit einem logischen UND verknüpft sind, indem sie die Ausdrucksequenz abbrechen, sobald ein Teilausdruck FALSE *ergibt.*

Dies ist jedoch abhängig vom Compiler bzw. der gewählten Optimierung.

7.4. BEDINGTE VERZWEIGUNGEN

Bedingte Verzweigungen werden verwendet, wenn in einem Programm (anhand der Daten oder einer Eingabe) eine Entscheidung zu treffen ist, die dazu führt, dass unterschiedliche oder zusätzliche Anweisungen

durchlaufen werden müssen. So ist z.B. die Berechnung der Steuerabzüge abhängig von der Steuerklasse.

7.4.1. DIE BEDINGTE BEWERTUNG

Die einfachste Art eine bedingte Verzweigung durchzuführen, besteht in der Verwendung des Operatorpaars für die bedingte Bewertung *„?:“*. Die Bewertung besteht aus einem Vergleich oder einer Berechnung, auf deren Grundlage eine der beiden nachstehenden Anweisungen durchgeführt wird:

```
Syntax:
   e1 ? e2 : e3;

Beispiel:
   Steuerklasse>1 ? Abzug=23 : Abzug=25;
```

Zunächst wird der Ausdruck *e1* bewertet. In Abhängigkeit vom Bewertungsergebnis (Ausdruck ist wahr oder falsch) wird dann entweder der Ausdruck *e2* oder der Ausdruck *e3* durchgeführt. Ist das Ergebnis des Ausdrucks *e1* gleich Null (*FALSE*), so wird *e3* bewertet, ist das Ergebnis von *e1* ungleich Null, so erfolgt die Bewertung von *e2*. Die bedingte Bewertung wird fast ausschließlich in Makros verwendet, kann aber auch in Anweisungen von Vorteil sein, wenn dadurch umständliche *if-else* Konstruktionen vermieden werden können.

```
Beispiel:
Steuerabzug = Einkommen *
              Steuerklasse>1 ? Abzug=23 : Abzug=25;
```

Die Definition von Makros und ihre Verwendung wird ausführlich im Kapitel über den Präprozessor behandelt. In den meisten Programmen spielt die bedingte Bewertung keine Rolle, da sie nur dazu geeignet ist eine einzelne Anweisung in Abhängigkeit von der Bewertung auszuführen. Zwar ist eine Blockklammerung im Makro denkbar, diese wäre aber extrem unübersichtlich. In Programmen wird daher fast ausschließlich die *if-else*-Anweisung verwendet.

7.4.2. DIE IF-ELSE ANWEISUNG

Die *if-else*-Anweisung (auch einfache Alternative genannt) kann für einfache Entscheidungen verwendet werden. In Abhängigkeit vom logischen Ergebnis des Ausdrucks in der Klammer nach dem *if* (meist ein Vergleich), wird entweder die nachstehende Anweisung (bei Ergebnis *TRUE*) oder die Anweisung nach dem zugehörigen *else* (bei Ergebnis *FALSE*) ausgeführt. Nach dem Schlüsselwort *if* bzw. *else* darf jeweils nur eine einzelne Anweisung stehen. Sollen in Abhängigkeit von der

Bewertung mehrere Anweisungen ausgeführt werden, so müssen diese Anweisungen zu einem Block zusammengefasst werden. Man beachte bei der *if-else*-Anweisung die eher unübliche Schreibweise, nach der Anweisung ein Semikolon zu setzen, wenn kein Block folgt:

```
Syntax:
if (Ausdruck) Anweisung_1;    // Kein Block
else Anweisung_2;

if (Ausdruck) { ... }         // Block nach if
else Anweisung_2;

if (Ausdruck) Anweisung_1;    // Block nach ELSE
else { ... }

if (Ausdruck) { ... }         // Mit Blöcken
else { ... }

if (Ausdruck) Anweisung_1;    // Ohne ELSE

if (Ausdruck) { ... }         // Ohne ELSE
```

```
//============================================================
// Programm VERZW2.CPP
//============================================================

#include <iostream>
#include <iomanip>

using namespace std;

int iSteuerklasseA    = 0;
int iSteuerklasseB    = 0;
int iAbzugProzentA    = 0;
int iAbzugProzentB    = 0;

void main (void)
{
   cout << "A-Steuerklasse eingeben : ";
   cin  >> iSteuerklasseA;

   //-----------------------------------------------------
   // if-Anweisung ohne Block
   //-----------------------------------------------------

   if (iSteuerklasseA == 4)
      iAbzugProzentA = 23;
   else
      iAbzugProzentA = 25;

   cout << "Abzüge = " << iAbzugProzentA << endl << endl;

   cout << "B-Steuerklasse eingeben : ";
```

```
   cin  >> iSteuerklasseB;

   //-----------------------------------------------------
   // if-Anweisung mit Block
   //-----------------------------------------------------

   if (iSteuerklasseB == 4)
   {
      iAbzugProzentB = 23;
      cout << "Prozentualer Abzug B beträgt 23 %" << endl;
   }
   else
   {
      iAbzugProzentB = 25;
      cout << "Oh je, satte 25 %% Abzug!" << endl;
   }
}
```

if-else-Anweisungen können beliebig verschachtelt werden. Allzu hohe Schachtelungstiefen (mehr als vier ineinander verschachtelte *if-else*-Anweisungen) sind jedoch unübersichtlich und sollten daher auf jeden Fall vermieden werden. Wie aus der Syntax ersichtlich, kann auf den *else*-Teil der Anweisung verzichtet werden, wenn er im Programm inhaltlich nicht benötigt wird. Obwohl syntaktisch nicht unbedingt notwendig, sollten bei der Verschachtelung von *if-else*-Anweisungen immer Blockklammern gesetzt werden, um die Lesbarkeit zu erhöhen. Das folgende Beispiel zeigt eine bedingte Verzweigung, die zwar syntaktisch korrekt ist, aber aufgrund der Schreibweise leicht missverstanden werden kann:

```
#include <iostream>

int iSteuerklasse = 0;
int iAbzugprozent = 0;

void main (void)
{
   if (iSteuerklasse > 1)
   if (iSteuerklasse == 4) iAbzugProzent = 23;
   else iAbzugProzent = 25;
}
```

Die Schreibweise lässt vermuten, dass die *else*-Anweisung zur Bedingung *iSteuerklasse > 1* gehört, was jedoch nicht der Fall ist. Die letzte offene Bedingung ist *iSteuerklasse == 4*, da die Anweisung später erfolgte und kein *else* besitzt. Zur Veranschaulichung der Zusammengehörigkeit des Programms ist die folgende Darstellungsform besser geeignet:

```
#include <iostream>
```

```
int iSteuerklasse = 0;
int iAbzugprozent = 0;

void main (void)
{
   if (iSteuerklasse > 1)
   {
      if (iSteuerklasse == 4)
         iAbzugProzent = 23;
      else
         iAbzugProzent = 25;
   }
}
```

Schon die Verwendung der geschweiften Blockklammern lassen jetzt keinerlei Zweifel mehr zu, dass die *else*-Anweisung zum zweiten *if* gehört. Möchte man hingegen im obigen Beispiel erreichen, dass das *else* zum ersten *if* gehört, so ist die Klammerung sogar unverzichtbar:

```
#include <iostream>

using namespace std;

int iSteuerklasse = 0;
int iAbzugprozent = 0;

void main (void)
{
   if (iSteuerklasse > 1)
   {
      if (iSteuerklasse == 4)
         iAbzugProzent = 23;
   }
   else
      iAbzugProzent = 25;
}
```

Durch die Blockbindung des zweiten *if* ist mit dem Schließen der Klammer (syntaktisches Ende des Blocks) auch immer das syntaktische Ende der eingeschlossenen Anweisungen verbunden. Eine häufige Fehlerquelle bei bedingten Verzweigungen ist die Verwechslung der Operatoren Zuweisung (=) und Vergleich (==). Die folgende Zeile mag auf den ersten Blick richtig erscheinen, kann aber niemals den *else*-Teil ausführen:

```
#include <iostream>

using namespace std;

int iSteuerklasse = 0;
int iAbzugprozent = 0;
```

```
void main (void)
{
   if (iSteuerklasse = 1)        // FEHLER !!!!!!!
   {
      iAbzugProzent = 23;
      cout << "Steuerklasse 1";
   }
   else
   {
      iAbzugProzent = 25;
      cout << "andere Steuerklasse";
   }
}
```

Die Bedingung nach *if* wird durch Seiteneffekte korrekt. Da die Zuweisung selbst ein Ergebnis hat (den zugewiesenen Wert Eins, der *TRUE* entspricht), ist die Bedingung immer wahr. Solcherlei Nebeneffekte sind in C legal. Wäre der zugewiesene Wert ungleich Null oder Eins, so wäre es vom erzeugten Maschinencode abhängig (und somit auch vom verwendeten Compiler), ob der *if* oder der *else*-Teil ausgeführt wird. Das Beispiel zeigt zudem die Notwendigkeit, zwei unterschiedliche Operatoren für Zuweisung und Vergleich zu verwenden. Da beide Möglichkeiten legal sind, hätte der Compiler im obigen Beispiel keine Möglichkeit zu entscheiden, ob der Vergleich oder die Zuweisung gemeint ist.

Um Fehler durch die Verwechslung von Vergleich und Zuweisung zu vermeiden, gibt es einen einfachen Trick., wenn auf einen konstanten Wert verglichen wird. Schreiben Sie bei Vergleichen (z.B. in der Bedingungsklammer einer if-Anweisung) immer die Konstante zuerst, also:

```
if (4 == A)  {...}
```

anstelle der zumeist verwendeten Form:

```
if (A == 4)  {...}
```

Liegt ein Schreibfehler vor und Sie haben statt des Vergleichsoperators die Zuweisung verwendet, meldet der Compiler einen Syntaxfehler, da der Konstanten natürlich kein neuer Wert zugeordnet werden kann. Dieser Trick funktioniert natürlich auch mit Werten, die mittels eines #define als symbolische Konstanten vereinbart wurden.

```
if (FALSE == BoolVariable)
   cout << "Variable enthält FALSE !! ";
```

7.4.3. DIE SWITCH ANWEISUNG

Die *switch*-Anweisung dient zur Vereinfachung der häufig vorkommenden, geschachtelten *if-else*-Abfragen. Bei der Eingabe eines Buchstaben für eine Entscheidung mit drei Möglichkeiten sieht eine *if-else*-Behandlung der Eingabe etwa so aus:

```
//============================================================
// Programm SWITCH1.CPP
//============================================================

#include <iostream>
#include <iomanip>

using namespace std;

char cEingabe = 'j';

void main (void)
{
   if (cEingabe == 'j')
   {
      cout << "Eingabe: Ja" << endl;
   }
   else
   {
      if (cEingabe == 'n')
      {
         cout << "Eingabe: Nein" << endl;
      }
      else
      {
         if (cEingabe =='h')
         {
            cout << "Eingabe: Hilfe" << endl;
         }
         else
         {
            cout << "Falsche Eingabe" << endl;
         }
      }
   }
}
```

Mit zunehmender Komplexität wird eine solche verschachtelte Auswertung immer schwieriger zu durchschauen - insbesondere, wenn die von der Bedingung abhängigen Anweisungen längere Blöcke bilden, als im Beispiel. Die *switch*-Anweisung löst genau dieses Problem auf eher tabellarische Weise:

```
//============================================================
// Programm SWITCH2.CPP
//============================================================
```

```
#include <iostream>
#include <iomanip>

using namespace std;

char cEingabe = 'j';

void main (void)
{
   switch (cEingabe)
   {
      case 'j' : cout << "Eingabe: Ja" << endl;
                 break;
      case 'n' : cout << "Eingabe: Nein" << endl;
                 break;
      case 'h' : cout << "Eingabe: Hilfe" << endl;
                 break;
      default  : cout << "Falsche Eingabe" << endl;
                 break;
   }
}
```

Die Anweisung kann allerdings, wie allgemein bei den sogenannten Auswahlanweisungen üblich, nur aus Werten der skalaren Typen auswählen (*char*, *int*, *short*, *long*, *enum*-Aufzählungen sowie den zugehörigen *unsigned* Varianten). Ein *switch* ist demnach weder mit komplexeren Datentypen (wie Strings oder Datensätze) noch mit Fließkommazahlen möglich.

```
Syntax:
   switch (SkalarVariable)
   {
      case Konstante1 : Anweisung1;
      case Konstante2 : Anweisung2;
                        break;
      default         : Anweisung3;
                        break;
   }
```

Bei der *switch*-Anweisung sind in C/C++ einige Besonderheiten zu beachten. So wird z.B. die Ausführung der Anweisungen nach einem *case* nicht automatisch abgebrochen, sondern der Programmierer muss den Abbruch mit der *break*-Anweisung ausdrücklich festlegen.

```
//============================================================
// Programm SWITCH3.CPP
//============================================================

#include <iostream>
#include <iomanip>
```

```
using namespace std;

char cZeichen = 'A';

void main (void)
{
   cout << "Zeichen eingeben : ";
   cin  >> cZeichen;

   switch (cZeichen)
   {
      case 'B'  : cout << "Buchstabe B"    << endl;
      case '\n' : cout << "Zeilenvorschub" << endl;
      case 65   : cout << "Buchstabe A"    << endl;
                  break;
      default   : cout << "Weder noch !! " << endl;
                  break;
   }
}
```

Auf das obige Beispiel bezogen bedeutet dies, dass bei *case 'B'* nicht nur der direkt folgende *printf*-Befehl ausgeführt wird, sondern auch die weiteren Anweisungen (bis zum Erreichen des nächsten *break* oder bis zum Blockende des *switch*). Mit der *break*-Anweisung wird die Abarbeitung des aktuellen Blocks künstlich beendet (der Abbruch mit *break* ist übrigens aus jedem Block heraus möglich).

Die Reihenfolge der *case*-Konstanten ist ebenso wenig vorgeschrieben, wie die Position der Sonderkonstanten *default*. Die Anweisungen nach *default* werden nur dann ausgeführt, wenn keine andere Auswahlanweisung zutrifft. Da die *default-Anweisung* alle sonstigen Fälle behandelt, darf es logischerweise nur einen *default*-Block pro *switch*-Anweisung geben. Die *break*-Anweisung nach *default* ist zwar in diesem Beispiel syntaktisch überflüssig, kann jedoch hilfreich und fehlervermeidend sein, wenn die Liste der Auswahlwerte später verlängert wird (das dann gegebenenfalls nötige *break* wird nicht versehentlich vergessen). Die *default*-Marke ist zudem optional, d.h. sie kann entfallen, wenn sie nicht gebraucht wird. Auch muss nach einer Marke nicht zwangsläufig eine Anweisung folgen, stattdessen kann man *case*-Anweisungen dazu benutzen um mehreren Konstanten den gleichen Anweisungsblock zuzuweisen:

```
//=============================================================
// Programm SWITCH4.CPP
//=============================================================

#include <iostream>
#include <iomanip>

using namespace std;
```

```
int iZahl = 0;

void main (void)
{
   cout << "Zahl eingeben : ";
   cin  >> iZahl;

   switch (iZahl)
   {
      case 17: cout << "Wert 17" << endl; // Anweisung 1
               break;
      case 0 :
      case 1 : cout << "Wert 0 oder 1" << endl; // Anw. 2
      case 45: cout << "Wert 45" << endl;       // Anw. 3
               break;
   }
}
```

Im obigen Beispiel wird *Anweisung 3* nur dann ausgeführt, wenn die Integervariable den Wert 45 hat. Ist *Zahl* Null oder Eins, so werden die *Anweisung 2* und *Anweisung 3* ausgeführt. Erst beim *break* wird die Ausführung der aufeinanderfolgenden Anweisungen unterbrochen. So wird beim Wert 17 nur *Anweisung 1* ausgewertet, da mit dem nachfolgenden *break* die Bearbeitung des Blockes abgebrochen wird.

Innerhalb einer *switch*-Anweisung können beliebige andere Instruktionen stehen. So können z.B. auch *switch*-Anweisungen verschachtelt werden. Wegen der mangelnden Übersichtlichkeit (die Tabellenstruktur geht optisch schnell verloren) ist davon jedoch abzuraten. Anstelle von verschachtelten *switch*-Anweisungen sollten besser Unterprogrammtechniken verwendet werden, um die Übersichtlichkeit zu erhalten.

7.5. FLUSSDIAGRAMM SYMBOL VERZWEIGUNG

Das Flussdiagrammsymbol für die bedingte Verzweigung ist eine Raute mit einem Eingang und zwei Ausgängen (s.u.). In die Raute wird die Bedingung so eingetragen, dass eine *TRUE* oder *FALSE*-Antwort gegeben werden kann - was den beiden Ausgängen entspricht. Die Ausgänge werden entsprechend mit dem Wort *TRUE* oder *FALSE* markiert (zuweilen finden sich auch andere Ausgangsbeschriftungen wie z.B. T / F, j / n, + / - oder 1 / 0).

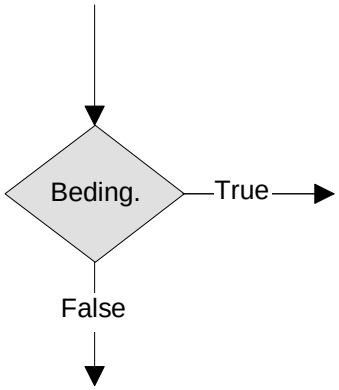


Abbildung 7-1 – Struktogrammsymbol Verzweigung

Das folgende Beispiel zeigt die Verwendung des Symbols in einem einfachen Flussdiagramm:

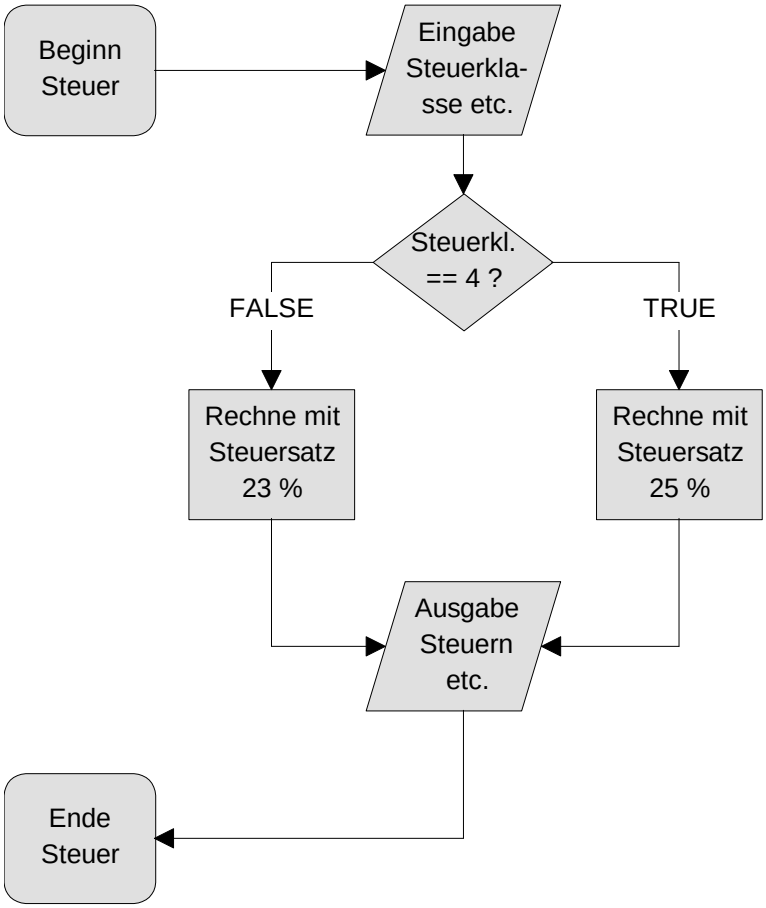


Abbildung 7-2 – Einfaches Flussdiagramm mit Verzweigung

7.6. AUFGABENTEIL

Bei allen Aufgaben ist ein Schwierigkeitsgrad angegeben. Der Schwierigkeitsgrad bezieht sich nicht allein auf die Aufgabenstellung sondern ggf. auch auf den Umfang der Ausgabe.

7.6.1. AUFGABE 1 (SCHWIERIG)

Wie El Commandante Caramba Garcia schnell bemerkt hat, ist er leider selbst einer der größten Steuerzahler seines Inselstaates. Daher verfügt er eine Kappungsgrenze für die Lohnsteuer von 50.000 Nachos. Für jeden Nacho oberhalb dieser Grenze muss keine Steuer gezahlt werden, wobei die Kopfsteuer vor Ermittlung der Kappung zu entrichten ist.

Auch die zahlreichen Proteste der Bauern nach Verkündung der Steuerreform sind am Regime nicht ganz spurlos vorbeigegangen. Unter dem Eindruck mehrerer Ladungen Dung vor dem Präsidentenpalast und unter Berücksichtigung der Tatsache, dass die von El Commandante erwogene gewaltsame Unterdrückung der Proteste dem Tourismus abträglich wäre, wurden die folgenden Änderungen eingeführt:

Verheiratete Steuerzahler zahlen nur noch 18% Lohnsteuer, unverheiratete hingegen 22%.

Um die Bevölkerung ein wenige besser kontrollieren zu können, hat El Commandante zudem einen Vertrag mit dem Oberhaupt (Seboss) der Bananisch Orthodoxen Kirche geschlossen Seboss Juanito Sombrero.

Dieser wird dem einfachen Volk die Botschaft zukommen lassen, dass es mit El Commandante den besten Anführer aller Zeiten bekommen hat, zum Ausgleich wird die Steuerbehörde von allen Steuerzahlern, die zugleich Kirchenmitglieder sind automatisch 3% Kirchensteuer erheben (zusätzlich zur Lohnsteuer). Gegen eine Gebühr von 70 Centimos versteht sich, die vom Kirchenmitglied zu zahlen ist. Außerdem hat der Seboss den Wunsch geäußert, dass die Kinderreichen Familien einen höheren Kinderfreibetrag bekommen. Daher werden nun das erste Kind mit 20 Nachos angerechnet, das zweite mit 25 Nachos und jedes weitere mit je 33.33 Nachos.

Geben Sie die zu zahlenden Steuern sowohl in Nachos als auch in Euro an. Ein Nacho hat einen Wert von 0,23561 Euro.

Testen Sie das Programm mit einigen typischen Durchschnittsverdiensten auf Porto Banana:

Bauer, Ehefrau, 2 Großeltern, 5 Kinder, Jahresverdienst: 970 Na
Professor, Ehefrau, 2 Großeltern, 2 Kinder, Jahresverdienst: 1623 Na
Parteifunktionär, Ehefrau, 1 Kind, Jahresverdienst: 47000 Na
El Commandante, unverheiratet, offizieller Jahresverdienst: 100.000 Na

7.6.2. AUFGABE 2 (EINFACH)

Schreiben Sie ein Programm, das den Verkaufspreis einer Ware in Porto Banana berechnet. Errechnen Sie den Verkaufspreis aus dem Einkaufspreis, den Fixkosten (70 % des Einkaufspreises), der Mehrwertsteuer (15 % des bis dahin errechneten Verkaufspreises), dem Gewinn (variabel in Prozent des Einkaufspreises).

Als Händler können sich dafür entscheiden, eine Ware als Sonderangebot anzubieten (Bildschirmabfrage). Bei einem Sonderangebot beträgt der Gewinn immer 10 % des Einkaufspreises und der Fixkostenanteil wird halbiert. Handelt es sich nicht um ein Sonderangebot, so gewähren Sie bei Barzahlung (Bildschirmabfrage) 10 % Rabatt. Für Sonderangebote kann kein Rabatt gewährt werden.

Geben Sie alle Endpreise (Sonderangebot und Normalpreis mit und ohne Rabatt), möglichst übersichtlich, auf dem Bildschirm aus.

Testen Sie das Programm mit folgenden Einkaufspreisen:

```
100.00  Nachos
 50.00  Nachos
 76.98  Nachos
```

8. WIEDERHOLUNGSANWEISUNGEN

Wie in fast allen Programmiersprachen, gibt es auch in C/C++ zwei Typen von Wiederholungsanweisungen (*for*- und *while*-Schleifen), die bedingten und unbedingten Schleifen. Als die einfachere Form wird von den meisten Programmierern die unbedingte *for*-Schleife betrachtet. Andere Compiler (z.B. PASCAL) kennen noch eine dritte Variante (die sogenannte *repeat-until*-Schleife), die in C/C++ aber durch eine Variation der *while*-Schleife gebildet wird.

8.1. FOR-SCHLEIFEN

In fast allen Sprachen zeichnet sich die *for*-Anweisung durch ein recht starres Schema aus, welches allerhöchstens die Manipulation der Schrittweite und die Wahl zwischen Auf- oder Abwärtszählen zulässt. Zumeist sind die Zählvariablen auch auf die skalaren Typen beschränkt (*char*, *int*, *short*, *long* und *enum*-Aufzählungstypen), so z.B. auch in PASCAL.

In C/C++ jedoch ist die unbedingte Schleife nur als besondere Form der bedingten Schleife realisiert, durch geschicktes Setzen der Parameter kann die unbedingte jederzeit in eine bedingte Schleife umgewandelt werden.

```
Syntax:
   for (Ausdruck1; Ausdruck2; Ausdruck3) Anweisung;

   for (Ausdruck1; Ausdruck2; Ausdruck3)
   {
      Anweisungsblock
   }
```

```
//============================================================
// Programm LOOP1.CPP
//============================================================

#include <iostream>
#include <iomanip>
#include <float.h>

using namespace std;

int i = 0;
int j = 3;
int k = 200;
double f = 1.0;

void main (void)
{
   for (i=0; i<10; i++)
      cout << "Durchgang „ << i << endl;
```

```
   for (i=j; i<k; i+=7)
   {
      cout << endl;
      cout << "Wert i = " << i << endl;
   }

   for (f=17.5; f<100.0; f+=0.5)
   {
      cout << "Wert von f " << f << endl;
   }
}
```

Die Bedeutung der drei Ausdrücke ist nur allgemein festgelegt. So ist *Ausdruck1* dazu gedacht die Schleifenvariable zu initialisieren. Der Ausdruck wird nur einmal, vor dem ersten Schleifendurchlauf ausgewertet, dient fast immer dazu, der Zählvariablen (hier *i*) einen Startwert zuzuweisen. *Ausdruck2* definiert die Abbruchbedingung - woran bereits zu erkennen ist, dass es sich bei der Anweisung *for* nicht um eine „echte“ unbedingte Schleife handelt. So ist im zweiten Beispiel der Abbruch von einer anderen als der initialisierten Variablen abhängig. Der typische Charakter einer unbedingten Schleife, eine feste Anzahl von Schleifendurchgängen zu besitzen (die spätestens zu Beginn der Schleife feststeht), wird in C/C++ somit völlig aufgehoben. Entsprechend seiner Bedeutung wird der *Ausdruck2* vor jedem Schleifendurchgang ausgewertet und die Wiederholung wird nur fortgesetzt, solange der Ausdruck *TRUE* ist.

Der dritte Ausdruck hingegen (*Ausdruck3*) wird nach jedem Schleifendurchgang ausgewertet. Hier befindet sich üblicherweise die Schleifenfortschaltung, also das herauf- oder herab zählen der Schleifenvariablen.

Wie bei den bedingten Verzweigungen darf auch der *for*-Schleife nur eine Anweisung folgen. Sollen innerhalb der Wiederholungsanweisung mehrere Befehle ausgeführt werden, muss wieder ein Block verwendet werden. Schleifen sind natürlich ebenfalls, wie die Verzweigungen, beliebig schachtel- und kombinierbar.

Die *for*-Schleife in C/C++ weist einige Besonderheiten auf. Wie bereits betont, handelt es sich nur um eine Sonderform der bedingten Schleife und die *for*-Anweisung wird vom Compiler auch in eine solche umgewandelt.

Es ist, wie in den nachstehenden Beispielen zu erkennen, aber auch erlaubt, dass ein Teil der Ausdrücke im Schleifenkopf der *for*-Anweisung entfallen können. Allerdings muss das jeweilige Semikolon erhalten bleiben (Leeranweisung), damit der Compiler ermitteln kann, welche der

Ausdrücke entfallen und welche besetzt sind. Die erste Beispielanweisung enthält z.B. keinerlei Initialisierung. Der Extremfall (alle Parameter fehlen) ist logischerweise eine Endlosschleife.

```
#include <iostream>
#include <iomanip>

using namespace std;

int i = 1;

void main (void)
{
   for (;i<1000; i*=2)
   {
      cout << "Wenn i<1000, dann i=i*2: " << i << endl;
   }

   for (;;)
   {
      cout << "Auf immer und ewig..." << endl;
   }
}
```

Eine solche Schleife kann dann nur noch durch Bedingungen beendet werden, die einen Abbruch des Blockes durch *break* oder *return* (in Unterprogrammen) herbeiführen. Für jeden der drei Ausdrücke können aber auch mehrere Anweisungen angegeben werden, diese müssen dann mit einem Komma getrennt werden. Ein nützliches Beispiel ist beispielsweise die Fortschaltung von zwei Variablen am Ende eines Blockes:

```
//============================================================
// Programm LOOP2.CPP
//============================================================

#include <iostream>
#include <iomanip>

using namespace std;

int i = 0;
int j = 0;

void main (void)
{
   for (i=0, j=1000; i<=j; i++, j--)
   {
      cout << "Annäherung i=" << i << " und j=" << j << endl;
   }
}
```

Syntaktisch ebenfalls korrekt, wenn auch sehr unübersichtlich, ist es, wenn die Fortschaltung in eine Anweisung eingebaut werden kann. Diese Anweisung kann in den Schleifenkopf hineingezogen werden, ein Schleifenrumpf (Anweisungsteil) entfällt dann häufig ganz.

```
#include <iostream>
#include <iomanip>

using namespace std;

int i = 0;

void main (void)
{
   for (i=0; i<=10; cout << "Zahl : " << i++ << endl));
}
```

Da die *for*-Schleife in Wirklichkeit nur eine Sonderform der *while*-Schleife (s.u.) ist, können auch *float*-Variablen zur Schleifengestaltung herangezogen werden:

```
//============================================================
// Programm LOOP3.CPP
//============================================================

#include <iostream>
#include <iomanip>

using namespace std;

double f = 0.0;

void main (void)
{
   for (f=0.0; f<=2.75; f+=0.25)
      cout << "f ist gleich " << f << endl;
}
```

Die Gestaltungsmöglichkeiten mit der *for*-Anweisung sind vielfältig, aber genau deshalb auch sehr fehleranfällig. Einige Programmiersprachen verbieten deshalb z.B. die Manipulation der Zählvariablen innerhalb des Schleifenrumpfs völlig (d.h. dass die Schleifenvariable im Schleifenrumpf nicht Ziel einer Zuweisung sein darf). Anders hingegen C/C++, welches geradezu dazu einlädt solche Manipulationen zur Effizienzsteigerung vorzunehmen. Insbesondere ist beim zweiten Ausdruck (Abbruchbedingung) darauf zu achten, dass die Operatoren „=“ und „==“ nicht verwechselt werden, da man sonst sehr leicht in eine Endlosschleife geraten kann.

Unter C-Programmierern gilt es als guter Stil, Zählvariablen der skalaren Typen (char, int, short und long) mit kurzen Variablennamen zu belegen. Insbesondere die Namen i, j, k, l, m und n werden dazu gern benutzt. Die Benennung von Zählern mit diesen Namen stammt aus der Mathematik.

Um der Lesbarkeit willen ist jedoch die Empfehlung auszusprechen, von extremen Optimierungen, insbesondere vom Hineinziehen der Anweisung in die Ausdrücke des Schleifenkopfes abzusehen.

8.2. WHILE-SCHLEIFEN

Die *while*-Schleife ist die eigentliche Schleife in C/C++. Sie kommt in zwei Ausführungen vor. Bei der ersten Form erfolgt die Bedingungsprüfung am Anfang, bei der zweiten am Ende der Schleife.

```
Syntax:
   while (expr) Anweisung;

   while (expr) {Anweisungen}
   do Anweisung while (Ausdruck);

   do {Anweisungen} while (Ausdruck);
```

```
//============================================================
// Programm WHILE1.CPP
// ============================================================

#include <iostream>
#include <iomanip>

using namespace std;

int i = 1;

void main (void)
{
   while (i<1000)
   {
      cout << "i ist kleiner 1000 ! " << i << endl;
      i *= 2;
   }

   i = 1;
   do
   {
      cout << "i ist kleiner 1000 ! " << i << endl;
      i *= 2;
   }
   while (i<1000);
}
```

Wie bei der *for*-Schleife darf der *while*- oder *do-while*-Anweisung nur ein einzelner Befehl folgen. Sollen innerhalb der Wiederholungsanweisung mehrere Instruktionen ausgeführt werden, so müssen diese (wie gehabt) in einem Block zusammengefasst werden. *while*-Schleifen sind natürlich ebenfalls - wie die Verzweigungen und die *for*-Schleifen - beliebig mit anderen Befehlen schachtel- und kombinierbar.

Die *while*-Anweisung wird manchmal auch kopfgesteuerte Schleife genannt, da die Bedingungsabfrage vor dem Schleifendurchlauf erfolgt. Entsprechend wird der *do-while*-Befehl gelegentlich als fußgesteuerte Schleife bezeichnet. Der wichtigste Unterschied zwischen *while* und *do-while* ist aber die Tatsache, dass im Gegensatz zum *do-while* der Anweisungsteil der *while*-Schleife unter Umständen gar nicht ausgeführt wird - genau dann, wenn die Schleifenbedingung von vornherein falsch (*FALSE*) ist. Im Gegenteil dazu wird der Anweisungsteil der *do-while*-Schleife auf jeden Fall mindestens einmal durchgeführt.

Da die *for*-Schleife, wie bereits mehrfach betont, nur ein Sonderfall der *while*-Anweisung ist, kann das *for* mühelos durch ein *while* simuliert werden. Die folgenden beiden Beispiele haben die gleiche Wirkung:

```
for (Ausdruck1; Ausdruck2; Ausdruck3) {Anweisungen};

Ausdruck1;
while (Ausdruck2)
{
   Anweisungen;
   Ausdruck3;
}
```

Die Gestaltungsmöglichkeiten der *while*-Anweisung sind demnach mit denen der *for*-Schleife identisch. Dennoch wird von vielen Programmierern die *for*-Anweisung bevorzugt, da sie alle wichtigen Daten der Wiederholungsanweisung am Anfang der Schleife (in der Klammer) zusammenfasst. Die potentiellen Fehlerquellen bei *while* und *do-while* sind identisch mit denen der *for*-Schleife – aber meist schwieriger zu finden, da die drei definierenden Ausdrücke der Schleife weit verstreut liegen können.

8.3. DIE ANWEISUNGEN BREAK UND CONTINUE

Mit der *break*-Anweisung kann der gerade bearbeitete *switch*- oder *while*-Block künstlich abgebrochen werden. Die restlichen Anweisungen bis zum Ende des Blocks werden dann ignoriert. Zum Abbrechen eines Unter- oder Hauptprogramms dienen die *return* (s.u.) bzw. *exit*-Anweisung, die später behandelt werden (siehe Kapitel über Funktionen).

```
Syntax:
```

```
   break;
   continue;
```

```
//=============================================================
// Programm BREAK1.CPP
//=============================================================

#include <iostream>
#include <iomanip>
#include <conio.h>

using namespace std;

int i = 0;
int j = 0;
char Eingabe = 'A';

void main (void)
{
   //-------------------------------------------------
   // Endlosschleife, mit BREAK abbrechbar
   //-------------------------------------------------
   for (i=0; ;i++)
   {
      cout << "Durchlauf" << i << ", Ende mit E" << endl;
      cin  >> Eingabe;
      if ('E' == Eingabe)
         break;
   }

   //-------------------------------------------------
   // Neudurchlauf, mit CONTINUE vorzeitig erzwungen
   //-------------------------------------------------
   for (i=1, j=0; j==0; /*NIX*/ )
   {
      cout << "Durchlauf " << j << endl;
      cout << "Bedingung " << i << endl;

      cout << "i verdoppeln (j/n)";
      Eingabe = getch ();

      if ('j' == Eingabe)
      {
         i *= 2;
         continue;
      }
      else
         j++;
   }
}
```

Die *continue*-Anweisung arbeitet ähnlich. Mit Hilfe des *continue* kann die nächste Wiederholung einer Schleife künstlich erzwungen werden,

wobei alle Anweisungen zwischen dem continue und dem Blockende ignoriert werden. Die *break*-Anweisung wird sehr häufig verwendet, insbesondere in der *switch*-Anweisung. Die *continue*-Anweisung hingegen findet man relativ selten.

8.4. DIE ÜBERFLÜSSIGE ANWEISUNG GOTO

Die wohl umstrittenste Anweisung unter den Verfechtern der prozeduralen Programmierung ist das *goto*. Es ist zwar mathematisch bewiesen, dass jedes Problem auch ohne *goto* lösbar ist - nur leider sind diese Lösungen zum Teil wesentlich aufwendiger oder umständlicher. Daher ist die unbeliebte Anweisung auch in C/C++ zu finden. Das *goto* wird hauptsächlich dann verwendet, wenn es gilt, aus mehreren ineinander geschachtelten Blöcken gleichzeitig herauszuspringen, wie im folgenden Beispiel:

```
for (i=0; i<n; i++)
   for (j=0; j<m; j++)
      if (i*j > 1000) goto ERROR;
         ...
ERROR:
   ...
```

Da eventuell auf dem Heap angelegte Werte nicht wieder korrekt freigegeben werden, sollte man mit der *goto*-Anweisung nicht nur so sparsam wie möglich verfahren, sondern sie eigentlich nur dann einsetzen, wenn ein aufgetretener Fehler es ohnehin notwendig macht, den Programmlauf zu beenden. Mit Sicherheit würde ähnliches mit dem Stack geschehen, wenn man Unterprogramme mit einem *goto* verlassen könnte, was genau aus diesem Grund in C/C++ verboten ist.

Spätestens in C++ sollte auf die *goto*-Anweisung gänzlich verzichtet werden, da es für das einzig akzeptierte Einsatzgebiet (Heraussprungen aus tiefen Blockschachtelungen im Fehlerfall) ein eigenes Konstrukt gibt, die *try-catch*-Anweisung (nähere Informationen dazu sind dem entsprechenden Kapitel zu entnehmen).

8.5. FLUSSDIAGRAMM SYMBOL SCHLEIFE

Ein eigenes Flussdiagrammsymbol für Schleifen gibt es nicht, stattdessen werden Kombinationen aus Anweisungen (Rechteck) und bedingter Verzweigung (Raute) verwendet. Die Rechtecke enthalten die Initialisierung und die Fortschaltung, die Raute enthält die Abbruchbedingung. Kennzeichnend für eine Schleife ist lediglich der Ausgangspfeil, der vom Anweisungsteil zur Bedingungsabfrage zurückführt.

Der Unterschied in der Darstellung zwischen der while- und der so-while-Schleife besteht allein in der Reihenfolge der Teilelemente, welche die Schleife bilden.

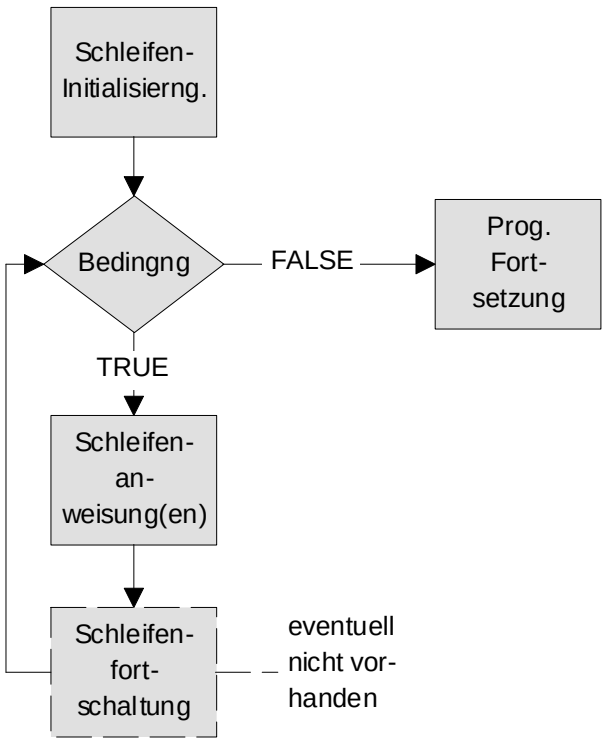


Abbildung 8-1 – Struktogrammsymbol while-Schleife

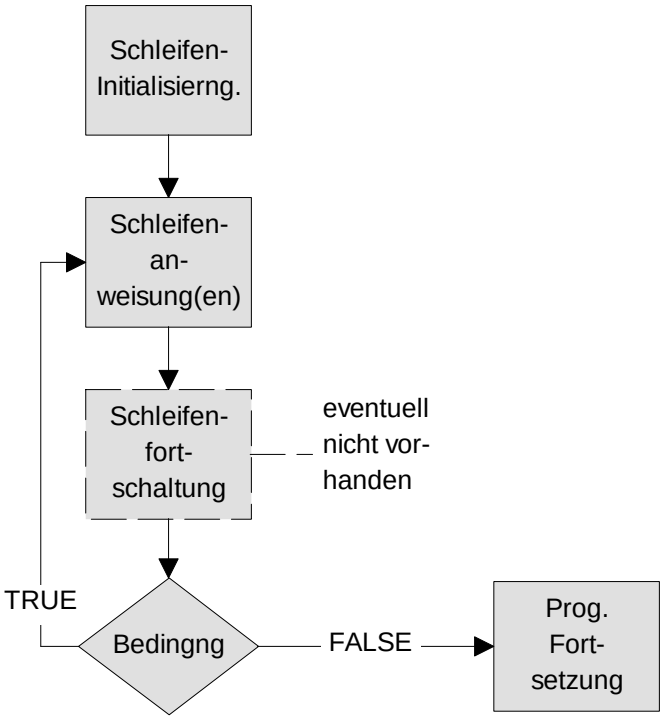


Abbildung 8-2 – Struktogrammsymbol do-while-Schleife

8.6. AUFGABENTEIL

Bei allen Aufgaben ist ein Schwierigkeitsgrad angegeben. Der Schwierigkeitsgrad bezieht sich nicht allein auf die Aufgabenstellung sondern ggf. auch auf den Umfang der Ausgabe.

8.6.1. AUFGABE 1 (EINFACH)

Um die Kundenfreundlichkeit der eiligst errichteten Beton-Touristenburgen zu messen und zu steigern hat El Commandante ein ebenso einfaches wie effizientes System entwickelt.

Jeden Monat werden die Einkünfte der Hotelanlagen an den Zentralen Rat für Tourismusentwicklung übermittelt. Der Manager der Bettenburg mit den höchsten Einnahmen erhält automatisch den großen Tourismus-Verdienstorden sowie den Titel eines „Helden um die ehrenvolle Fortentwicklung aller Besucherströme in Porto Banana“ (HudeFaBiPB).

Dem Manager mit den geringsten Einnahmen droht ein mehrwöchiger Kuraufenthalt im „Erziehungslager zur Förderung von Leistung und Gehorsam“, dem berüchtigten Campo Torturlini.

Schreiben Sie ein Programm, welches die Erträge der zehn größten Hotelanlagen einliest und folgende Werte ausgibt:

- die größte erzielte Einnahme
- die kleinste erzielte Einnahme
- den Durchschnitt der erzielten Einnahmen

Die Namen der Hotelanlagen sollen zunächst nicht miterfasst werden.

Beachten Sie bitte, dass es zur Lösung des Problems nicht notwendig ist, die zehn Einnahmen zwischen zu speichern.

8.6.2. AUFGABE 2 (MITTEL)

El Commandante legt sehr viel Wert auf den zügigen Ausbau der touristischen Infrastruktur. Weitere Hotelkomplexe für reiche Reisende mit Drang zum All-Inclusive Angebot werden unter strenger Bewachung durch die örtlichen Polizei- und Militärbehörden errichtet.

Jeden Monat werden die Einkünfte der Hotelanlagen an den Zentralen Rat für Tourismusentwicklung übermittelt. Der Manager der Bettenburg mit den höchsten Einnahmen erhält automatisch den großen Tourismus-Verdienstorden sowie den Titel eines „Helden um die ehrenvolle Fortentwicklung aller Besucherströme in Porto Banana“ (HudeFaBiPB).

Dem Manager mit den geringsten Einnahmen droht ein mehrwöchiger Kuraufenthalt im „Erziehungslager zur Förderung von Leistung und Gehorsam“, dem berüchtigten Campo Torturlini.

Schreiben Sie ein Programm, welches die Erträge einer beliebigen Anzahl von Hotelanlagen (mindestens jedoch einer) einliest und folgende Werte ausgibt:

- die größte erzielte Einnahme
- die kleinste erzielte Einnahme
- den Durchschnitt der erzielten Einnahmen

Die Namen der Hotelanlagen sollen zunächst nicht miterfasst werden.

Beachten Sie bitte, dass es zur Lösung des Problems nicht notwendig ist, die Einnahmen zwischen zu speichern.

9. FUNKTIONEN

Das Bilden von Funktionen ist die einzige Methode, mit der in C/C++ Unterprogramme realisiert werden können. Im Gegensatz zu vielen anderen Programmiersprachen (z.B. PASCAL und MODULA) kennt C/C++ keinen Unterschied zwischen Prozeduren und Funktionen. Da es dem Programmierer jedoch völlig frei steht zu entscheiden, ob eine Funktion ein Ergebnis liefert oder nicht (bzw. ob dieses Ergebnis schlicht ignoriert wird), ist es sehr einfach Funktionen zu schreiben, welche die gleiche Aufgabe erfüllen wie eine Prozedur. Meist sind solche Funktionen durch den Ergebnistyp *void* gekennzeichnet. Da es in C/C++ nur Funktionen als Unterprogrammtyp gibt, ist die Verwendung von Funktionen nicht an die gleichen Regeln gebunden wie in Programmiersprachen, die diese Unterprogrammtypen unterscheiden (z.B. dass Prozeduren nicht in Ausdrücken und Funktionen nur in Ausdrücken stehen dürfen). Daraus ergeben sich neben zusätzlichen Programmfreiheiten auch eine Menge potentieller Fehlerquellen. Der grundsätzliche Funktionsaufbau hat folgendes Aussehen:

```
Syntax:
   Ergebnistyp Funktionsname (Parameterdeklaration)
   {
      lokale Variablen;
      Anweisungen;
      return (...);
   }
```

Der Ergebnistyp (Funktionstyp) bestimmt, von welchem Datentyp das ermittelte Resultat der Funktion ist. Als Ergebnistypen sind nur die einfachen Datentypen, zusammengesetzte Datentypen (*struct*, siehe dazu Kapitel über Strukturen) und Zeiger darauf, bzw. Pointer auf komplexe Datentypen erlaubt (die Pointer und ihre Verwendung in Funktionen werden in den folgenden Abschnitten über Felder und Strukturen ausführlich behandelt). Nach dem Ergebnistyp steht der Funktionsname. Für Funktionsnamen gelten die gleichen Regeln wie für Variablennamen. Die Eindeutigkeit muss gewahrt bleiben, d.h. keine zwei Variablen und Funktionen dürfen den gleichen Namen tragen.

Mit der Parameterliste werden einer Funktion zu bearbeitende Daten übergeben, die Parameterdeklaration wird weiter unten behandelt. Der Ergebnistyp sowie die Parameter (und damit auch die Parameterdeklaration) sind optional. Innerhalb der Funktion müssen die lokalen Variablen, sofern überhaupt notwendig, zuerst aufgeführt werden. Syntaktisch gesehen könnten die Variablen zwar auch erst kurz vor ihrer Benutzung deklariert werden, der Übersichtlichkeit halber gehören sie jedoch an den Anfang der Funktion (Das ursprüngliche K&R-

C lässt eine Deklaration nur am Anfang zu). Der minimale Aufbau einer Funktion hat also folgende Struktur:

```
Syntax:
   Funktionsname ()
   {
   }
```

Hierbei wird angenommen, dass es sich um eine Funktion vom Typ Integer handelt (Voreinstellung), diese Annahme wird bei allen Funktionen ohne ausdrückliche Angabe des Ergebnistyps gemacht.

Die Angabe des Ergebnistyps wird gelegentlich vernachlässigt, wodurch die nicht deklarierten Ergebnistypen automatisch zu Integer werden. Dies hat jedoch weitreichende Konsequenzen. So kann der Compiler nicht überprüfen, ob solche Funktionen, die eigentlich vom Typ void sind, versehentlich in einer Berechnung auftauchen und daher Zufallswerte einbringen. Sie sollten daher die Disziplin üben, alle void-Ergebnistypen auch anzugeben.

Eine weitere Einschränkung ist die Tatsache, dass Funktionen nicht verschachtelt werden können, anders als in PASCAL ist es also nicht möglich, lokale Funktionen zu bilden (Funktionen die innerhalb einer Funktion definiert werden).

9.1. ALLGEMEINER FUNKTIONSAUFBAU

9.1.1. DIE RETURN-ANWEISUNG

Von jeder Funktion, die nicht den Ergebnistyp *void* hat, wird angenommen, dass sie ein Ergebnis liefert. Das Ergebnis wird durch das Schlüsselwort *return* festgelegt.

```
int Beispielname (void)
{
   ...
   return (17);
}
```

Nach dem Schlüsselwort *return* darf, ggf. in Klammern, ein beliebiger Ausdruck stehen, der ein Ergebnis vom Typ der Funktion erbringen muss:

```
int Beispielname (void)
{
   ...
   return (i+7-x);
}
```

Anschließend wird die Funktion beendet und das Ergebnis an das aufrufende Programm übergeben, wo es in einer Berechnung verwendet, zugewiesen oder ignoriert werden kann. Das bedeutet - auch wenn eine Funktion einen bestimmten Ergebnistyp zugewiesen bekommen hat - so besteht keine Verpflichtung mittels der *return*-Anweisung auch wirklich einen Wert an die aufrufende Funktion zurückzugeben (man sollte es jedoch tun):

```
Integervariable = Beispielname () + Zahl2;
```

Wird lediglich *return* angegeben, also weder Variable noch Konstante als Ergebnis genannt, wird nur die Funktion beendet. Diese Vorgehensweise wird z.B. in *void*-Funktionen verwendet, um das Unterprogramm vorzeitig zu verlassen:

```
void Beispiel2 (void)
{
   ...
   if (TRUE == Ende) return;
   else ...
   ...
}
```

Der *return*-Wert einer Funktion kann in Unterprogrammen vom Typ *void* auch ganz entfallen, die Funktion endet dann mit der abschließenden, geschweiften Klammer des Funktionsblocks. Werden Ergebnisse eines anderen Typs zurückgegeben, so findet zuvor eine implizite Typkonvertierung statt.

9.1.2. LOKALE VARIABLEN

Die innerhalb einer Funktion deklarierten Variablen- und Parameternamen sind lokal, d.h. auch nur innerhalb der Funktionsblocks zugänglich. Im Normalfall wird der nötige Speicherplatz für diese Variablen bei Aufruf der Funktion automatisch belegt (Allokation). Mit dem Ende der Funktion wird dieser Speicherplatz wieder freigegeben (Deallokation). Die Deklaration lokaler Variablen erfolgt analog zur Deklaration der globalen Variablen am Anfang des Programms. Letztere sind überall im Programm (nicht aber in anderen Modulen) zugreifbar, sofern sie nicht überdeckt werden (s.u.).

Verwenden Sie anstelle von globalen Variablen möglichst lokale, da diese von einem guten, optimierenden Compiler häufig in Registern untergebracht werden können, was eine Beschleunigung im Zugriff auf die Variable und ihren Inhalt bewirkt.

Die Deklaration lokaler Variablen erfolgt nach der öffnenden, geschweiften Klammer der Funktion. Da *main* eine Funktion wie jede andere ist (mit der Ausnahme, dass mit *main* die Abarbeitung gestartet wird), können auch in *main* lokale Variablen deklariert werden. Sollen lokale Variablenwerte über die Beendigung des Funktionslaufes hinaus erhalten werden, so kann dies durch Angabe des Schlüsselwortes *static* geschehen (siehe dazu das Kapitel über Speicherklassen). Auf den Wert, der zuletzt in der Variable enthalten war, kann dann beim nächsten Funktionsaufruf wieder zurückgegriffen werden:

```
//============================================================
// Programm STATIC.CPP
//============================================================

#include <iostream>
#include <iomanip>

using namespace std;

int i = 0;

void aufrufanzahl (void)
{
   static long anz = 0;

   cout << "Dies ist der " << ++anz << ".te Aufruf" << endl;
}

void main (void)
{
   for (i = 0; i < 10; i++)
   {
      aufrufanzahl ();
   }
}
```

Für die Gültigkeit von Namen in Programmen gelten in C/C++, wie bereits beschrieben, die gleichen Regeln wie in anderen Programmiersprachen. Auf jeder Ebene (global, innerhalb einer Funktion etc.) muss jeder Name einmalig sein. Allerdings kann in einer anderen Ebene, z.B. innerhalb einer Funktion, ein Name verwendet werden, der schon in der aufrufenden Funktion oder global existiert. Diese Namen werden dann überdeckt (Variablenüberdeckung), d.h. dass ihre Werte erhalten bleiben, es kann jedoch nur auf die lokale Variable dieses Namens zugegriffen werden. Typische Beispiele sind die oft verwendeten Namen für Zählvariablen von Schleifen.

9.1.3. FUNKTIONSTYPEN

Der Typ einer Funktion ist definiert als der Typ ihres Ergebniswertes. Berechnet eine Funktion eine Zahl vom Typ *long*, so ist *long* auch der

Typ der Funktion. Erlaubte Typen sind alle einfachen Datentypen wie z.B. *char*, *short*, *int*, *long*, *float* und *double*, Aufzählungstypen (*enum*) die Strukturen (*struct*) sowie Pointer auf diese Typen. Da Felder (Arrays, in C oft auch Vektoren genannt) immer über einen Zeiger auf ihr erstes Element angesprochen werden, ergibt sich hier keine Typkollision wenn versucht wird als Ergebnistyp ein Feld zu erhalten. Das Resultat ist dann der Zeiger auf das Ergebnisfeld:

```
double  abc () {...}
int    *def () {...}
void    ghi () {...}
```

Das erste Beispiel definiert eine Funktion mit dem Ergebnistyp *double*, die zweite hingegen liefert einen Zeiger auf einen Integerwert als Resultat. Die dritte aufgeführte Funktion hat keinen Rückgabetyp und entspricht einer Prozedur.

9.1.4. PARAMETERÜBERGABE

In den bisherigen Funktionsbeispielen stand hinter dem Namen jeweils eine Klammer mit dem Schlüsselwort *void*. Dieses *void* hat die gleiche Bedeutung wie beim Ergebnistyp, es ist die explizite Bekanntgabe, dass die Funktion keine Übergabewerte verarbeiten soll. Sollen hingegen Übergabewerte erlaubt sein, so sind für diese Werte symbolische Namen (Parameternamen) und die Datentypen (Parametertyp) zu vereinbaren. Die tatsächlich übergebenen Datentypen der Parameter können von den formal vereinbarten Parametern abweichen. In diesem Fall werden die aktuellen Parameter-Datentypen den formalen Datentypen durch implizite Typumwandlung angepasst. Dies geschieht nach den gleichen Regeln, die auch angewendet werden, wenn eine Zuweisung zwischen unterschiedlichen Datentypen durchgeführt wird. Wird ein größerer Datentyp in einen kleineren Datentyp umgewandelt, wird lediglich eine Warnung generiert. Diese sollte man jedoch sehr ernst nehmen und das Warning gegebenenfalls durch ein Typcasting (siehe entsprechendes Kapitel) ausdrücklich unterdrücken.

```
//============================================================
// Programm FACUL.CPP
//============================================================

#include <iostream>
#include <iomanip>

using namespace std;

long i = 0;

long facul (long x)
{
```

```
   long ergebnis = x;

   for ( ; x>2; ergebnis *= (--x));
   return (ergebnis);
}

void main (void)
{
   i = facul (7);
   cout << "Ergebnis : " << i << endl;
}
```

Das Beispiel zeigt eine erste, arbeitsfähige Funktion zur Berechnung der Fakultät einer Zahl. Die Funktion heißt *facul* und kann von anderen Programmteilen (z.B. *main*) aufgerufen werden. Die Funktion liefert ein Ergebnis vom Typ *signed long* zurück. Um die Fakultät einer Zahl zu berechnen, muss man der allgemeinen Rechenvorschrift in *facul* natürlich mitteilen, welchen Wert diese besitzt. Die dazu notwendigen Angaben bilden die Parameterliste, die in den runden Klammern nach dem Funktionsnamen steht. In diesem Fall ist es eine Zahl vom Typ long. Der Parametername *x* ist ein Platzhalter für eine beliebige Zahl oder Variable:

```
LongVar1 = 17 + facul (Zahl1); // Variable  übergeben
LongVar2 = facul (7);          // Konstante übergeben
LongVar2 = facul (7+Zahl2);    // Ausdruck  übergeben
facul (Zahl3);                 // ignoriere Ergebnis
```

Beim Funktionsaufruf von *facul* verlässt der Computer den gerade bearbeiteten Programmteil, merkt sich aber die Stelle, um dorthin zurückzukehren. Der Ausdruck oder Wert, dessen Fakultät berechnet werden soll, wird zunächst ermittelt (der Wert einer Variablen wird aus dem Speicher geholt, bzw. die Bewertung eines Ausdrucks abgeschlossen) und anschließend auf eine neu angelegte Variable namens *x* kopiert. Innerhalb der Funktion *facul* wird außerdem eine Hilfsvariable mit dem Namen *ergebnis* angelegt (Allokation), auf der die Berechnung der Fakultät erfolgt. Am Ende der Funktion wird das Ergebnis der Berechnung an das aufrufende Programm zurückgegeben, die Variablen *ergebnis* und *x* werden aus dem Speicher entfernt (Deallokation) und die Arbeit wird an der Unterbrechungsstelle fortgesetzt. Sowohl der Parameter *x* als auch die Variable *ergebnis* sind nur temporär vorhanden, und zwar exakt solange, wie sich der Computer in der Funktion *facul* aufhält. Solche temporären Variablen heißen lokal (s.u.). Durch die Tatsache, dass *return*-Werte entfallen dürfen sind Konstruktionen, die einmal einen Wert zurückgeben und in einem anderen Fall nicht, formal gesehen legal. Solche Funktionen sind aber recht problematisch, da solche Ergebnisse nur schwer (wenn nicht gar

unmöglich) zu interpretieren sind und sehr leicht zu Fehlern führen, die kaum nachvollziehbar sind:

```
int abc (int x)
{
   if (x < 0) return (1);   // so bitte nicht !!!!
}
```

Auch Funktionen, die mit dem Schlüsselwort *void* (als Funktionen ohne Rückgabewert) gekennzeichnet wurden, können auf eine Variable zugewiesen werden. Das Ergebnis ist dann der Wert, der zufällig im Ergebnisregister steht – Sinn macht das natürlich nicht, ein guter Compiler unterbindet dies auch.

Die Parametertypen und die Parameteranzahl der aktuellen Parameter müssen mit den Typen und der Anzahl der formalen Parameter übereinstimmen. Hier ist der Programmierer in die Verantwortung genommen, einige ältere C-Compiler überprüfen dieses weder zur Übersetzungs- noch zur Laufzeit. Ein weiteres Problem (und eine Quelle sehr schwer zu entdeckender Fehler) ergibt sich aus der Tatsache, dass die tatsächliche Reihenfolge der Parameterübergabe nicht immer definiert ist, sondern von einigen Compilern bei der Optimierung geändert werden kann. Zusammen mit der Möglichkeit Ausdrücke im Funktionsaufruf anzugeben, können sich Zweideutigkeiten ergeben, wie im folgenden Beispiel:

```
a = 4;
funkt_1 (a=2, a+3);   // Falsch, so bitte nicht !!
```

Abhängig davon, ob die Zuweisung an die Variable *a* vor oder nach der Auswertung von *a+3* erfolgt, werden die Werte *funkt_1 (2,5)* oder *funkt_1 (2,7)* übergeben. In solchen Fällen ist es günstiger, die Parameterausdrücke vorher auszuwerten und in lokalen Variablen zwischen zu speichern.

9.1.5. ÜBERGABEVARIANTEN

Die Übergabe der Parameter erfolgt bei einfachen Datentypen „Call by Value“, d.h. der Inhalt des Parameters wird auf eine neu im Speicher angelegte Variable kopiert und mit dieser Kopie wird gearbeitet. Bei Feldern (Arrays) hingegen erfolgt die Übergabe implizit „Call by Pointer“, d.h. der Inhalt wird nicht kopiert, sondern lediglich die Adresse auf den Speicherbereich übergeben. Daraus folgt, dass, anders als bei der Übergabe „Call by Value“, jede Veränderung des Objektes in der Funktion auch für die aufrufende Funktion Gültigkeit besitzt (s.u.). Soll auch für

einen einfachen Datentyp eine Übergabe über Pointer erfolgen, so muss dieses explizit über einen Pointer in der Parameterliste kenntlich gemacht werden:

```
void tauschen (int *x, int *y)
{
   int z;

   z  = *x;
   *x = *y;
   *y =  z;
}
```

Bei der Verwendung eines C++ Compilers gibt es noch eine dritte Form des Aufrufes, den „Call by Reference“. Bei dieser Variante erfolgt der Aufruf automatisch über einen Zeiger (wie bei „Call by Pointer“), anders als bei der Pointerübergabe muss man die Dereferenzierung aber nicht von Hand vornehmen. Durch die Verwendung der Refrence-Variante entfällt eine ganze Reihe von Fehlerquellen (hauptsächlich durch die entfallenden Indirektionen) – die Programme werden somit erheblich sicherer. Im Grunde wird durch den „Call by Reference“-Aufruf der „Call by Pointer“ weitgehend überflüssig. Wegen der höheren Fehlerraten sollte man in neuen Programmen und Funktionen den „Call by Pointer“ gänzlich unterlassen. Die Funktion *tauschen* hat dann das Aussehen:

```
void tauschen (int& x, int& y)
{
   int z;

   z = x;
   x = y;
   y = z;
}
```

Es ist wichtig, die Unterschiede zwischen den Aufrufvarianten genau zu kennen. Die folgenden Programmbeispiele verdeutlichen die Unterschiede. Alle Programme sind identisch bis auf die Aufrufvariante. Die Zeilennummern sind lediglich Kommentare der relevanten Zeilen und dienen zur Orientierung und vertiefenden Erläuterung weiter unten. Die Nummerierung entspricht außerdem weitgehend der Ablaufreihenfolge. Zunächst das Programmbeispiel für den „Call by Value“:

```
//=========================================================
// Programm CBV.CPP
//=========================================================

#include <iostream>
```

```
#include <iomanip>

using namespace std;

void CallByValue (int x, int y);    // Vorwärtsdeklaration

void CallByValue (int x, int y)
{
   /*4*/ int z = x;
   /*5*/ x = y * 2;
   /*6*/ y = z;
   /*7*/ cout << "x = " << x << " und y = " << y << endl;
}

void main (void)                    // Hauptprogramm
{
   /*1*/ int a = 3, b = 5;
   /*2*/ cout << "a = " << a << " und b = " << b << endl;
   /*3*/ CallByValue (a, b);
   /*8*/ cout << "a = " << a << " und b = " << b << endl;
}

//-----------------------------------------------------
// Ergebnisausgaben :
// a = 3  und b = 5
// x = 10 und y = 3
// a = 3  und b = 5
```

Im zweiten Programmbeispiel werden nicht die Variablen, sondern lediglich ihre Speicheradressen an das aufgerufene Programm übergeben („Call by Pointer“):

```
//=============================================================
// Programm CBP.CPP
//=============================================================

#include <iostream>
#include <iomanip>

using namespace std;

void CallByPointer (int *x, int *y);   // Vorwärtsdeklaration

void CallByPointer (int *x, int *y)
{
   /*4*/ int z = *x;
   /*5*/ *x = *y * 2;
   /*6*/ *y = z;
   /*7*/ cout << "x = " << *x << " und y = " << *y << endl;
}

void main (void)                       // Hauptprogramm
{
   /*1*/ int a = 3, b = 5;
```

```
   /*2*/ cout << "a = " << a << " und b = " << b << endl;
   /*3*/ CallByPointer (&a, &b);
   /*8*/ cout << "a = " << a << " und b = " << b << endl;
}

//-----------------------------------------------------
// Ergebnisausgaben :
// a = 3  und b = 5
// x = 10 und y = 3
// a = 10 und b = 3
```

Als letztes die Variante „Call by Reference“, die nur unter C++ verfügbar ist.

```
//============================================================
// Programm CBR.CPP
//============================================================

#include <iostream>
#include <iomanip>

using namespace std;

void CallByReference (int& x, int& y); // Vorwärtsdeklaration

void CallByReference (int & x, int & y)
{
   /*4*/ int z = x;
   /*5*/ x = y * 2;
   /*6*/ y = z;
   /*7*/ cout << "x = " << x << " und y = " << y << endl;
}

void main (void)                    // Hauptprogramm
{
   /*1*/ int a = 3, b = 5;
   /*2*/ cout << "a = " << a << " und b = " << b << endl;
   /*3*/ CallByReference (a, b);
   /*8*/ cout << "a = " << a << " und b = " << b << endl;
}

//-----------------------------------------------------
// Ergebnisausgaben :
// a =  3 und b = 5
// x = 10 und y = 3
// a = 10 und b = 3
```

Kennzeichnend beim „Call by Pointer“ sind die Verweisoperatoren „*&*“ und „***“. Der Operator „*&*“ weist den Compiler an, nicht den gespeicherten Wert zu nehmen, sondern die Adresse an welcher dieser Wert gespeichert ist. Der Operator „***“ vor einer Zeigervariablen weist

den Rechner an, den Inhalt der angegebenen Adresse (Pointerinhalt) zu ermitteln oder zu ändern.

Vergleich der Parameter-Übergabevarianten									
	Call by Value C / C++			Call by Pointer C / C++			Call by Reference C++		
Quell-code Zeile	Adresse	Inhalt	Name	Adresse	Inhalt	Name	Adresse	Inhalt	Name
1	1000	3	a	1000	3	a	1000	3	a
	1002	5	b	1002	5	b	1002	5	b
3	1000	3	a	1000	3	a, *x	1000	3	a, x
	1002	5	b	1002	5	b, *y	1002	5	b, y
	2000	3	x	2000	1000	x			
	2002	5	y	2004	1002	y			
4	1000	3	a	1000	3	a, *x	1000	3	a, x
	1002	5	b	1002	5	b, *y	1002	5	b, y
	2000	3	x	2000	1000	x	2000	3	z
	2002	5	y	2004	1002	y			
	2004	3	z	2008	3	z			
5	1000	3	a	1000	10	a, *x	1000	10	a, x
	1002	5	b	1002	5	b, *y	1002	5	b, y
	2000	10	x	2000	1000	x	2000	3	z
	2002	5	y	2004	1002	y			
	2004	3	z	2008	3	z			
6	1000	3	a	1000	10	a, *x	1000	10	a, x
	1002	5	b	1002	3	b, *y	1002	3	b, y
	2000	10	x	2000	1000	x	2000	3	z
	2002	3	y	2004	1002	y			
	2004	3	z	2008	3	z			
8	1000	3	a	1000	10	a	1000	10	a
	1002	5	b	1002	3	b	1002	3	b

Tabelle 9-1: Vergleich der Parameter-Übergabevarianten

Anders beim „Call by Reference“, dessen Vorteile hier deutlich zu Tage treten. Es ist keine Referenzierung und Dereferenzierung per Hand nötig und dennoch ist das Ergebnis dem „Call by Pointer“ gleich (bei besserer Lesbarkeit).

Zur weiteren Verdeutlichung ist es am besten, sich den Inhalt des Speichers vorzustellen, die angegebenen Adressen (Tabelle 9-1) sind dabei fiktiv.

Wie leicht zu erkennen ist, bleiben die Variablen *a* und *b* im linken Ablauf („Call by Value“) unverändert. Bei *x* und *y* handelt es sich um Kopien von *a* und *b*.

Im mittleren Ablauf („Call by Pointer“) hingegen enthalten *x* und *y* die Adressen von *a* und *b*, sie verweisen auf die Speicherstellen, daher werden alle Veränderungen von *x* und *y* als Änderungen von *a* und *b* wirksam.

Gleiches gilt für den rechten Ablauf („Call by Reference“), nur dass hier die Ausgangsobjekte für den Entwickler sozusagen „Alias-Namen“ annehmen. Intern im Rechner läuft hingegen genau der gleiche Mechanismus ab wie im mittleren Fall. Das ganze Pointerhandling wird hier allerdings komfortabel vom Compiler erledigt.

9.2. FUNKTIONS-OVERLOADING

C++ erweitert den Sprachumfang von ANSI-C um das *Function-Overloading*, die Möglichkeit den gleichen Funktionsnamen mehrfach zu verwenden. Einzige Bedingung ist, dass die beiden Funktionen mit gleichem Namen für den Compiler eindeutig unterscheidbar sind. Diese Unterscheidung wird immer anhand der Parameterleiste getroffen. Als unterschiedlich gelten dabei Funktionen, die eine unterschiedliche Parameteranzahl (zu Einschränkungen dieser Regel siehe Abschnitt über Defaultparameter) und/oder unterschiedliche Parameterdatentypen aufweisen:

```
char *outfloat (float x, int v, int n)
{
   char fout [255] = "";
   sprintf (fout, "%*.*f", v, n, x);
   return (fout);
}
```

Überladene Funktionen müssen nicht, sollten aber gleiche oder zumindest sehr ähnliche Funktionalitäten ausführen. Ältere C++ Compiler benötigen das Schlüsselwort *overload*, gefolgt vom zu überladenen Funktionsnamen. Dies sollte ursprünglich ein Schutzmechanismus gegen „versehentliches“ Überladen von Funktionen sein. Es hat sich jedoch gezeigt, dass ein solcher Schutz nicht notwendig ist, da die Wahrscheinlichkeit eines versehentlichen *function overloading* sehr klein ist. Selbst wenn dieses „Versehen“ stattfindet ist es unkritisch, da der Compiler ja in der Lage ist beide Funktionen auseinanderzuhalten, allein die Lesbarkeit des Programms leidet darunter.

Muss man das Schlüsselwort *overload* verwenden, so ist im obigen Beispiel vor den beiden Funktionen die folgende Zeile einzufügen:

```
overload outfloat;
```

9.3. SCHACHTELUNG UND VORWÄRTSDEKLARATION

Anders als in anderen Programmiersprachen wie z.B. PASCAL oder ALGOL ist es in C/C++ nicht erlaubt Funktionsdefinitionen zu schachteln, d.h. lokale Funktionen zu definieren, alle Funktionen in C/C++ sind global.

```
//-----------------------------------------------------
// Illegal, diese Art der Verschachtelung ist in C/C++
// verboten
//-----------------------------------------------------

int abc ()
{
   int def ()   // Fehler!
   {
   }
}

void main (void)
{
}
```

Die Aufrufe von Funktionen können selbstverständlich geschachtelt werden, d.h. aus einem Unterprogramm heraus kann ein anderes Unterprogramm aufgerufen werden. Hierbei ist zu beachten, dass nur solche Unterprogramme aufrufbar sind, die vor der aufrufenden Funktion definiert wurden. In einem solchen Fall bleiben die lokalen Variablen erhalten, da die Funktion, aus der heraus eine weitere Funktion aufgerufen wurde, noch nicht abgeschlossen ist.

```
//-----------------------------------------------------
// Fehler, da Funktion def noch nicht bekannt ist !
//-----------------------------------------------------

int abc (int x)
{
   def (7);
}

//-----------------------------------------------------
// Erlaubt, Funktion abc ist bereits bekannt s.o.
//-----------------------------------------------------

int def (int y)
{
   abc (5);
```

```
}

void main (void)
{
}
```

Funktionen, die sich gegenseitig aufrufen sind durchaus nicht ungewöhnlich.

Daher besteht in C, wie in vielen anderen Sprachen auch, die Möglichkeit der sogenannten Vorwärtsdeklaration (Funktionsprototyp). Diese vereinbart eine Funktion, bevor sie definiert wird (ein Fehler wie im Beispiel oben wird dadurch vermieden). In der Vorwärtsdeklaration müssen Typ und Parameter der später zu definierenden Funktion vorab aufgeführt werden.

Viele Programmierer führen Vorwärtsdeklarationen für alle Funktionen auf, die innerhalb eines Programms oder Moduls vorkommen. Dadurch kann man bei umfangreichen Teilen schnell einen Überblick über die Funktionalität bestimmter Quelltexte gewinnen.

```
int def (int y);     // Vorwärtsdeklaration

int abc (int x)
{
   def (7);     // Erlaubt, da Funktion vorwärtsdeklariert ist
}

int def (int y)
{
   abc (5);     // Erlaubt, da Funktion bereits bekannt ist
}

void main (void)
{
   // So ist es richtig
}
```

Das Beispiel oben zeigt eine typische Vorwärtsdeklaration. Es handelt sich dabei immer um eine rudimentäre Funktionsvereinbarung ohne Funktionsrumpf.

9.3.1. PROTOTYPING ALS PFLICHT

Die Vorwärtsdeklaration macht den Namen und die Parameterliste der Funktion im Programm bekannt. So können diese benutzt werden, bevor die eigentlichen Arbeitsanweisungen der Funktion übersetzt worden sind.

Im Unterschied zu ANSI-C sind in C++ die Funktionsprototypen – mehr oder weniger – vorgeschrieben. Bei der Verwendung von Klassen (s.u.) müssen alle Methoden (Funktionsprototypen) schon innerhalb der Klassendeklaration aufgeführt werden.

Werden Funktionen überladen (eine Möglichkeit, die es in ANSI-C nicht gegeben hat), so ist eine Prototypangabe ebenfalls unumgänglich. Aus Gründen der Kompatibilität müssen aber auch noch die „alten“ C-Funktionen (d.h. Funktionen ohne Prototypen) übersetzt und gelinkt werden. Man sollte diese Form der Funktionserstellung bei neuen Projekten jedoch gänzlich vermeiden und – sofern möglich – auch bei der Änderung „alter“ Programme auflösen.

9.3.2. DEFAULT-PARAMETER

Eine weitere Neuerung von C++ betrifft die Möglichkeit bei der Deklaration von Funktionsprototypen sogenannte Default-Parameter festzulegen.

Dazu wird im Prototyp, also der Funktionsdeklaration in der Headerdatei, einem Parameter ein bestimmter Wert zugeordnet. Diese Zuordnung MUSS in der Headerdatei erfolgen und darf nicht bei der Implementation der Datei wiederholt werden.

```
#ifndef _TEST_H_
   #define _TEST_H_

   int abc (int x = 7);

#endif
```

Nun ist es erlaubt beim Aufruf der Funktion den vorbelegten Parameter fortzulassen, wenn der Wert den man sonst übergeben würde, der Vorbelegung entspricht:

```
#include <test.h>

int abc (int x)
{
   return (x+3);
}

void main (void);
{
   cout << abc (4) << endl; // Ergebnis 7
   cout << abc ()  << endl; // Ergebnis 10
}
```

Sinn der Vorbelegung ist es, sogenannte optionale Parameter zu bilden, denn häufig verlängert man die Parameterleisten von Funktionen, um

Sonderfälle abzufangen. Diese Sonderfälle erfordern oftmals die Übergabe eines anderen Wertes. Dieser soll dann anzeigen, dass ein bestimmtes Verhaltensmuster erwünscht ist (z.B. ob eine Ausgabe formatiert erfolgen soll, also z.B. mit führenden Nullen). Statt eine neue Funktion zu schreiben – und (bei Änderungen) den doppelten Wartungsaufwand in Kauf zu nehmen – verwendet man zumeist Zusatzparameter, die in 90% (oder mehr) immer den gleichen Wert besitzen. In C muss man nach der Erweiterung der Parameterleiste nun alle Funktionsaufrufe um den neu eingeführten Parameter ergänzen.

Nicht so in C++. Da man typischerweise gerade erst auf den Ersten Ausnahmefall gestoßen ist, ist dieser Fall auch der erste, wo man den Parameter braucht. Man legt also das bisher überall verwendete Verhalten als Default-Fall fest und übergibt den dazu benötigten Wert im Prototyp. Dadurch brauchen die anderen Aufrufe der Funktion nicht ergänzt werden, denn diese fügen den fehlenden Parameter beim Compilerlauf automatisch aus dem Prototyp hinzu. Der Zusatzparameter kann im „Standardfall“ also entfallen.

Es gibt jedoch eine Reihe von Einschränkungen, die man bei der Verwendung von Default-Parametern beachten muss. So dürfen die Vorgabewerte nur von rechts beginnend (vom Ende der Parameterliste) vergeben werden und es dürfen keine „Lücken“ entstehen:

```
abc (int x,   float y,     int z=TRUE); // erlaubt
abc (int x,   float y=5.0, int z=TRUE); // erlaubt
abc (int x=2, float y=5.0, int z=TRUE); // erlaubt

abc (int x=2, float y,     int z=TRUE); // verboten
abc (int x,   float y=5.0, int z     ); // verboten
abc (int x=2, float y,     int z     ); // verboten
abc (int x=2, float y=5.0, int z     ); // verboten
```

Zusätzlich muss der Entwickler darauf achten, dass keine Doppeldeutigkeiten entstehen (z.B. durch Overloading), da der Rechner sonst nicht entscheiden kann, welche Funktion aufzurufen ist. Solche Fehler meldet ein C++-Compiler üblicherweise als *ambiguity error*.

Beschränkt man sich bei der Vorbelegung auf die letzten beiden Parameter, so hat man anschließend die folgenden, legalen Aufrufmöglichkeiten für die Funktion *abc*:

```
//============================================================
// Programm DEFPARAM.CPP
//============================================================

#include <iostream>
```

```
using namespace std;

#define TRUE 1

int abc (int x=2, float y=5.0, int z=TRUE)
{
   cout << "x : " << x << endl;
   cout << "y : " << y << endl;
   cout << "z : " << z << endl;
   return (x+3);
}

void main (void)
{
   abc (4, 3.0, 12);  // Kein Default benutzt
   abc (4, 3.0);      // dritter Parameter ist TRUE
   abc (4);           // 2. und 3. Parameter sind 5.0 und TRUE
}
```

9.4. REKURSIONEN

C/C++ ist als Programmiersprache voll rekursionsfähig, d.h. jeder Funktion ist es erlaubt sich selbst aufzurufen. Dabei werden alle lokalen Variable der Funktion neu angelegt. Die Sicherung der gerade aktuellen lokalen Werte erfolgt auf dem Stack (Stapelspeicher):

```
//============================================================
// Programm FACULREC.CPP
//============================================================

#include <iostream>

using namespace std;

long i = 0;

long facul (long n)
{
   if (n > 1)
   {
      return (n * facul (n-1));
   }
   else
   {
      return (1);
   }
}

void main (void)
{
   i = facul (7);
   cout << i;
}
```

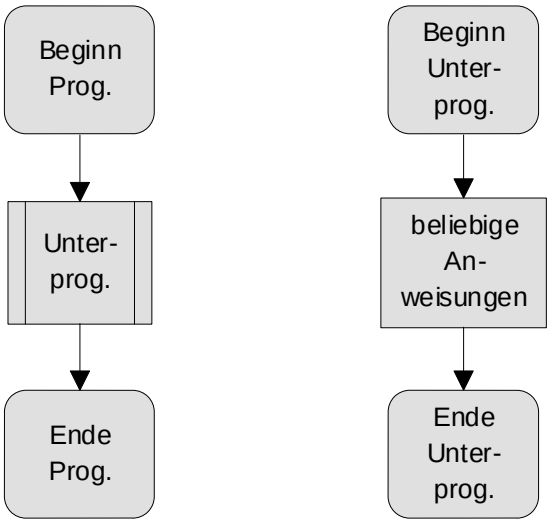
Ein rekursiver Aufruf erfolgt typischer Weise in einer bedingten Verzweigung, welche die Abbruchbedingung der Rekursion abfragt. Der Programmierer trägt die Verantwortung für den Abbruch der Rekursion selbst. Erfolgt kein Abbruch, so wird die Rekursion bis zum Speicherüberlauf fortgeführt (ein Fall der fast immer mit einem Systemabsturz endet).

Zu besonderen Schwierigkeiten bei der Abbruchkontrolle kann es führen, wenn die Rekursion nicht direkt erfolgt (wie im obigen Beispiel), sondern indirekt, durch mehrere sich gegenseitig aufrufende Funktionen.

9.5. DIE FUNKTION MAIN

In jedem ausführbaren C/C++ Programm muss es genau eine Funktion mit dem Namen *main ()* geben. Diese stellt die Hauptfunktion (Hauptprogramm) dar, bei welcher mit der Abarbeitung des Programms begonnen wird. Das Hauptprogramm hat keine vorgeschriebene Position innerhalb der Reihenfolge der Funktionen, kann also an beliebiger Stelle stehen. Üblicherweise ist *main* am Ende zu finden, da dadurch Teile der (sonst unbedingt notwendigen) Vorwärtsdeklarationen entfallen können.

9.6. FLUSSDIAGRAMMSYMBOL FÜR FUNKTIONEN



10. Abbildung 9-1 – Struktogrammsymbol Unterprogramm

Eigene Unterfunktionen werden in Flussdiagrammen durch ein Rechteck oder durch ein Rechteck (ähnlich der einfachen Anweisung) mit zusätzlichen senkrechten Linien dargestellt s.u.). Die letztere Variante wird benutzt, wenn die Funktion in einem eigenen Flussdiagramm detailliert wird.

10.1. AUFGABENTEIL

Bei allen Aufgaben ist ein Schwierigkeitsgrad angegeben. Der Schwierigkeitsgrad bezieht sich nicht allein auf die Aufgabenstellung sondern ggf. auch auf den Umfang der Ausgabe.

10.1.1. AUFGABE 1 (EINFACH)

Erweitern Sie das Programm zur Lohnsteuerberechnung in Porto Banana (siehe Aufgabe 7.6.1). Benutzen Sie zur Gestaltung des Programms weitgehend Funktionen.

10.1.2. AUFGABE 2 (EINFACH)

Schreiben Sie eine Funktion, welche die Fakultät einer Zahl berechnet. Die Fakultät einer Zahl besteht aus der Integerzahl selbst, multipliziert mit all ihren Vorgängern bis Null.

So berechnet sich z.B.

```
die Fakultät der Zahl 3 aus: 1 * 2 * 3
die Fakultät der Zahl 5 aus: 1 * 2 * 3 * 4 * 5
```

Wie lautet die größte Zahl, von der man die Fakultät mit Hilfe einer Variablen vom Typ *long double* ermitteln kann?

10.1.3. AUFGABE 3 (MITTEL)

Schreiben Sie eine Funktion, welche die Fakultät einer Zahl rekursiv berechnet. Die Fakultät einer Zahl besteht aus der Integerzahl selbst, multipliziert mit all ihren Vorgängern bis Null.

So berechnet sich z.B.

```
die Fakultät der Zahl 3 aus: 1 * 2 * 3
die Fakultät der Zahl 5 aus: 1 * 2 * 3 * 4 * 5
```

Wie lautet die größte Zahl, von der man die Fakultät mit Hilfe einer Variablen vom Typ *long double* ermitteln kann?

11. PRÄPROZESSOR-ANWEISUNGEN

Der C/C++-Präprozessor ist eine vorgeschaltete Übersetzungsstufe, die dazu verwendet wird z.B. symbolische Konstanten durch Werte zu ersetzen oder Headerdateien einzufügen. Alle Anweisungen und Auswertungen des Präprozessors werden mit dem Zeichen „#“ eingeleitet. Anders als Programmbefehle wird die Auswertung mit dem Zeilenende und nicht mit einem Semikolon abgeschlossen. Soll (oder muss) eine Präprozessor-Anweisung über mehr als eine Programmzeile gehen, so kann die Anweisung durch den Backslash (\) auf mehrere Zeilen aufgeteilt werden.

11.1. DIE #INCLUDE ANWEISUNG

Die *include*-Anweisung wird verwendet, um an einer bestimmten Stelle des Programmtextes einen anderen Quelltext (meist eine Headerdatei) einzubinden. Die folgende Tabelle gibt eine Übersicht über die Headerdateien, die laut ANSI-Standard bindend vorgeschrieben sind.

Die meisten Compiler enthalten eine Vielzahl weiterer Headerdateien, welche die Funktionalität des Compilers (je nach Lieferumfang) erheblich erweitern. Ein Großteil dieser Dateien ist jedoch hochgradig abhängig vom Betriebssystem oder der Hardware und muss daher in Programmen, die zur Portierung auf andere Systeme vorgesehen sind, vermieden werden:

Standard Header Dateien in ANSI-C	
ASSERT.H	Zusicherungen (Assertion) Debugging Hilfsfunktion für Zusatzinformation bei Programmabbruch
CTYPE.H	Zeichentypen (Charactertype) Definition von Funktionen zur Erkennung von ASCII-Zeichen (isalpha etc.), sowie Umwandlung von Klein- und Großbuchstaben
ERRNO.H	Fehlernummer (Errornumber) Fehlermeldungen, Auswertung der internen Fehlernummern, insbesondere für mathematische Funktionen
FLOAT.H	Fließkommazahlen (Floatingpoint) Funktionen zur Darstellung von Fließkommazahlen, Bestimmung der Rundungsmethode etc.
LIMITS.H	Grenzwerte (LIMITs) Minimal- und Maximalwerte der einfachen Datentypen
LOCALE.H	Lokale Einstellungen (LOCALE) Länderspezifische Besonderheiten (z.B. Währungszeichen)
MATH.H	Mathematische Funktionen (Mathematics) u.a. trigonometrische Funktionen (Sinus / Kosinus / Tangens) sowie Funktionen zur Fließkommazahlendarstellung, Rundungsmethode etc.
SETJMP.H	Sprungadresse setzen (Set Jumpaddress) Debuggingfunktionen für Sprungfehler

Standard Header Dateien in ANSI-C	
SIGNAL.H	Signalverarbeitung (Signals) Interrupt-Handling und zur Steuerung bei Laufzeitfehlern
STDARG.H	Parameterhandling (Standard Argumenthandling) Definitionen zum Aufbau von Funktionen mit Parameterlisten mit variabler Länge
STDDEF.H	Standarddefinitionen (Standard Definitions) Standardmakros, main
STDIO.H	Standard Ein- und Ausgabe (Standard Input-Output) Datentypsynonyme (Word = unsigned int), Standardkonstanten wie z. B. EOF (End of File) etc.
STDLIB.H	Standardbibliothek (Standard Library) Sonstige Funktionen, die nicht zu einer der anderen Headerdateien passen (Zufallszahlen / Umwandlung von Zeichenketten in Werte)
STRING.H	Zeichenkettenfunktionen (Stringfunctions) Vergleich, Kopieren, Suchen usw.
TIME.H	Zeitfunktionen (Timefunctions) Systemzeit- und Datumsfunktionen

Tabelle 11-1: Standard-Headerdateien in ANSI-C

Die einzubindenden C/C++-Quelldateien können sowohl Anweisungen (Funktionen), wie auch Deklarationen oder Vorwärtsdeklarationen (siehe Abschnitt über Funktionen) enthalten.

Die Deklaration von Variablen in Headerdateien ist nicht empfehlenswert, da es bei mehrfacher Einbindung in verschiedenen Modulen leicht zu Fehlern kommen kann.

Ein Fehler tritt immer dann auf, wenn globale Variablen enthalten sind, die dann natürlich mehrfach deklariert werden (Verletzung der Namenseindeutigkeit), es sei denn es handelt sich um eine external-*Vereinbarung (siehe Abschnitt über Speicherklassen).*

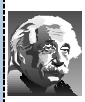
Das Einfügen von Funktionen ist ebenfalls ungünstig, da die eingebundenen Quelltexte bei jedem Compileraufruf mit übersetzt werden. Um unnötigen Zeitaufwand zu vermeiden empfiehlt es sich solche Programmteile separat zu übersetzen und einfach zusammen zu linken, bzw. fertige Funktionen mit allgemeinem Charakter in eine eigene Bibliothek zu stellen.

```
Syntax:
   #include "Name"
   #include <Name>
```

Wird der Dateiname in spitzen Klammern (<*Name*>) geschrieben, wird danach im Standardpfad für include-Dateien gesucht (sofern ein solcher angegeben wurde, was bei den meisten Compilern über die integrierte Oberfläche zwingend der Fall ist). Wird der Dateiname in

Anführungsstrichen geschrieben, so wird zunächst im Standardpfad für include-Dateien und anschließend noch im aktuellen im Verzeichnis (Directory) gesucht.

Der Compiler nimmt die erste Datei des angegebenen Namens die er findet. Achten Sie also darauf eindeutige Namen zu verwenden und vermeiden Sie es unterschiedliche Versionen der gleichen (eigenen) Headerdatei im Pfad zu halten.



Die Angabe der Quelldateinamen kann auch den kompletten Pfad beinhalten, um z.B. eigene Headerdateien in einem eigenen Verzeichnis zu sammeln.

```
#include <iostream>
#include "my_inc1.h"           // im akt. Verzeichnis suchen
#include "basis\my_inc2.h"     // im Unterverz. BASIS des
                               // aktuellen Verzeichnis suchen
#include "d:\basis\my_inc3.h" // Datei genau dieser Stelle

using namespace std;

void main (void)
{
   cout << "Hallo Welt !";
}
```

Include-Anweisungen lassen sich verschachteln, d. h. innerhalb einer eingebundenen Datei kann wiederum eine Include-Anweisung stehen. Um ein endloses gegenseitiges Aufrufen (Rekursion) zu vermeiden, ist darauf zu achten, dass jede Headerdatei nur einmal ausgewertet wird. Die genaue Vorgehensweise wird im Abschnitt über den #*if*-Befehl (s.u.) geschildert.

11.2. DIE #DEFINE ANWEISUNG

Der #*define*-Befehl vereinbart eine symbolische Konstante, die beim Lauf des Präprozessors, mittels Textersetzung, durch den angegebenen Wert ersetzt wird.

Es gilt in C/C++ als guter Stil, alle symbolische Konstanten die mit #define erzeugt wurden mit Großbuchstaben zu benennen, um sie eindeutig von Variablen- und Funktionsvereinbarungen unterscheiden zu können.

Die #*define*-Anweisung entspricht weitgehend der Deklaration symbolischer Konstanten in anderen Programmiersprachen, ist aber flexibler, da hier eine reine Textersetzung durchgeführt wird. So lassen sich z.B. auch die geschweiften Klammern durch die Worte *BEGIN* und

END ersetzen, so dass eine Annäherung an die Pascal-Syntax möglich wäre.

```
Syntax:
   #define SYMBOLISCHER-NAME Ersatztext
```

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

using namespace std;

#define BEGIN   {
#define END     }
#define WORD    unsigned int
#define HALLO   "Guten Morgen "
#define MWST    1.15
#define OUTPUT  "%s Zahl * MWST ist: %5.2f"

WORD Zahl1 = 33;

void main (void)
BEGIN
   WORD Zahl2 = 1;

   cout << "Werte : " << HALLO << Zahl1 * MWST << endl;
   printf (OUTPUT, HALLO, Zahl1 * MWST);
END
```

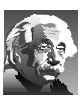
Das obige Beispiel zeigt eine ganze Reihe von Textersetzungen. Wichtig ist festzuhalten, dass nur Texte in Anweisungen, nicht aber in Textkonstanten ersetzt werden. Daher kann man in *OUTPUT* problemlos *MWST* als Text verwenden. Das Beispiel oben wird durch den Präprozessor, vor der Übersetzung, zu folgendem Text geändert:

```
...

unsigned int Zahl1 = 33;

void main (void)
{
   unsigned int Zahl2 = 1;

   cout << "Werte : " << "Guten Morgen "<< Zahl1*1.15 << endl;
   printf ("%sZahl * MWST ist: %5.2f", "Guten Morgen ",
           Zahl1*1.15);
}
```



Durch die Verwendung von #define-Anweisungen kann die Lesbarkeit des Quelltextes enorm erhöht werden. Es ist daher

ratsam, vor allem Rückgabewerte von Funktionen, die Status- oder Fehlerinformationen darstellen, durch symbolische Namen zu belegen - z.B.

```
  if (Ergebnis = = DATEI_NICHT_GEFUNDEN)
```

statt

```
  if (Ergebnis = = 0)
```

Nach Definition des ANSI-Standards ist es sogar möglich, *#define*-Anweisungen rekursiv aufzurufen, wobei ein vorher vorhandener, symbolischer Wert überschrieben wird:

```
#define ZW   10
#define HZW  100*ZW
#define ZW   HZW/2
```

Allerdings ist es nicht sonderlich übersichtlich, solche rekursiven Definitionen häufiger zu verwenden, da sie das Verständnis des Programmtextes doch erheblich vermindern. Stattdessen sollte besser eine neue symbolische Konstante eingeführt werden.

11.3. DIE #UNDEF ANWEISUNG

Der *#undef*-Befehl macht die Definition einer Konstanten über Textersetzung (*#define*) unwirksam. Das Löschen einer Textersetzungs- oder Makrodefinition (siehe Kapitel über Makros) führt dazu, dass eine Konstante oder ein Makro ab dem *#undef*-Befehl nicht mehr ersetzt wird. Der Makro- oder Konstantenname kann mit neuen Werten oder Anweisungen belegt werden.

```
Syntax:
   #undef SYMBOLISCHER-NAME
   #undef MAKRO-NAME

Beispiel:
   #undef MAX
   #undef BEGIN
```

Von der Löschung und Neubelegung sollte nur sehr sparsam Gebrauch gemacht werden, da die häufige Änderung eines symbolischen Namens ein Programm undurchschaubar machen kann.

11.4. DIE #IF ... #ENDIF ANWEISUNG

Die Befehle *#if ... #endif* oder *#if ... #else ... #endif* werden meist zur bedingten Kompilierung oder zur Steuerung der Headerdatei-Auswertung eingesetzt.

```
Syntax:
   #if KONSTANTE
   #if KONSTANTER_LOGISCHER_AUSDRUCK
```

Hierbei wird untersucht, ob die genannte Konstante einen von Null verschiedenen Wert besitzt. Wenn ja, werden die Anweisungen zwischen *#if* und *#endif* ausgeführt. Eine Konstante besteht aus Ausdrücken von *int*- und *char*-Konstanten, die mit allen Operatoren außer der Zuweisung verbunden sein können. Funktionsaufrufe sind nicht erlaubt.

```
Syntax:
   #ifdef SYMBOLISCHER_bzw_MAKRO_NAME
   #ifndef SYMBOLISCHER_bzw_MAKRO_NAME

   ANSI-Syntax:
      #if defined SYMBOLISCHER_bzw_MAKRO_NAME
      #if !defined SYMBOLISCHER_bzw_MAKRO_NAME
```

Hierbei wird untersucht, ob bereits eine Konstante bzw. ein Makro mit dem angegebenen Namen existiert (*#ifdef*) oder ob der Name noch frei ist (*#ifndef*). Abhängig davon werden die Anweisungen zwischen *#ifdef*, *#ifndef* und *#endif*, bzw. *#else* und *#endif* ausgeführt.

```
Syntax:
   #else
   #elif KONSTANTE_ODER_VERGLEICH
```

Diese Anweisungen sind eingeführt worden um eine volle syntaktische Äquivalenz zum *if-else* zu haben, sie bezeichnen die Alternativen zu *#if*, *#ifdef* und *#ifndef*. Einige Compiler definieren neben der ANSI-Anweisung *#elif* auch die Anweisung *#elsif* (*else if*). Der *#endif*-Befehl beendet eine Anweisung zur bedingten Kompilierung oder zur bedingten Headerdatei-Auswertung in C/C++.

```
Syntax:
   #endif
```

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

using namespace std;

#ifndef EOF
   #define EOF -1
   #define KONTROLLTEXT "EOF war nicht definiert"
#else
   #if EOF==0
```

```
      #undef EOF
      #define EOF -1
   #endif
   #define KONTROLLTEXT "EOF redefiniert"
#endif

void main (void)
{
   cout << KONTROLLTEXT;
}
```

Der *#ifndef*-Befehl hat besondere Bedeutung bei der Auswertung von *#include*-Befehlen. Da diese sich gegenseitig aufrufen dürfen, ist es notwendig, dafür zu sorgen, dass keine Endlosschleife entsteht. Die übliche Vorgehensweise besteht darin, eine symbolische Konstante einzuführen:

```
//-------------------------------------------------------------
// AUFBAU EINES EIGENEN HEADERFILES (hier MYHEADER.H)
//-------------------------------------------------------------

#ifndef _MYHEADER_H_     // ifndef dient dazu einen rekursiven
   #define _MYHEADER_H_  // Aufruf zu verhindern

   #include <my2.h>      // Header kann andere Header
                         // includen

   #define TRUE  1       // Header kann #defines enthalten
   #define FALSE 0

   // ... weitere Anweisungen

#endif
```

Bei der ersten Auswertung der Headerdatei wird auf das Vorhandensein der symbolischen Konstante geprüft und festgestellt, dass diese nicht existiert. Entsprechend werden die Befehle zwischen *#ifndef* und *#endif* ausgewertet. Diese enthalten u.a. die Instruktion, die symbolische Konstante anzulegen, die für den restlichen Compilerlauf verhindert, dass die Befehle der Headerdatei ein zweites Mal ausgewertet werden können.

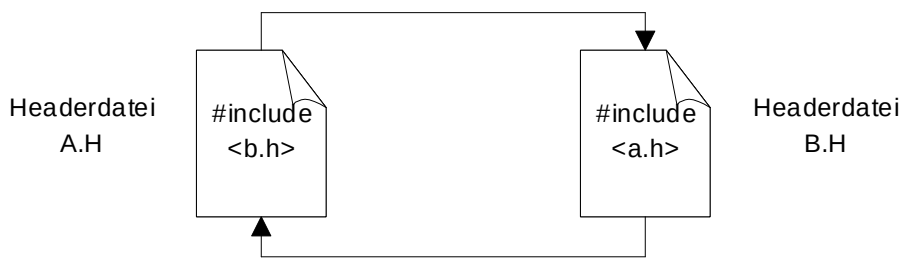


Abbildung 11-1 – #include-Probleme

11.5. DIE #PRAGMA ANWEISUNG

Die #*pragma*-Anweisung ist erst mit dem ANSI-Standard eingeführt worden und ermöglicht die Nutzung von Compilerbesonderheiten (z.B. Erzwingen der Codierung für einen bestimmten mathematischen Koprozessor).

```
Syntax:
   #pragma COMPILER-DIREKTIVE
```

Die Anweisungen, die einem #*pragma*-Befehl folgen können sind dementsprechend von Compiler zu Compiler (und auf unterschiedlicher Hardware) verschieden.

11.6. DIE #ERROR ANWEISUNG

Die #*error*-Anweisung ist ebenfalls erst mit dem ANSI-Standard eingeführt worden. Der Befehl ermöglicht die Ausgabe eigener Fehlermeldungen zur Übersetzungszeit (Compilerfehler, Syntaxfehler etc.):

```
Syntax:
   #error BENUTZEREIGENE_FEHLERMELDUNG

Beispiel:
   #if defined (_ANSI_C_) && defined (_OLD_C_)
      #error Ist nicht möglich, bitte entscheiden
   #endif
```

11.7. AUFGABENTEIL

Bei allen Aufgaben ist ein Schwierigkeitsgrad angegeben. Der Schwierigkeitsgrad bezieht sich nicht allein auf die Aufgabenstellung sondern ggf. auch auf den Umfang der Ausgabe.

11.7.1. AUFGABE 1 (EINFACH)

Schreiben Sie eine eigene Headerdatei, in der Sie eine Reihe von Konstanten definieren.

Achten Sie auf eine saubere Abgrenzung der Headerdatei gegenüber rekursiven Aufrufen.

Definieren Sie folgende Konstanten:

- Den logischen Wert TRUE
- Den logischen Wert FALSE
- Die Mehrwertsteuer als Prozentsatz
- Die Mehrwertsteuer als Faktor
- Texte für die Wochentage

12. MAKROS

In C können Makros nur über den Präprozessor erzeugt werden, eine Methode, die – wie man im folgenden Abschnitt sehen wird, sehr anfällig für Fehler ist. Aus diesem Grund hat man in C++ nach einer anderen Lösung gesucht und die Inline-Makros in Form einer Funktionsdeklaration zugelassen, die vom Compiler automatisch in ein Makro umgewandelt wird.

12.1. #DEFINE ALS MAKRO-DEFINITION

Die Makrodefinition mit *#define* ist im Prinzip nur die Erweiterung der Textersetzung um Parameter, wobei die Parameterliste unmittelbar an den Makronamen anschließen muss. D.h. zwischen Namen und Parameterliste darf sich kein Leerzeichen befinden, da der Compiler alle Zeichen nach einem Leerzeichen als Teil des Ersetzungstextes betrachtet. Die Parameter werden durch die später im Programmtext angegebenen Variablen und Konstanten ersetzt.

Wie *#define*-Anweisungen können auch mehrzeilige Makroanweisungen geschrieben werden, wenn die Zeile mit dem „ \ “-Zeichen beendet wird. Makros sind nicht unproblematisch und führen oftmals zu schwer entdeckbaren Fehlern.

```
Syntax:
   #define MAKRO-NAME(Parameterliste) Anweisung
```

```
//============================================================
// Programm MAKRO1.CPP
//============================================================

#include <iostream>
#include <iomanip>

using namespace std;

#define MAX(x,y) (((x)<(y))?(y):(x))
#define MIN(x,y) (((x)<(y))?(x):(y))
#define ABS(x)   ((x)<0?-(x):(x))

int x = 1;
int y = 2;
int z = -3;

void main (void)
{
   x = MAX(x,y);      // x wird  2
   cout << x << endl;

   y = MIN(y,z);      // y wird –3
   cout << y << endl;
```

```
   z = ABS(z);         // z wird  3
   cout << z << endl;
}
```

Die Beispiele scheinen einen geradezu verschwenderisch erscheinenden Gebrauch an Klammern zu zeigen. Dennoch sind diese unbedingt notwendig, um versteckte, kaum auffindbare, semantische Fehler zu umgehen. Die folgenden Beispiele, in denen Makros Schritt für Schritt entwickelt und verbessert werden, zeigen die Vielzahl versteckter Fehler auf und verdeutlichen die Problematik.

```
#define QUAD(x) x*x
```

Das Makro zur Bildung des Quadrats einer Zahl, scheint auf den ersten Blick einleuchtend und korrekt zu sein - es wird meistens auch richtige Ergebnisse liefern. Der folgende (häufige) Aufruf des Makros wird z.B. richtig verarbeitet:

```
i = QUAD (j);

// wird in i = j*j; übersetzt
```

Anders im folgenden Fall, bei welchem die mangelnde Klammerung zu einem inhaltlichen Fehler durch die Operatorprioritäten führt. Erst nach der Übersetzung ist problemlos zu erkennen, dass das Makro offensichtlich zu falschen Ergebnissen führt:

```
i = QUAD (j+1);

// wird in i = j+1*j+1; übersetzt
```

Das Ergebnis ist offensichtlich nicht das Quadrat der Summe $j+1$ sondern das Resultat der Berechnung $2j+1$. Die Lösung des Problems liegt natürlich in der Makrodefinition, die, um solche Fälle zu vermeiden, wie folgt aussehen muss:

```
#define QUAD(x) ((x)*(x))
```

Jetzt wird der Aufruf auch mit berechneten Parametern korrekt durchgeführt und erbringt das erwartete Ergebnis:

```
i = QUAD (j+1);

// wird in i = ((j+1)*(j+1)); übersetzt
```

Ein weiteres, leider unlösbares Problem bei der Verwendung von Makros sind unbeabsichtigte Seiteneffekte. Wie im folgenden Beispiel, bei dem eine zuweisende Operation in der Makroklammer zu Fehlern führt:

```
i = QUAD (j++);

// wird in i = ((j++)*(j++))übersetzt
```

In diesem Beispiel wird die Variable *g* gleich zweimal inkrementiert und das Ergebnis dadurch verfälscht. Es sollte daher strikt vermieden werden, variablenverändernde Operationen in einer Makroanweisung zu verwenden.

Ein weiteres Problem kann sich ergeben, wenn Makros mit mehreren Anweisungen definiert werden:

```
#define PRINTADD(a) cout << a; a=a+1
```

Auch diese Makro-Anweisung wird im Normalfall korrekt arbeiten, in Verbindung mit bedingten Verzweigungen und Schleifen kommt es aber zu Fehlern wie dem folgenden:

```
if (0 == b) PRINTADD (b);

// wird in if (b == 0) cout << b; b=b+1; übersetzt
```

Wie leicht zu erkennen ist, wird die Anweisung *b=b+1;* auf jeden Fall ausgeführt - unabhängig davon, ob die *if*-Bedingung *TRUE* oder *FALSE* ist. Die naheliegende Lösung besteht darin, das Makro mittels geschweifter Klammern zu einem Block zu binden:

```
#define PRINTADD(a) {cout << a; a=a+1}
```

Diese Variante schafft zwar im Fall der einfachen bedingten Verzweigung Abhilfe, ist aber letztlich ebenso falsch wie der erste Lösungsversuch. Der Fehler tritt jetzt auf, wenn versucht wird, das Makro in einem *if-else*-Konstrukt zu verwenden:

```
if (b == 0) PRINTADD (b);
else cout << "Sonst";

// wird in if (b == 0) {cout << b; b=b+1};
//         else cout << "Sonst"; übersetzt
```

Da die *if*-Anweisung mit dem Schließen der geschweiften Klammer abgeschlossen ist, bezieht sich die *else*-Anweisung syntaktisch auf die

Leeranweisung „;“. Störend ist also das Semikolon nach der geschweiften Klammer. Diese verhindert, dass dem *if* ein *else* folgen kann. Der Compiler wird in diesem Fall eine Fehlermeldung ausgeben, die besagt, dass ein *else*-Befehl ohne zugehörige *if*-Anweisung vorliegt. Noch schwieriger zu finden wird der Fehler, wenn von vorher (durch Verschachtelung) noch eine offene *if*-Anweisung besteht, die nun - inkorrekter Weise, aber syntaktisch richtig - mit der *else*-Anweisung verbunden wird (siehe dazu auch das Kapitel über „bedingte Verzweigungen“). Die einzige Möglichkeit, dieses Dilemma zu lösen, ist die Verwendung des Komma-Separators anstelle der geschweiften Klammern. Der Separator verhindert, dass die Makroanweisung voneinander getrennt behandelt werden können, die Anweisung wird aber auch nicht syntaktisch als Block abgegrenzt:

```
#define PRINTADD(a) cout << a, a=a+1
```

Der Fehler bei der Einbindung des Makros in *if-else*-Anweisungen tritt jetzt nicht mehr auf. Leider macht die Verwendung des Kommaseparators Programme und Makros schnell unübersichtlich, so dass eine ausgiebige Verwendung nicht unbedingt ratsam ist:

```
if (b == 0) PRINTADD (b);
else cout << "Sonst";

// wird in if (b == 0) cout << b, b=b+1;
//         else cout << "Sonst"); übersetzt
```

Die Verwendung von Makros bietet dennoch einige Vorteile. So werden Quellprogramme meist lesbarer und die erzeugten Programme schneller (die lokalen Variablen müssen nicht erst über den Stack gesichert und wiederhergestellt werden). Der vielleicht entscheidende Vorteil ist die Tatsache, dass Makros mit allen Zahlentypen zusammenarbeiten, da keine typisierten Parameter notwendig sind.

Auf der anderen Seite sind Makros recht anfällig für Fehler und vergrößern die ausführbaren Programme, da der gleiche Programmcode mehrfach erzeugt wird.

Die folgende Tabelle zeigt die vordefinierten Makros, wie sie im ANSI-Standard festgelegt sind:

ANSI Debug Informations-Makros	
Makroname	Bedeutung
_ _FILE_ _	ist der Name der Quelldatei als Zeichenkette
_ _TIME_ _	ist die Übersetzungszeit als Zeichenkette HH:MM:SS
_ _DATE_ _	ist das Übersetzungsdatum als Zeichenkette

ANSI Debug Informations-Makros	
Makroname	Bedeutung
	MMMM TT JJ
_ _STDC_ _	ganzzahlige Konstante als Unterscheidung zwischen dem Kernighan/Ritchie und dem ANSI-Standard (_ANSI_C_ oder _OLD_C_)
_ _LINE_ _	Nummer der aktuellen Quellcodezeile

Tabelle 12-1: ANSI Debug-Information Makros

12.2. INLINE-MAKROS

Wie anhand der Beispiele leicht ersichtlich ist, ist die Programmierung von Präprozessormakros sehr schwierig und anfällig für Fehler. Um dieses zu umgehen, hat man in C++ die Möglichkeit von *inline*-Funktionen hinzugefügt. Dies sind keine Funktionen im eigentlichen Sinne, denn der entsprechende Code der *inline*-Funktion wird an der Aufrufstelle direkt eingefügt (statt eines Funktionsaufrufes mit Übergabe der Parameter auf dem Stack). Dadurch wird zwar der ausführbare Programmcode verlängert, da der Code an jeder Aufrufstelle abgelegt ist, andererseits wird die Lesbarkeit des Programms genauso erhöht, als wäre es eine echte Funktion.

```
Syntax:
   inline returntyp funktionsname (Parameterliste)
   {
      Anweisungen
   }
```

Dadurch dass die *inline*-Makros programmtechnisch wie eine Funktion gehandhabt werden, entfallen die Probleme, die sich, wie oben gezeigt, aus der Makrodefinition ergeben können.

```
//============================================================
// Programm MAKRO2.CPP
//============================================================
#include <iostream>
#include <iomanip>

using namespace std;

inline int MAX(int x, int y) { return (x<y ?  y : x);}
inline int MIN(int x, int y) { return (x<y ?  x : y);}
inline int ABS(int x)        { return (x<0 ? -x : x);}

int x =  1;
int y =  2;
int z = -3;

void main (void)
{
```

```
   x = MAX(x,y);        // x wird  2
   cout << x << endl;

   y = MIN(y,z);        // y wird –3
   cout << y << endl;

   z = ABS(z);          // z wird  3
   cout << z << endl;
}
```

Allerdings entfällt, wie man sehen kann, auch einer der größten Vorteile der Makros – die Typunabhängigkeit.

Dieser Verlust lässt sich aber durch einen von zwei C++ Mechanismen ausgleichen, dem *Function-Overloading* (siehe oben) und den *Templates* (siehe unten). Bei der Funktionsüberladung wird der Name der Funktion (hier der *inline*-Function) erneut in einer Funktionsdefinition verwendet. Bedingung ist allerdings, dass sich die Parameterleiste der Überladungsfunktion von der ursprünglichen unterscheidet. Eine Unterscheidung allein anhand des Returntyps ist nicht möglich. So können alle benötigen Makros über die *inline*-Funktionen abgedeckt werden (hier am Beispiel der *MAX*-Funktion):

```
inline int   MAX (int   x, int   y) { return (x<y ? y : x); }
inline long  MAX (long  x, long  y) { return (x<y ? y : x); }
inline float MAX (float x, float y) { return (x<y ? y : x); }
```

Der zweite Mechanismus, die Verwendung von *Templates*, ist eleganter und vermeidet auch die Notwendigkeit des Function-Overloading von Hand.

12.3. AUFGABENTEIL

Bei allen Aufgaben ist ein Schwierigkeitsgrad angegeben. Der Schwierigkeitsgrad bezieht sich nicht allein auf die Aufgabenstellung sondern ggf. auch auf den Umfang der Ausgabe.

12.3.1. AUFGABE 1 (EINFACH)

Schreiben Sie ein Makro für die SGN-Funktion. Das SGN-Makro soll das Vorzeichen einer Zahl oder das resultierende Vorzeichen eines Ausdrucks zurückgeben.

12.3.2. AUFGABE 2 (EINFACH)

Erstellen Sie für das SGN-Makro ein eigenes Headerfile.

13. TEMPLATES

Templates sind Schablonen, bei denen die eigentlichen Datentypen ausgespart werden. Der Compiler sorgt dafür, dass alle benötigten Varianten der Funktion automatisch erzeugt werden. Ruft man z.B. eine über *template* definierte Funktion mit *double* und *long* Daten auf, so erzeugt der Compiler die Überladungsfunktion dafür selbständig – genau wie ein Makro:

```
//============================================================
// Programm TEMPLAT1.CPP
//============================================================

#include <iostream>
#include <iomanip>

using namespace std;

template <class T> T MAX (T x, T y)
{
   return (x > y) ? x : y;
};

void main (void)
{
   cout << MAX (100, 111) << endl;    // Ergebnis 111
   cout << MAX (1.3, 1.1) << endl;    // Ergebnis 1.3
}
```

Um ein *template* einzusetzen ist zunächst die Deklaration eines abstrakten Datentyps notwendig, hier *T*. Dieser abstrakte Datentyp kann (da als *template* gekennzeichnet) vom Compiler durch die benötigten Datentypen ersetzt werden. Im Beispiel oben würden z.B. alle Parameter und der Rückgabetyp als *double* oder *long* betrachtet werden. Schwierigkeiten ergeben sich allerdings, wenn man versucht, unterschiedliche Datentypen zu mischen, denn im Beispiel wurde ja allen Parametern und dem Rückgabewert den gleichen Datentyp zugewiesen:

```
#include <iostream>
#include <iomanip>

using namespace std;

template <class T> T MAX (T x, T y)
{
   return (x > y) ? x : y;
};

void main (void)
{
   cout << MAX (100, 111) << endl;    // Ergebnis 111
   cout << MAX (1.3, 1.1) << endl;    // Ergebnis 1.3
```

```
   cout << MAX (1.3, 2)   << endl;   // FEHLER !
}
```

Das Problem lässt sich scheinbar beheben, wenn man ein *template* mit zwei Datentypen verwendet:

```
#include <iostream>
#include <iomanip>

using namespace std;

template <class T, class U> T MAX (T x, U y)
{
   return (x > y ? x : y);
};

void main (void)
{
   cout << MAX (100, 111) << endl;    // Ausgabe 111
   cout << MAX (1.3, 1.1) << endl;    // Ausgabe 1.3
   cout << MAX (1.3, 2)   << endl;    // Ausgabe 2
   cout << MAX (2.3, 2)   << endl;    // Ausgabe 2.3
   cout << MAX (2, 1.3)   << endl;    // Ausgabe 2
   cout << MAX (1, 2.3)   << endl;    // Ausgabe 2 - FEHLER
}
```

Wie man sieht, ist die letzte Ausgabe inhaltlich falsch, da das Ergebnis eigentlich 2.3 sein sollte. Dies liegt daran, dass der Rückgabetyp <*class T*> ist und davon abhängt, welchen Typ man als ersten Parameter übergeben hat. Der naheliegende Gedanke einfach drei *templates* zu verwenden ist aber nicht möglich, da der Compiler keinerlei Anhaltspunkt hat, welchen Datentyp er für den Rückgabewert verwenden soll und daher einen Fehler meldet:

```
#include <iostream>
#include <iomanip>

using namespace std;

template <class T, class U, class V> T MAX (U x, V y)
{
   return (x > y ? x : y);
};

void main (void)
{
}
```

Einzige Möglichkeit dieses Problem zu lösen, ist einen möglichst großen Datentyp als Rückgabewert anzugeben (also üblicherweise *double* oder

long double). Dies schränkt die erhoffte Flexibilität durch die Verwendung des *template* jedoch deutlich ein:

```
template < class U, class V> double MAX (U x, V y)
{
   return (x > y ? x : y);
};
```

14. TYPATTRIBUTE

Die Vergabe von Typattributen ist eine weitere Möglichkeit, unter C/C++ auf die Eigenschaften einer Variablen Einfluss zu nehmen. Dazu ist es hilfreich, sich zu vergegenwärtigen, welche Eigenschaften das Verhalten einer Variablen bestimmen (siehe Abbildung).

Einige der aufgezeigten Variableneigenschaften (Name, Adresse, Wert und Typ) sollten aus den Zusammenhängen, die in den vorangegangenen Kapiteln vermittelt wurden, bekannt sein.

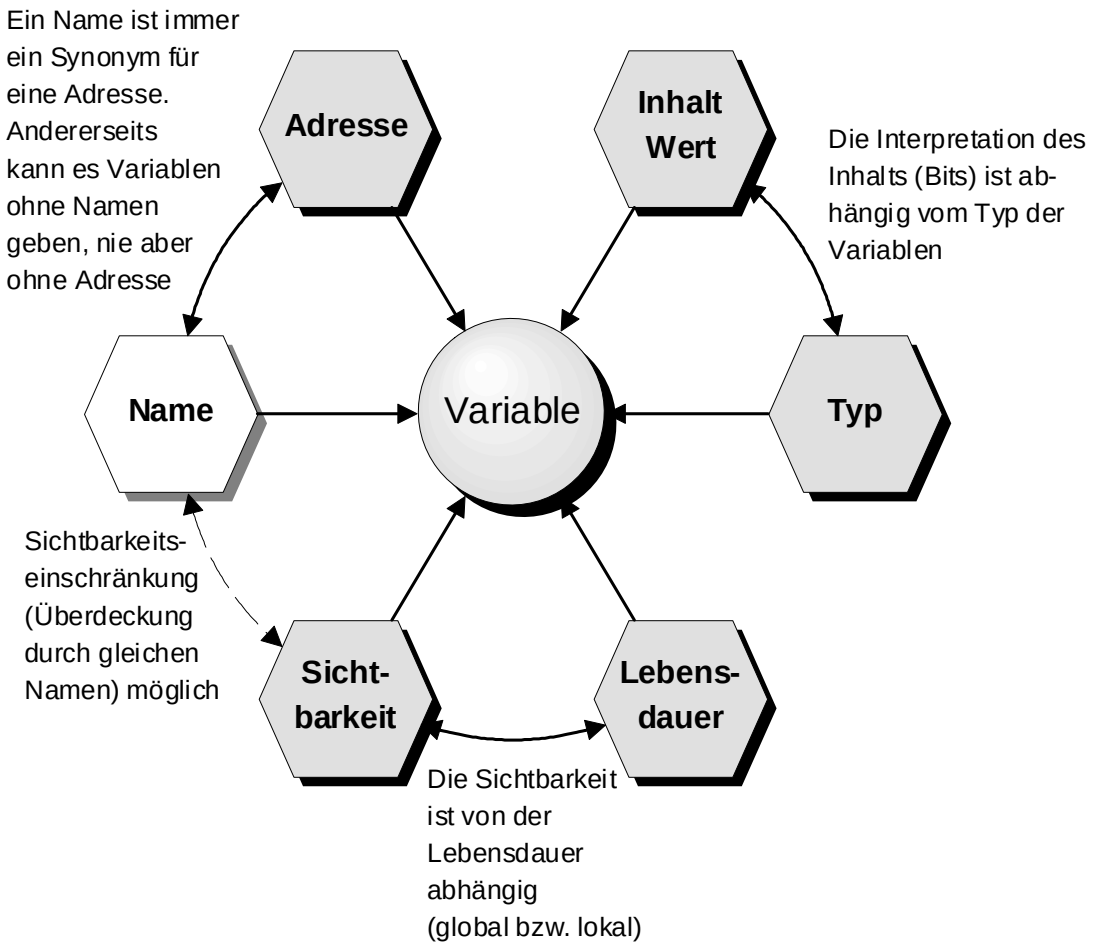


Abbildung 14-1 – Typattribute

- Name: der Name einer Variablen ist immer nur ein Synonym für ihre Adresse. Die Notwendigkeit eine Adresse mit einem Namen zu versehen ergibt sich allein aus der Tatsache, dass komplexe Programme nicht zu verstehen sind, wenn die einzelnen Variablenadressen nicht mit sinnvollen Namen, die ihre Aufgabe und ihren Inhalt repräsentieren, versehen sind.

- Adresse: jede Variable muss irgendwo im Speicher abgelegt werden und benötigt eine bestimmte Menge an Platz (Bits). Dieser Ort ist die physikalische Anfangsadresse der Variablen im Adressraum des Computers. Die Adresse einer Variablen ändert sich während des Programmlaufs nicht.

- Inhalt: der Inhalt oder Wert einer Variablen wird durch ein Bitmuster dargestellt, welches an der Adresse der Variablen gespeichert ist. Die Länge des Bitmusters ist abhängig vom Typ der Variablen.

- Typ: der Typ ist die Vorschrift an den Rechner, wie er den Inhalt der Variablen zu interpretieren hat. Das gleiche Bitmuster kann in C/C++ z.B. die Zahl 65 beziffern aber auch den Buchstaben ‚A' repräsentieren. Die Interpretation durch den Typ kann, bei geeignetem Bitmuster und bekannten Überleitungsvorschriften zur Laufzeit verändert werden (siehe Kapitel über Casting).

- Lebensdauer: Die Lebensdauer bezeichnet den Zeitraum zwischen Reservierung einer Adresse für ein Programm und der Freigabe dieses Speicherstücks für andere Zwecke. Je nach Art (dynamisch oder statisch) und Ort der Vereinbarung (global oder lokal) ist die Lebensdauer einer Variablen auf den Programmlauf oder den Lauf eines Unterprogramms (Funktion) beschränkt. Hält die Lebensdauer über das Ende des Programmlaufes an (nur bei dynamischen Variablen möglich), so spricht man von einem Speicherleck (Memory-Leakage).

- Sichtbarkeit: Die Sichtbarkeit ist eng verzahnt mit der Lebensdauer. Sie bezeichnet die Stellen im Programm, in der die Variable nicht nur existiert, sondern auch im Zugriff ist. Der Zugriff auf eine globale Variable kann z.B. durch eine lokale Variable gleichen Namens verhindert werden (Überdeckung).

C/C++ kennt insgesamt fünf verschiedene Arten von Typattributen, mit denen für alle Datentypen bestimmt werden kann, auf welche Weise der zugewiesene Speicherplatz verwaltet und der Zugriff auf die Variablen vorgenommen werden soll. Diese Attribute sind: *auto*, *const*, *register*, *extern* und *static*.

14.1. DAS TYPATTRIBUT AUTO

Alle lokalen Variablen sind vom Typ *auto*, sofern nicht explizit ein anderes Typattribut angegeben wird. Sie haben die Eigenschaft, dass die Variablen dieses Attributs nach Beendigung der Abarbeitung der Funktion wieder aus dem Speicher entfernt werden.

Wenn Sie Programme schreiben wollen, die auf unterschiedlichen Betriebssystemen laufen, dann sollten Sie immer vollständige Variablendeklarationen angeben. Schreiben Sie also auch die Angaben, die auf dem Entwicklungssytem voreingestellt sind:

auto unsigned long int TestVariable = 200;

Sie können dann sicher sein, dass Portierungsfehler in der Variablendeklaration, beim Übersetzen auf unterschiedlicher Hardware, auf ein Minimum reduziert werden.

Manche Compiler lassen u.a. auch Änderungen der Voreinstellungen für den Standard-Datentyp zu. Solche Änderungen können z.B. dann zu Fehlern führen, wenn Sie als Voreinstellung ihres Compilers den Datentyp *unsigned int* gewählt haben, auf einem anderen Rechner aber *int* (also *signed int*) verwendet wird.

Da es sich bei diesem Typattribut um die Voreinstellung handelt, die ohnehin vom Compiler vorausgesetzt wird (sofern keine abweichenden Angaben gemacht werden), wird das Schlüsselwort nur sehr selten verwendet. Auch alle globalen Variablen sind (rein syntaktisch gesehen) vom Attribut *auto* - allerdings mit einer Gültigkeitsbeschränkung, die „zufällig“ genau der Dauer des gesamten Programmlaufes entspricht, weshalb die Angabe des Schlüsselwortes unterbleibt (häufig sogar vom Compiler als Fehler gemeldet wird).

```
#include <iostream>
#include <iomanip>

using namespace std;

void fnAutoBeispiel (void)
{
   auto signed int Test = 7;

   cout << Test << endl;
}

void main (void)
{
   fnAutoBeispiel ();
}
```

14.2. DAS TYPATTRIBUT REGISTER

Das Attribut *register* ist im Grunde genommen nur eine Variante des Attributs *auto*. Der wesentliche Unterschied besteht darin, dass der Compiler versucht, die entsprechende Variable bevorzugt in einem freien Prozessorregister unterzubringen. Das Typattribut wird nur dann einen

Effekt, wenn auch ein Register zur Verfügung steht, welches für die Gültigkeitsdauer der Variable unbenutzt ist und somit uneingeschränkt zur Verfügung steht. In einem Prozessorregister ist der Zugriff besonders schnell, da der Wert nicht erst im Speicher angefordert und über den Engpass Systembus in den Prozessor (CPU) gebracht werden muss.

```
#include <iostream>
#include <iomanip>

using namespace std;

void fnRegisterBeispiel (void)
{
   register signed int i = 0;
   for (i=0; i<1000; i++)
   {
      cout << i << endl;
   }
}

void main (void)
{
   fnRegisterBeispiel ();
}
```

Ist kein Register frei, so wird die *register*-Variable wie eine normale *auto*-Variable behandelt und im Speicher abgelegt. Die Angabe der Schlüsselwortes *register* lohnt sich nur bei Variablen, die häufig aufgerufen werden (wie z.B. eine Zählvariable in einer *for*-Schleife). Mit der zunehmenden Verbreitung von Hardware-Caches im Prozessor (die letztlich genau das Gleiche leisten) verringert sich auch der Geschwindigkeitsgewinn, der sich durch die Verwendung von *register*-Variablen erzielen lässt.

14.3. DAS TYPATTRIBUT STATIC

Das Attribut *static* veranlasst den Compiler dazu, den Wert einer lokalen Variablen auch nach Beendigung des Funktionsblockes im Speicher zu behalten. Außerhalb der Funktion kann auf die *static*-Variable allerdings nicht zugegriffen werden, da ihr Name und ihre Adresse nur innerhalb der Funktion bekannt (gültig) sind, in der sie deklariert wurde (wäre es anders, bestünde kein Unterschied zu einer globalen Variable).

Die Lebensdauer der Variablen wird also auf den Programmlauf ausgedehnt, ihre Sichtbarkeit bleibt jedoch auf ihren Funktionsblock beschränkt.

```
//=============================================================
// Programm LOCALSTA.CPP
//=============================================================

#include <iostream>
#include <iomanip>

using namespace std;

void fnStaticBeispiel (void)
{
   static signed int WieOft = 0;

   WieOft++;
   cout << WieOft << endl;
}

void main (void)
{
   fnStaticBeispiel ();
   fnStaticBeispiel ();
   fnStaticBeispiel ();
}
```

Bei einem erneuten Funktionsaufruf sind jedoch die zuvor erarbeiteten Werte in den *static*-Variablen noch erhalten, was sonst nur durch die Verwendung einer globalen Variablen möglich wäre. Wird eine *static*-Variable initialisiert (wie im Beispiel oben), so wird die Initialisierungszuweisung nur beim allerersten Funktionsaufruf durchgeführt. Das Typattribut *static* kann nur in Unterprogrammen (Funktionen) verwendet werden.

Im Zusammenhang mit Klassendeklarationen in C++ kommt dem Attribut *static* eine zusätzliche, etwas andere Bedeutung hinzu, die im Abschnitt über Klassen ausführlich behandelt wird.

14.4. DAS TYPATTRIBUT EXTERN

Das Attribut *extern* hat seine Bedeutung im Zusammenspiel verschiedener Module in größeren Programmsystemen. Es weist den Compiler an, für eine Variable keinen neuen Speicher zu belegen, sondern beim Binden der Programmteile (Linking) die Variable aus dem globalen Deklarationsteil eines anderen Moduls zu übernehmen.

Die Sichtbarkeit einer Variablen wird also auf ein weiteres Programm-Modul ausgedehnt, ihre Lebensdauer ist ohnehin der Programmlauf, da es sich um eine globale Variable handeln muss.

```
//=============================================================
// Programm DATEIA.CPP
//=============================================================

//-------------------------------------------------------------
// Benötigt Projekt mit DATEIA.CPP und DATEIB.CPP
//-------------------------------------------------------------

#include <iostream>
#include <iomanip>

using namespace std;

void fnExternBeispiel (void); // Prototype

int AusA = 6;

void main (void)
{
   fnExternBeispiel ();
}
```

```
//=============================================================
// Programm DATEIB.CPP
//=============================================================

//-------------------------------------------------------------
// Benötigt Projekt mit DATEIA.CPP und DATEIB.CPP
//-------------------------------------------------------------

#include <iostream>
#include <iomanip>

using namespace std;

void fnExternBeispiel (void)
{
   extern int AusA;

   cout << AusA << endl;
}
```

Die Deklaration mit *extern* unterstützt die Methodik des modularen Programmierens, bei der einzelne Programmteile separat erstellt und kompiliert werden. Erst beim Linking werden die einzelnen Teile zu einem Gesamtprogramm zusammengefasst. Das Attribut *extern* verweist immer auf eine Variable gleichen Namens in einem anderen Modul, die im folgenden Programmteil (Funktion) benutzt werden soll. Die *extern*-Variable muss in genau einem der Module global definiert sein. Die Deklaration mit *extern* muss in jeder Funktion, in der diese Variable verwendet werden soll, wiederholt werden - da sonst natürlich eine lokale

Variable gleichen Namens erzeugt wird (eine globale *extern*-Deklaration ist in einigen Compilern nicht möglich).

14.5. DAS TYPATTRIBUT CONST

Die Deklaration mit *const* erzeugt eine konstante Variable, d.h. der Inhalt dieser Variablen darf nach der Initialisierungszuweisung nicht mehr verändert werden. Die Verwendung als Ziel einer Zuweisung erzeugt automatisch einen Fehler bei der Übersetzung des Programms:

```
const double MWST = 0.15;

void main (void)
{
   double Preis = 100.0;

   Preis = Preis * MWST + Preis;  // korrekt
   MWST = 0.20;                   // Fehler, wird verhindert
}
```

Der Vorteil von Konstanten gegenüber der #*define*-Anweisung ist, dass der Inhalt der Variablen nur ein einziges Mal im Speicher vorhanden ist.

Dies ist insbesondere bei langen Textkonstanten, die an mehreren Stellen verwendet werden, wichtig, da eine #*define*-Konstante einen Text natürlich mehrfach im Programm ablegt.

Textkonstanten sollten niemals als #define deklariert werden, sondern immer als Konstante vom Typattribut const. Die Deklaration über das Typattribut spart erheblich globalen Speicherplatz, da die Texte nicht mehrfach im Programm abgelegt werden.

15. FELDER / ARRAYS

Felder werden in C/C++ durch Angabe des Feldtyps, eines Namens und der Anzahl der im Feld enthaltenen Elemente definiert. Die Dimensionierung erfolgt in eckigen Klammern. Alle Feldindizes werden grundsätzlich von Null an gezählt, so dass eine Bereichsangabe für den Index nicht möglich ist (anders als z.B. in PASCAL, wo Start- und Endwert des Index angegeben werden).

```
//-----------------------------------------------------
// eindimensionales Feld, 66 long, Bereich 0-65
//-----------------------------------------------------
long array [66];

//-----------------------------------------------------
// zweidimen. Feld, 12 double, Bereich 0-2 x 0-3
//-----------------------------------------------------
double ffeld [3][4];

//-----------------------------------------------------
// dreidim. Feld, 729 float, Bereich 0-8 x 0-8 x 0-8
//-----------------------------------------------------
float mtx [9][9][9];
```

15.1. EINDIMENSIONALE ARRAYS

Bei jeder Felddefinition ist daran zu denken, dass die Indexzählung immer bei Null beginnt. Der höchste legale Index liegt also um den Wert Eins niedriger, als die Angabe in der Klammer bei der Deklaration. Der Zugriff auf ein einzelnes Feldelement erfolgt über die Angabe des entsprechenden Index, wobei als Index jede ganzzahlige Konstante, jeder ganzzahlige Ausdruck oder Variableninhalt erlaubt ist:

```
//============================================================
// Programm ARRAY1.CPP
//============================================================

#include <float.h>
#include <iostream>
#include <iomanip>

using namespace std;

long array [66];
double ffeld [3][4];

void main (void)
{
   array [65] = 1;              // identisch mit: array['A']=1;
   cout << array [65]  << endl;
   cout << array ['A'] << endl;
```

```
   array ['A'] = 2;            // identisch mit: array[65]=2;
   cout << array [65]  << endl;
   cout << array ['A'] << endl;

   ffeld [array[64+1]][3] = 100.0 * 17.6;
   cout << ffeld [array[65]][3] << endl;
}
```

Das folgende Beispiel zeigt einen Zugriff auf einen illegalen (nicht im vereinbarten Bereich liegenden) Indexzugriff. Der Compiler hat zum Zeitpunkt der Übersetzung keine Möglichkeit Fehler dieser Art zu unterbinden.

```
#include <float.h>
#include <iostream>
#include <iomanip>

using namespace std;

float mtx [9][9][9];

void main (void)
{
   mtx [3+4-1][4+4+4][2*2] = 5.5 * 5.5;   // illegaler Index
   cout << ffeld mtx [6][12][4] << endl;  // illegaler Index
}
```

Das erste Beispielprogramm weist dem letzten Feldelement des Feldes *array* den Wert Eins zu. Die zweite Indexangabe im Beispiel ist syntaktisch identisch mit dem ersten, da zur Übersetzungszeit (Kompilierung) alle Buchstabenkonstanten in ASCII-Zahlenwerte umgewandelt werden (In C/C++ ist *char* ein ganzzahliger Typ, ähnlich wie z.B. *unsigned short*). Der ASCII-Code von ‚A' ist 65. Im dritten Beispiel wird der Feldinhalt von *array [65]* als Index für die erste Dimension des Feldes *ffeld* ausgelesen, hat also den Wert 1 (im Beispiel oben).

Das letzte Beispiel zeigt eine unangenehme Eigenschaft (und häufige Fehlerquelle) von C/C++. Der Zugriff auf die Feldvariable *mtx [6][12][4]* ist nach der Vereinbarung von *mtx* (mit den Dimensionsgrößen 9 x9 x 9) offensichtlich illegal. Dennoch führt der Computer den Zugriff aus (ohne Fehlermeldung oder Warnung), da C/C++ keinerlei Indexüberprüfung zur Lauf- oder Ubersetzungszeit durchführt. Der zugewiesene Wert wird an den errechneten Speicherplatz geschrieben und zerstört dort höchstwahrscheinlich wichtige Daten.

Der Zugriff auf jede Art von Feldern erfolgt in C/C++ grundsätzlich durch Pointer, d.h. der Computer merkt sich nur die Anfangsadresse des ersten Feldelementes und errechnet die Position des angegebenen Elementes

aus den aufgeführten Indizes. Daraus ergibt sich, dass die folgenden Ausdrücke syntaktisch gleichwertig sind:

```
array [4]  = 7;
*(array+4) = 7;
```

Im ersten Fall wird auf das Datenelement mit dem Index Vier zugegriffen (also auf das fünfte Datenelement). Im zweiten Beispiel wird über den Pointer zugegriffen, dabei wird zunächst der Pointer *array* ausgewertet, der die Adresse des ersten Feldelementes darstellt. Da der Pointer auf Integer (zwei Byte Länge) zeigt, wird auf die enthaltene Adresse viermal die Länge eines *int* aufaddiert (Adressenberechnung), also insgesamt acht Byte. Bei einer Initialisierung mit dem Wert Null kann, für die oben erfolgten Anweisungen, von einem Speicherbild ausgegangen werden, wie es in den nachstehenden Tabellen dargestellt ist. Der Sternoperator erlaubt dann den Zugriff auf den Inhalt der Variablen und somit die korrekte Zuweisung.

Die folgende Tabelle legt die Deklaration `short int sarray[66];` zugrunde.

Aufbau eines eindimensionalen Feldes			
Adresse	Wert / Inhalt	Namen	*Datentyp*
keine, nur logischer Name	1000	sarray	Pointer auf short int
1000	0	sarray [0]	short int
1002	0	sarray [1]	short int
1004	0	sarray [2]	short int
1006	0	sarray [3]	short int
1008	0	sarray [4]	short int
1010	0	sarray [5]	short int
...	...	...	short int
1128	0	sarray [64]	short int
1130	1	sarray [65]	short int

Tabelle 15-1: Aufbau eines eindimensionalen Feldes

15.2. ZWEIDIMENSIONALE ARRAYS

Da, wie bereits erwähnt, eine enge Verwandtschaft zwischen Feldern und Zeigern besteht, lässt sich diese auch zum eigenen Vorteil ausnutzen. Man kann dies z.B. bei mehrdimensionalen Arrays tun, die in allen Programmiersprachen üblicherweise von statischer Länge und rechteckig sind. Im Normalfall ist gilt dies auch für C/C++:

```
int twodimarray [3][4];
```

Das dabei erzeugte Feld lässt sich leicht als rechteckiger Block von Variablen auffassen und darstellen:

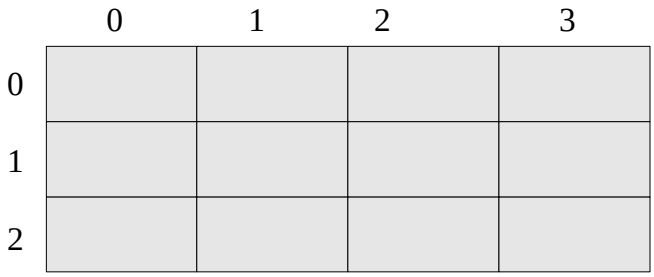


Abbildung 15-1: zweidimensionales Feld konstanter Länge

Wie bereits oben geschildert, werden Pointer auf Integer gebildet, die auf jeweils vier Feldelemente verweisen. Der Name selbst (hier *twodimarray*) entspricht einem Pointer auf Pointer auf Integer. Mit relativ wenig Aufwand lassen sich nun in C/C++ auch Felder unterschiedlicher Länge zu mehrdimensionalen Arrays zusammenfassen:

```
int first  [4];
int second [3];
int third  [5];
int *twodim [3] = { first, second, third };
```

Das Feld *twodim* speichert nun nicht mehr Integerwerte, sondern Pointer auf Integer. Die oben verwendete Graphik würde, auf dieses Beispiel übertragen, so aussehen:

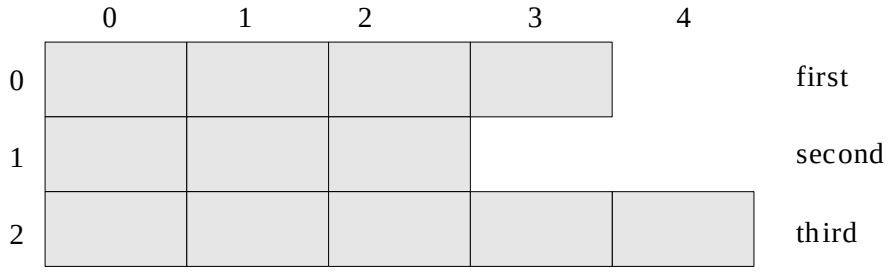


Abbildung 15-2: zweidimensionales Feld variabler Länge

Da der Aufbau dieses Feldes formal mit dem eines normalen, zweidimensionalen Feldes identisch ist, können auch die einzelnen Variablen ganz normal angesprochen werden:

```
int first  [4];
int second [3];
int third  [5];
int *twodim [3] = { first, second, third };

void main (void)
{
   twodim [0][0] = 3;
   twodim [1][0] = 2;
```

```
   twodim [2][0] = 4;
   twodim [2][4] = 7;
}
```

Für die Einhaltung der Arraygrenzen ist, wiederum, allein der Programmierer verantwortlich. Bei Feldern mit variabler zweiter Dimension hat es sich daher als sehr praktisch erwiesen, die Länge des jeweiligen Teilstücks in die erste Speicherstelle zu schreiben:

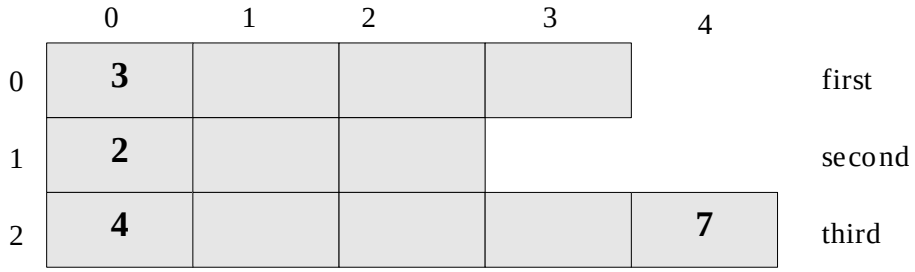


Abbildung 15-3: zweidimensionales Feld mit Größenangabe

Eine solche Vorgehensweise erlaubt es dem Programmierer außerdem, an einer definierten Speicherstelle die Länge des Arrays abzufragen und eine Überprüfung der Legalität des Zugriffs durchzuführen.

Aufbau eines zweidimensionalen Feldes			
Adresse	Wert / Inhalt	Namen	*Datentyp*
keine, nur logischer Name	2000	ffeld	Pointer auf Pointer auf double
keine, nur logischer Name	2000	ffeld [0]	Pointer auf double
keine, nur logischer Name	2016	ffeld [1]	Pointer auf double
keine, nur logischer Name	2032	ffeld [2]	Pointer auf double
2000	0.0	ffeld [0][0]	double
2004	0.0	ffeld [0][1]	double
2008	0.0	ffeld [0][2]	double
2012	0.0	ffeld [0][3]	double
2016	0.0	ffeld [1][0]	double
2020	0.0	ffeld [1][1]	double
2024	0.0	ffeld [1][2]	double
2028	1760.0	ffeld [1][3]	double
...	...		
2044	0.0	ffeld [2][3]	double

Tabelle 15-2: Aufbau eines zweidimensionalen Feldes

15.3. ARRAY-INITIALISIERUNG

Wie einfache Variable auch, können Felder schon bei der Deklaration mit Werten initialisiert werden. Die Angabe der Werte muss (für jede

angegebene Dimension) mit geschweiften Klammern zu einem Block zusammengefasst werden:

```
int a1 [5] = {0,1,2,3,4};

//-------------------------------------------------------------
// Zuweisung erlaubt, da a1 [2] schon bekannt ist
//-------------------------------------------------------------
int a2 [2][3] = {
                   { 0,  1, a1[2]},
                   {10, 11,    12}
                };

int b [][][] =
   {{{100,101,102,103},{110,111,112,113},{120,121,122,123}},
   {{200,201,202,203},{210,211,212,213},{220,221,222,223}}};

//-------------------------------------------------------------
// identisch zu b, aber besser lesbar
//-------------------------------------------------------------
 int a3 [2][3][4] = {
                        {
                           {100,101,102,103},
                           {110,111,112,113},
                           {120,121,122,123}
                        },
                        {
                           {200,201,202,203},
                           {210,211,212,213},
                           {220,221,222,223}
                        }
                     };

void main (void)
{
}
```

Die Inhalte der Beispiele *a3* und *b* sind identisch. Das Beispiel *b* zeigt lediglich eine kompaktere Schreibweise. Die Klammern nach *b* können leer bleiben, da die Anzahl der Elemente pro Dimension direkt abgeleitet werden kann. Der Rechner ermittelt die Anzahl der aufgeführten Elemente und trägt diese als Dimensionierung automatisch ein. Wie bereits aus den Beispielen ersichtlich ist, sollte die Initialisierung immer vollständig erfolgen, eine Teilinitialisierung ist zwar möglich, aber unübersichtlich und unüblich.

15.4. POINTER / ZEIGER / ADRESSEN

Während in den meisten strukturierten Programmiersprachen fast ausschließlich mit Werten und nur im Ausnahmefall mit Pointern gearbeitet wird, sind Zeiger in C/C++ unumgänglich. Zu jeder deklarierten Variablen kann man ohne besonderen Aufwand die Adresse

ermitteln, bzw. zu jeder Adresse den zugehörigen Inhalt der Speicherstelle. Die nachstehende Deklaration erzeugt im Speicher den darunter abgebildeten Zustand (Adressen sind fiktiv)

```
int  x  = 7;
int *px = &x;
```

Variablen Name	Adresse	Adresse entspricht	Wert	Wert entspricht
x	10000	&x oder px	7	x oder *px
px	10002	&px	10000	px

Die Handhabung von Zeigern in C/C++ ist ebenso einfach wie anfällig für Fehler. Aber gerade der ausgiebige Gebrauch von Pointern ist eine der wichtigsten Grundlagen für den Geschwindigkeitsvorteil, den C bietet. Ein Pointer wird immer einem bestimmten Typ zugeordnet (es gibt also keine Zeiger an sich, sondern z.B. Zeiger auf *long* oder Zeiger auf *float*). Da komplexen Datenstrukturen, wie z.B. die Felder, ebenfalls nur über Zeiger erreichbar sind, kann durch einfache Indizierung oder Addition das Folgeelement ermittelt werden:

```
int i;           // einfache Integervariable
int *ip;         // Zeiger auf einen Integerwert
int f ();        // Funktion, die Int.wert als
                 // Ergebnis liefert
int *fip ();     // Funktion, die Zeiger auf Integer
                 // als Ergebnis liefert
int (*pfi) ();   // Zeiger auf Funktion, welche einen
                 // Integerwert als Funktionsergebnis
                 // liefert
int ai [4];      // Feld, entspricht einem int-Zeiger
                 // auf das 1. Element des Feldes
int *aip [4];    // Feld mit 4 Zeigern auf INT, aip
                 // selbst ist ein Zeiger auf den
                 // ersten Zeiger auf INT
```

Dabei entspricht die Deklaration **fip ()* syntaktisch der Deklaration **(fip ())*. Der Zugriff auf Werte (mit Hilfe des Pointers) ist genauso einfach und wird ebenfalls über den „*“-Operator bewerkstelligt. Wie schon mehrfach betont, erfolgt die Vereinbarung eines Feldes immer über einen Zeiger auf das erste Element.

Für die letzten beiden der oben angeführten Beispiele bedeutet dies, dass *ai* (ohne Angabe eines Index) automatisch ein Pointer auf Integer ist - und *aip* ein Pointer auf einen Pointer auf Integer.

```
//============================================================
// Programm POINTER1.CPP
//============================================================

#include <stdlib.h>
#include <iostream>
#include <iomanip>

using namespace std;

//------------------------------------------------------------
// globale Variablen
//------------------------------------------------------------
int *ptr     = NULL;              // Pointer auf Integer
int  x       = 6;                 // Integervariable
int  ai  [4] = {11, 12, 13, 14};  // Feld von Integer-Zahlen
int *aip [4] = {NULL, NULL, NULL, NULL};
                                  // Feld von Zeigern auf INT

//------------------------------------------------------------
// Ausgabefunktion
//------------------------------------------------------------
void fnAusgabe (void)
{
   static int i = 1;              // Zähler für Ausgabezeile
   cout << i << ". x: " << setw(2) << x << " ptr: " << ptr
        << " *ptr: " << setw (2) << *ptr << endl;
   i++;
}

//------------------------------------------------------------
// Hauptprogramm
//------------------------------------------------------------
void main (void)
{
   ptr = &x;      // Zuweisung der Adresse von x, lies: setze
                  // den Zeiger auf Adresse der INT-Var. x
   fnAusgabe();   // Ergebnisausgabe

   *ptr = 7;      // Zuweisung von 7 an x, lies: setze Inhalt
                  // der Variablen, auf die ptr zeigt auf 7
   fnAusgabe();   // Ergebnisausgabe

   ptr = ai;      // Erlaubt, da beides Pointer auf Integer
   fnAusgabe();   // Ergebnisausgabe

   aip[1] = &ai[1];
   ptr = aip [1]; // Erlaubt, aip enthält als Elemente
                  // INT-Zeiger
   fnAusgabe();   // Ergebnisausgabe

   aip[0] = &x;   // Pointer-Initialisierung (vorher NULL)
   ptr = *aip;    // Erlaubt, *aip entspricht aip [0]
   fnAusgabe();   // Ergebnisausgabe

   x = ai [3];    // Inhalt von ai [3] nach x
```

```
   fnAusgabe();   // Ergebnisausgabe

   aip[2]=&ai[2]; // Pointer-Initialisierung (vorher NULL)
   x = *aip [2];  // INT-Wert, auf den Pointer aip [2] weist
   fnAusgabe();   // Ergebnisausgabe

   x = *(*aip);   // Erlaubt, entspricht *aip [0]
   fnAusgabe();   // Ergebnisausgabe
}
```

Aus den bisher gezeigten Beispielen lassen sich folgende Grundstrukturen für die Arbeit mit Adressen (Zeigern) ableiten:

```
*pointer_var;  // Greift auf den Inhalt der
               // Speicherstelle zu, auf die der
               // Pointer zeigt.
&variable;     // Ermittelt die Adresse einer
               // Variablen im Speicher.
```

Dieses Zugriffsprinzip lässt sich im Grunde beliebig erweitern. So definiert z.B. die folgende Anweisung einen Pointer auf einen Pointer auf Integer:

```
int *(*pptr);  // Pointer auf einen Pointer auf
               // Integer
pptr = &ptr;   // Zuweisung der Adresse des Int-
               // Pointers
```

Wie man sieht, hat jedes Objekt eine Adresse, auf die der Programmierer direkt zugreifen kann. Dies eröffnet vielfältige Manipulationsmöglichkeiten, insbesondere im Zusammenhang mit Feldern und anderen komplexen Datenstrukturen.

Bei mehrdimensionalen Arrays ist nicht nur ein Zugriff auf die einzelnen Datenelemente möglich, sondern auch die Verarbeitung und Manipulation der Speicheradressen:

```
//============================================================
// Programm POINTER2.CPP
//============================================================

#include <stdlib.h>
#include <iostream>
#include <iomanip>

using namespace std;

//------------------------------------------------------------
// globale Variable
//------------------------------------------------------------
int x3d [3][3][3] =                        // 3D-Feld
```

```
   {
     { {111, 112, 113}, {121, 122, 123}, {131, 132, 133} },
     { {211, 212, 213}, {221, 222, 223}, {231, 232, 233} },
     { {311, 312, 313}, {321, 322, 323}, {331, 332, 333} },
   };

int *ip = NULL;       // Pointer auf Integer
int *(*pip) = NULL;   // Pointer auf INT-Pointer

//------------------------------------------------------------
// Ausgabefunktion
//------------------------------------------------------------
void fnAusgabe (void)
{
   static int i = 0;
   cout << i << ". ip: " << setw(3) << ip << " *ip: "
        << setw(3) << *ip
        << " x3d [0][0][0] - x3d [0][0][2]: " << setw(3)
        << x3d [0][0][0] << " | " << setw(3) << x3d [0][0][1]
        << " | " << setw(3) << x3d [0][0][2] << endl;
   i++;
}

//------------------------------------------------------------
// Hauptprogramm
//------------------------------------------------------------
void main (void)
{
   ip = &(x3d [1][1][1]);  // Zeiger auf Datenelement
   fnAusgabe ();

   ip = x3d [0][0];        // Zeiger auf Datenelement
   fnAusgabe ();

   *(ip+2) = 7;            // Drittes Datenelement = 7
   ip [2] = 7;             // Drittes Datenelement = 7
   fnAusgabe ();

   ip  = &(x3d [0][0][0]); // Zeiger auf 1. Element
   pip = &ip;              // Zeiger auf Zeiger auf 1. Element
   *(*pip) = 3;            // Erstes Element = 3
   fnAusgabe ();
}
```

Im folgenden Deklarationsbeispiel ist noch einmal angegeben, welche Bedeutung sich ergibt, wenn Dimensionsindizes ausgelassen werden. Die Deklaration von *y3d* ist hier ein Array mit drei Elementen, jedes Element für sich verweist auf ein Feld von fünf Zeigern. Die Zeiger wiederum verweisen auf ein Feld mit jeweils sieben Integerwerten:

```
int y3d [3][5][7]  // Deklaration

y3d                // Startadresse des Gesamtfeldes
```

```
                    // kann einem Zeiger auf Zeiger
                    // auf Zeiger auf INT zugewiesen
                    // werden

y3d [i]             // kann einem Zeiger auf Zeiger
                    // auf INT zugewiesen werden

y3d [i][j]          // kann einem Zeiger auf INT
                    // zugewiesen werden

y3d [i][j][k]       // kann einem INT zugewiesen
                    // werden
```

Der Unterschied zwischen einem mehrdimensionalen Feld und einem Feld von Pointern auf Arrays scheint gering zu sein. Dennoch gibt es kleine, aber entscheidende Unterschiede zwischen beiden Formen. So muss einem Zeiger erst ein Element zugewiesen werden, bei einem Array hingegen wird der benötigte Speicherplatz direkt reserviert.

Ein entscheidender Vorteil ist jedoch, dass Felder, auf welche von (separat definierten) Pointern aus verwiesen wird, nicht alle die gleiche Länge haben müssen (s.o.). Stattdessen können sie auf Arrays unterschiedlichsten Umfangs verweisen. Für ein zweidimensionales Feld in dem einmal 20 und einmal 50 Zahlen zu speichern sind, sieht der Unterschied wie folgt aus:

```
int *ip [2];   // Array mit 2 Zeigern auf Integer
int f1 [20];   // Array mit 20 Integer-Zahlen
int f2 [50];   // Array mit 50 Integer-Zahlen
int a [2][50]; // 2D-Array,althergebrachte Weise
ip [0] = f1;   // Feldanfang an Pointerfeld zuweisen
ip [1] = f2;   // Feldanfang an Pointerfeld zuweisen
ip [0][0] = 7; // Wertzuweisung an das erste Element
a [0][0] = 7;  // Wertzuweisung an das erste Element
```

Wie leicht zu sehen ist, können beide Formen tatsächlich synonym zueinander verwendet werden. Allerdings ist der Verwaltungsaufwand bei der Pointerlösung wesentlich höher (nicht zu vergessen, dass zudem die Fehlerwahrscheinlichkeit durch ungewollte Überschreitung des Feldgrenzen steigt). Dafür wird überflüssiger Speicherplatz gespart und das Programm bleibt insgesamt flexibler, da die Zuweisung der Felder (und damit ihre Größe oder Reihenfolge) leicht geändert werden kann.

15.5. POINTER-ARITHMETIK

Da man in C/C++ die Adressen von Objekten im Speicher leicht ermitteln kann, liegt es nahe, auch die Möglichkeiten zur direkten Manipulation solcher Adressen (bzw. der sich ergebenden Pointer) zu schaffen. Diese

Vorgehensweise wird insbesondere bei der Stringmanipulation gern und intensiv genutzt (siehe Abschnitt über Zeichenketten). Ein Vorteil liegt darin, dass die arithmetischen Operationen auf Pointern die Länge der Objekte auf die gezeigt wird, automatisch berücksichtigen. In C/C++ ist die Manipulation von Pointern durch eine ganze Reihe von Operatoren möglich, die der nachstehenden Liste zu entnehmen sind.

Pointer-Operatoren	
Operator	Bedeutung
=	Zuweisung zwischen zwei Pointern gleichen Typs, beide Zeiger verweisen anschließend auf das gleiche Objekt
+	Addition einer ganzzahligen Konstante oder eines Variablenwertes auf einen Pointer
+=	Addition einer ganzzahligen Konstante oder eines Variablenwertes auf einen Pointer
-	Subtraktion einer ganzzahligen Konstante oder eines Variablenwertes von einem Pointer. Oder Subtraktion zweier Pointer, die jedoch auf den gleichen Typ weisen müssen
-=	Subtraktion einer ganzzahligen Konstante oder eines Variablenwertes von einem Pointer
++	Inkrementieren eines Pointers (Entspricht der Addition der Bytelänge des Typs auf den der Pointer zeigt)
--	Dekrementieren eines Pointers (Entspricht der Subtraktion der Bytelänge des Typs auf den der Pointer zeigt)
==	Vergleich zweier Pointer, Test auf Gleichheit
!=	Vergleich zweier Pointer, Test auf Ungleichheit
<	Vergleich zweier Pointer, Test auf Kleiner
>	Vergleich zweier Pointer, Test auf Größer
<=	Vergleich zweier Pointer, Test auf Kleiner oder Gleich
>=	Vergleich zweier Pointer, Test auf Größer oder Gleich

Tabelle 15-3: Pointer-Operatoren

Man beachte, dass die direkte Addition zweier Pointer nicht erlaubt ist. Dennoch wäre diese durchaus realisierbar, indem man die Adresse des zweiten Pointers in einen *long*-Wert wandelt (wird von einem Compiler mit einem Warning quittiert) und anschließend als Variablenwert addiert. Allerdings ist diese Operation ohnehin sinnlos. Die Subtraktion hingegen ist erlaubt (bei Zeigern auf den gleichen Typ). Sie liefert einen *long*-Wert, der angibt, um wie viele Feldelemente die beiden Pointer auseinander liegen. Dies ergibt natürlich nur einen Sinn, wenn beide Zeiger auf Teile des gleichen Feldes verweisen.

Die Zuweisung einer Konstanten an einen Pointer (direkte Adressierung) ist grundsätzlich verboten!

```
int *ip;

void main (void)
{
   ip  = 2000;   // Verboten: Konstante als Adresse
}
```

```
int *ip;

void main (void)
{
   ip +=    2;   // Erlaubt
}
```

Das erste Beispiel erzeugt einen Fehler, da versucht wird dem Zeiger *ip* willkürlich einen Wert (und damit einen wahrscheinlich nicht reservierten Speicherbereich) zuzuweisen. Im Zeitalter von Multitasking-Rechnern, in deren Speicher mehrere Rechenprozesse (Programme) gleichzeitig laufen, ist eine solche feste Vorgabe ohnehin ein absolutes Tabu, da der Programmcode *relocable* (frei im Speicher verschiebbar) sein muss. Bei diesen Rechnern kommen ohnehin nur indirekte Adressierungsmethoden in Frage. Die zweite Anweisung hingegen ist völlig legal. Angenommen *ip* zeigt auf die Adresse 5000, dann wird der Zeiger um zwei Integerbreiten (= 4 Byte) verschoben und würde anschließend auf die Adresse 5004 verweisen. Wäre *ip* als Zeiger auf *double* deklariert worden (meist 8 Byte), so würde der Pointer anschließend auf die Adresse 5016 zeigen.

Eine stete Fehlerquelle bei der Pointer-Arithmetik sind natürlich die Vorrangregeln der Operatoren. Auch vielen geübten Programmierer unterlaufen hier immer wieder Fehler. Angenommen man hat einen Pointer auf das 10. Element eines Feldes aufgebaut und möchte, von diesem Pointer ausgehend, mittels Pointer-Arithmetik, auf das 5. Element zugreifen und es zum aktuellen Element hinzuaddieren:

```
*pointer += *pointer - 5;
*pointer += (*pointer) - 5;
```

Korrekt ist lediglich das folgende Beispiel, da die Priorität des Verweisoperators (*) höher ist als die der Subtraktion, wird bei einem ungeklammerten Ausdruck zuerst der Inhalt ermittelt und davon subtrahiert. Bei den Beispielen oben wird also der Inhalt von der aktuellen Speicherzelle um den Wert 5 verringert.

```
*pointer += *(pointer - 5);
```

15.6. DIE INITIALISIERUNG VON POINTERN

Pointer lassen sich in gleicher Weise initialisieren, wie andere Variable. Nur ist hierbei zu bedenken, dass Pointer nur über andere Variablen (auf die sie anschließend verweisen) Werte erhalten können. Diese Variablen müssen (logischerweise) bereits zuvor deklariert und gegebenenfalls initialisiert worden sein:

```
int x;
int *ptr    = &x;
int *(*ptr) = &ptr;
int *pt2    = ptr;
```

Auch Felder von Pointern können bei Bedarf initialisiert werden, was in Funktionen gelegentlich von Vorteil sein kann:

```
//-------------------------------------------------------------
// Funktion mit Return-Wert "Zeiger auf Char"
//-------------------------------------------------------------

int *Monatsname (int n)
{
   static char *Name [] = {"ERR", "Jan", "Feb", "Mär",
                           "Apr", "Mai", "Jun", "Jul",
                           "Aug", "Sep", "Okt", "Nov",
                           "Dez"};
   return ((n < 1 || n > 12) ? Name [0] : Name [n]);
}
```

15.7. HÄNGENDE REFERENZEN

Unter hängenden Referenzen versteht man Zeiger auf Variablenwerte, die nicht mehr existieren oder die nicht mehr reserviert sind. Der Zeiger weist dann im übertragenen Sinne ins „Nichts“. Da die entsprechenden Speicherbereiche nicht mehr mit sinnvollen Daten belegt sind, ist entweder ein inhaltlicher Fehler die Folge oder ein Systemabsturz.

Besonders kritisch sind hängende Referenzen wenn sie in einer indirekten Verweiskette auftreten, d.h. wenn ein Zeiger auf einen Zeiger verweist, der nicht mehr existiert. Soll nun ein Schreibzugriff auf den Wert am Ende der Kette erfolgen, so ist es durchaus möglich, dass in Bereiche geschrieben wird, die eigentlich dem Betriebssystem vorbehalten sind:

```
int *ip;       // globale Variable, Pointer

void abc ()    // Funktion
{
   int x;      // lokale Variable
   ip = &x;    // globale Variable verweist auf lokale
}              // hier entsteht die hängende Referenz
```

```
void main ()
{
   abc ();     // Funktionsaufruf
}
```

Hängende Referenzen entstehen besonders dann, wenn die Blockgrenzen von Funktionen (Lebensdauerbegrenzung von lokalen Variablen) nicht beachtet werden. Es ist daher strikt zu vermeiden, einen globalen Pointer auf eine lokale Variable zeigen zu lassen.

Interessanterweise kann die hängende Referenz vermieden werden, indem die Variable *x* in der Beispielfunktion als *static* deklariert wird, also auch nach Ende der Funktion den reservierten Speicherplatz behält. Der Inhalt der *static*-Variable ist über den Pointer anschließend auch vom Hauptprogramm aus zugänglich.

15.8. DER BUBBLESORT-ALGORITHMUS

Eine ausführliche Darstellung der Sortierverfahren Bubblesort und Quicksort in C und C++ ist in Kapitel 20 zu finden. Die Darstellung hier beschränkt sich auf die für die Aufgabenstellung notwendigen Informationen.

Der Bubblesort, auch iteratives Austauschen oder direkte Auswahl genannt, ist einer der einfachsten, zugleich aber auch einer der ineffizientesten Sortieralgorithmen. Dafür bietet er jedoch den Vorteil, nur sehr wenig Speicherplatz zu benötigen.

Bei kleinen Datenmengen ist die Anzahl der mittleren Bewegungen[5] (M) und der notwendigen Vergleiche (C) relativ uninteressant, sie beträgt beim Bubblesort:

$$C = (n^2 - n) / 2$$
$$M[min] = 3 * (n-1)$$
$$M[max] = trunc (n^2/4) + 3*(n-1);$$
$$M[mit] = n*(ln_n + g)$$

Der Grundgedanke des Bubblesort ist sehr einfach. Man sucht immer das noch verbleibende, kleinste Element der Restliste und stellt dieses an die aktuelle Position. D.h. im ersten Durchlauf wird das erste Feldelement mit allen folgenden Elementen verglichen. Ist eines der Folgeelemente kleiner als das Element in der ersten Stelle, so tauscht man die beiden Elemente aus und setzt die Suche fort. Dadurch wird immer mit dem bisher

[5] Die Anzahl der Bewegungen ist die Anzahl der notwendigen Umstellungen der Feldelemente (Umkopieren).

kleinsten, gefundenen Element verglichen. Hat man mit allen Elementen verglichen, so steht in der ersten Zelle des Arrays jetzt das kleinste Element.

In den nächsten Schritten vergleicht man nun das zweite Element (das erste ist ja schon das kleinste) mit allen folgenden Elementen (3. bis n. Element), um das zweitkleinste zu finden. Danach folgt das drittkleinste usw.

Am Schluss braucht man nur noch das vorletzte Element mit dem letzten Element zu vergleichen und ggf. zu tauschen, dann ist das Feld sortiert.

Der Algorithmus wird Bubblesort genannt, weil — plastisch ausgedrückt — die kleinste noch nicht sortierte Zahl im Array „aufsteigt“ wie eine Blase (Bubble) in einem Mineralwasser.

	(1)	(2)	(3)	(4)	(5)	(6)	(7)
1	5	3	1	1	1	1	1
2	7	7	7	5	3	3	3
3	3	5	5	7	7	5	5
4	1	1	3	3	5	7	6
	...	...	...	...	...	...	...
n	6	6	6	6	6	6	7

Für den Bubblesort benötigt man also zwei ineinander geschachtelte Schleifen, die erste Schleife von 1. bis zum n-1. Element, die zweite vom jeweils aktuellen bis zum n. Element (nicht vergessen, in C/C++ beginnt die Zählung bei 0 !).

Hier ein Beispiel für ein Array mit 30 Elementen:

```
void main (void)
{
   int i, j;

   for (i=0; i<=28; i++)
   {
      for (j=i+1; j<30; j++)
      {
         ... /* hier die Austausch-Anweisungen */
      }
   }
}
```

15.9. AUFGABENTEIL

Bei allen Aufgaben ist ein Schwierigkeitsgrad angegeben. Der Schwierigkeitsgrad bezieht sich nicht allein auf die Aufgabenstellung sondern ggf. auch auf den Umfang der Ausgabe.

15.9.1. AUFGABE 1-A (MITTEL)

Schreiben Sie ein Programm, das zehn Zahlen einliest und anschließend sortiert. Verwenden Sie zum Einlesen und sortieren ein einfaches Array mit zehn Speicherplätzen. Geben Sie anschließend das Feld sortiert wieder aus. Achten Sie bitte darauf, was geschieht wenn sich gleiche Zahlen im Array befinden.

15.9.2. AUFGABE 1-B (SCHWIERIG)

Schreiben Sie ein Programm, das insgesamt zehn Datumsangaben einliest und anschließend sortiert. Überlegen Sie sich eine passende Datenstruktur (Feld), die es Ihnen ermöglicht mit möglichst wenig Programmaufwand auszukommen. Die Eingabe soll aus Tag, Monat und Jahr bestehen. Verwenden Sie zum Einlesen und sortieren ein zweidimensionales Array mit drei mal zehn Speicherplätzen. Geben sie am Ende zwei Listen aus:

- Datumsangaben nach Zeit sortiert (also 1.12.1975 vor 3.7.1985)
- Eine Geburtstagsliste (Jahreszahlen spielen keine Rolle).

15.9.3. AUFGABE 2 (MITTEL)

Schreiben Sie ein Programm, das zehn Zahlen auf einem Array einliest und diese folgendermaßen weiterverarbeitet:

- Bilden Sie die Summe über alle Zahlen.
- Bilden Sie die Summe über alle geraden Zahlen.
- Bilden Sie das Produkt über alle Zahlen mit ungeradem Index.
- Bilden Sie das Produkt über die ersten fünf Zahlen.
- Ziehen Sie die Summe der ersten vom Produkt der zweiten fünf Zahlen ab.
- Bilden Sie den Mittelwert aller Zahlen mit geradem Index.
- Bilden Sie den Mittelwert aller ungeraden Zahlen.

15.9.4. AUFGABE 3 (SCHWIERIG)

Schreiben Sie ein Programm, das für einen einzelnen Arbeitnehmer die täglich geleistete Arbeitszeit speichern kann. Verwenden Sie dazu ein zweidimensionales Array, dessen zweite Dimension von variabler Länge ist (Vereinbarung über Pointer). Fügen Sie außerdem einen Programmteil an, der die Gesamtstundenzahl der Monate und des Jahres auswertet.

16. ZEICHENKETTEN / STRINGS

In diesem Abschnitt wird die einfache Stringverarbeitung vorgestellt, die in C und C++ vorhanden ist.

Im nachfolgenden Abschnitt ist das Stringstream-Konzept von C++ zu finden, welches die einfache Standardform der Stringverarbeitung abgelöst hat, wie die IO-Streams die Standard-IO.

Da beide Konzepte unter C++ vorhanden sind, ist es (wie bei der Stream-IO) eher eine Frage der Gewöhnung, für welche Befehle man sich während des Entwicklungsprozesses entscheidet.

Zeichenketten werden in C immer als Felder des Typs *unsigned char* oder, was dem entspricht, als Pointer auf Character gebildet. Da Strings von erheblich unterschiedlicher Länge sein können und die Entwickler der Programmiersprache keine Maximallänge vorschreiben wollten (wie z.B. in älteren Releases von Turbo-Pascal), wird das Ende eines Strings durch eine binäre Null (*NULL*, auch *NULL*-Byte oder String-Terminator genannt) gekennzeichnet. Da ein solches Zeichen nicht auf der Tastatur existiert, wird es in einem Quelltext durch das Sonderzeichen *'\0'* dargestellt.

Der Programmierer muss allerdings immer im Gedächtnis behalten, dass der String dadurch um ein Zeichen länger ausfällt als für den vorgesehenen Inhalt eigentlich nötig gewesen wäre. Fehlt die Stringende-Kennung, so wird bei einer Bearbeitung oder Ausgabe des Strings durch eine Funktion, das vorgesehene Stringende überschritten und die Bearbeitung solange fortgeführt, bis irgendwo im Speicher ein Null-Wert steht. Dies ist meist weit hinter dem eigentlich vorgesehenen Ende der Zeichenkette - natürlich können dadurch wichtige Daten zerstört werden. Die Deklaration eines Strings sieht wie folgt aus:

```
//============================================================
// Programm STRING1.CPP
//============================================================

#include <iostream>
#include <iomanip>

using namespace std;

//------------------------------------------------------------
// Stringdeklarationen mit Initialisierung
// Alle vier Deklarationen haben ein identisches Ergebnis
//------------------------------------------------------------
char Gruss1 [6] = "Hallo";
char Gruss2 []  = {'H','a','l','l','o','\0'};
char Gruss3 []  = "Hallo";
char Gruss4 [6] = {'H','a','l','l','o','\0'};
```

```
void main (void)
{
   cout << Gruss1 << endl;
   cout << Gruss2 << endl;
   cout << Gruss3 << endl;
   cout << Gruss4 << endl;
}
```

Die beiden aufgeführten Beispiele sind inhaltlich identisch. Die Stringvariablen *Gruss1* und *Gruss4* haben eine festgelegte Maximallänge von sechs Zeichen, mit den Indizes Null bis Fünf. Von diesen sechs Zeichen sind aber nur insgesamt fünf Zeichen für Benutzerdaten nutzbar, da ja noch das Stringende-Zeichen angefügt werden muss. Die Schreibweise der Zeichenkettenvariablen *Gruss2* und *Gruss4* ist erheblich umständlicher und daher unüblich. Hier ist sie lediglich aufgeführt, um den Feldcharakter von Strings zu verdeutlichen.

Verwenden Sie für die Zuweisung einer leeren Zeichenkette (s.u.) oder eines Stringende-Zeichens jeweils eine symbolische oder echte Konstante:

```
#define EMPTYSTRING ""
#define STRINGEND '\0'
```

bzw.

```
const char EmptyString [] = "";
const char StringEnd      = '\0';
```

Wie ein Feld kann auch ein String ohne Benutzerangabe der Feldlänge definiert werden, der Rechner zählt dann die Zeichen des Initialisierungsstrings.

Die Initialisierungen der beiden aufgeführten Beispiele sind austauschbar. Bei der zweiten Initialisierungsvariante muss das Stringende-Zeichen von Hand angegeben werden (Aufzählung aller Einzelzeichen im String), bei Angabe der Zeichenkette in doppelten Hochkommata, setzt der Rechner das Stringende-Zeichen automatisch.

In Zeichenketten können alle Fluchtsymbole verwendet werden, die auch für die Ausgabe von Zeichenketten definiert wurden (siehe Tabelle 5-4: Fluchtsymbole in der Standard-IO).

```
//=========================================================
// Programm STRING2.CPP
//=========================================================

#include <stdio.h>

//---------------------------------------------------------
```

```
// Zeiger auf eine Zeichenkette
//-------------------------------------------------------------
char *Name = "Charlie Brown";

//-------------------------------------------------------------
// Zuweisung eines konstanten Strings an einen Zeiger
//-------------------------------------------------------------
char *Zkette1 = "Test für eine Zeichenkette";

//-------------------------------------------------------------
// Zuweisung eines variablen Strings an einen Zeiger
//-------------------------------------------------------------
char *Zkette2 = Name;

//-------------------------------------------------------------
// Beispiel für eine Leere Zeichenkette
//-------------------------------------------------------------
char Leer [20] = "";

void main (void)
{
   printf ("%s\n", Name);
   printf ("%s\n", Zkette1);
   printf ("%s\n", Zkette2);
   printf ("%s\n", Leer);
}
```

Da es sich bei Stringvariablen immer nur um einen Zeiger auf jenen Speicherbereich handelt, in welchem der Text steht, ist das Kopieren einer Zeichenkette nicht durch eine Zuweisung möglich. Die folgende Anweisung würde lediglich zwei Pointer auf den gleichen Bereich zeigen lassen, nicht jedoch den Text duplizieren:

```
char *Name  = "Charlie Brown";
char *Name2 = "Woodstock";

void main (void)
{
   Name2 = Name;
}
```

Die beschriebene Operation hat im Speicher die folgende Auswirkung bei der Zuweisung der Pointer:

Vor der Zuweisung:			
Variable	Adresse	Inhalt	Inhalt des Verweises
Name	5000	1000	"Charlie Brown"
Name2	6000	1100	"Woodstock"

Tabelle 16-1: Auswirkungen einer Stringzuweisung 1

Nach der Zuweisung:			
Variable	Adresse	Inhalt	Inhalt des Verweises
Name	5000	1000	"Charlie Brown"
Name2	6000	1000	"Charlie Brown"

Tabelle 16-2: Auswirkungen einer Stringzuweisung 2

Wie leicht zu erkennen ist, verweist nach der Zuweisung kein Variablenname mehr auf den Speicherinhalt *"Woodstock"*. Der Speicherplatz ist dadurch für den aktuelle Programmlauf endgültig verloren gegangen, da es keinerlei Methode gibt, die Adresse des Textes zu ermitteln (der Speicherplatz kann ohne Pointer auch nicht wieder freigegeben werden). Um eine Zeichenkette zu kopieren, muss man daher auf eine der vielen Bibliotheksfunktionen zurückgreifen, die unten kurz beschrieben werden. Für Stringnamen gelten die üblichen Regeln für Pointer und Felder.

Eine Besonderheit ist bereits erwähnt worden - wird einem String bei der Initialisierung eine Zeichenkette zugewiesen, welche in doppelten Hochkommata steht, so wird am Ende der Zeichenkette das *'\0'*-Zeichen automatisch angefügt. Setzt man eine Zeichenkette jedoch selbst zusammen (wie bei *Gruss2*), so muss das Stringende-Zeichen von Hand angefügt werden. Es sei noch einmal betont, dass alle Standardfunktionen zur Stringverarbeitung das Stringende-Zeichen benötigen, um korrekt zu arbeiten.

16.1. ZEICHENKETTEN GRUNDFUNKTIONEN

Die folgende Auswahl an Zeichenketten-Funktionen ist üblicherweise in der Standardbibliothek enthalten. Um diese Funktionen nutzen zu können, muss über den *#include*-Präprozessor-Befehl das Headerfile *<string.h>* eingebunden werden. Eine der wichtigsten Bibliotheksfunktionen ist sicherlich das Kopieren einer Zeichenkette, da dieses, wie bereits oben beschrieben, nicht durch eine einfache Zuweisung möglich ist.

16.1.1. ZEICHENKETTEN KOPIEREN

```
Syntax:
   char *strcpy  (char *s, char *t);
   char *strncpy (char *s, char *t, int Anzahl);
```

```
//=============================================================
// Programm STRING3.CPP
//=============================================================

#include <stdio.h>
#include <string.h>
```

```
char Name1 [40] = "Charlie Brown";
char Name2 [40] = "Woodstock";

void main (void)
{
   printf ("\n1. Name: %s \n2. Name: %s", Name1, Name2);

   strcpy (Name1, "Snoopy");
   printf ("\n1. Name: %s \n2. Name: %s", Name1, Name2);

   strcpy (Name1, Name2);
   printf ("\n1. Name: %s \n2. Name: %s", Name1, Name2);

   strncpy (Name1, "Snoopy", 4);
   printf ("\n1. Name: %s \n2. Name: %s", Name1, Name2);
}
```

Die Funktion *strcpy* kopiert die Zeichenkette *t* auf die Zeichenkette *s* und gibt den Zeiger auf *s* als Ergebnis zurück. Hierbei sind *s* und *t* jeweils Pointer auf eine Zeichenkette, d.h. die Feldnamen der Strings (ohne Index).

Die Funktion *strncpy* ist identisch mit der Funktion *strcpy*, kopiert aber nur die angegebene *Anzahl* an Zeichen. Wenn die Zeichenkette *t* länger als die angegebene *Anzahl* ist, so bleibt die Zielzeichenkette *s* ohne ein abschließendes Stringende-Symbol (!). Ist die Zeichenkette *t* hingegen kürzer als die in *Anzahl* angegebene Zeichenmenge, dann wird *s* bis auf *Anzahl* Zeichen mit Stringende-Symbolen aufgefüllt.

16.1.2. ZEICHENKETTE ANHÄNGEN

```
Syntax:
   char *strcat  (char *s, char *t);
   char *strncat (char *s, char *t, int Anzahl);
```

```
//=========================================================
// Programm STRING4.CPP
//=========================================================

#include <stdio.h>
#include <string.h>

char Name [80] = "Charlie Brown";

void main (void)
{
   printf ("\n Peanuts: %s", Name);
   strcat (Name, ", Snoopy");
```

```
   printf ("\n Peanuts: %s", Name);

   strcat (Name, " & Co");
   printf ("\n Peanuts: %s", Name);

   strncat (Name, " immer komisch", 7);
   printf ("\n Peanuts: %s", Name);
}
```

Die Konkatenation, d.h. das Zusammenfügen von Stringinhalten wird in C von der Funktion *strcat* realisiert. Die Parameter *s* und *t* sind jeweils vom Typ Pointer auf Character, d.h. die Feldnamen der Strings (ohne Index). Die Funktion fügt eine Kopie von *t* an *s* an und gibt den Zeiger auf *s* als Funktionsergebnis zurück.

Die Funktion *strncat* ist identisch mit der Funktion *strcat*, kopiert aber maximal nur die angegebene *Anzahl* an Zeichen. Wenn die Zeichenkette *t* länger als die angegebene *Anzahl* ist, so werden *Anzahl* Zeichen und ein Stringende-Zeichen an *s* angehängt (im Gegensatz zu strncpy, s.o., wo die Zielzeichenkette ohne ein abschließendes Stringende-Symbol bleibt). Ist die Zeichenkette *t* hingegen kürzer als die in *Anzahl* angegebene Zeichenmenge, dann werden entsprechend weniger Zeichen an *s* angefügt.

16.1.3. ZEICHEN IN EINER ZEICHENKETTE SUCHEN

```
Syntax:
   char *strchr  (char *s, int c);
   char *strrchr (char *s, int c);
```

```
//=========================================================
// Programm STRING5.CPP
//=========================================================

#include <stdio.h>
#include <string.h>

char Name1 [40] = "Charlie Brown";
char Name2 [40] = "Woodstock";
char CharVar    = 'B';

void main (void)
{
   char *StrPtr1 = NULL;
   char *StrPtr2 = NULL;

   StrPtr1 = strchr (Name2, 'o');
   StrPtr2 = strchr (Name2, CharVar);
```

```
   printf ("\n1. Name: %s \n2. Name: %s", Name1, Name2);
   printf ("\n1. Pointer: %s", StrPtr1);
   printf ("\n2. Pointer: %s", StrPtr2);
}
```

Die Funktion *strchr* sucht im String *s* nach dem angegebenen Zeichen *c*. Wird ein solches Zeichen gefunden, so gibt die Funktion einen Zeiger auf die Position dieses Zeichens zurück. Ist das Zeichen *c* in der Zeichenkette *s* nicht enthalten, so wird ein *NULL*-Zeiger zurückgegeben. Ist das Zeichen mehrfach vorhanden, so wird der Zeiger auf das erste Vorkommen des Zeichens *c* in *s* zurückgeliefert.

Die Funktion *strrchr* verhält sich wie *strchr*, durchsucht die Zeichenkette aber von rechts nach links, findet also das letzte in *s* vorkommende Zeichen *c*.

16.1.4. ZEICHENKETTE IN EINER ZEICHENKETTE SUCHEN

```
Syntax :
   char *strstr (char *s, char *t);
```

```
//============================================================
// Programm STRING6.CPP
//============================================================

#include <stdio.h>
#include <string.h>

char Name1 [40] = "Charlie Brown";
char Name2 [40] = "Woodstock";
char Suche []   = "Brown";

void main (void)
{
   char *StrPtr1 = NULL;
   char *StrPtr2 = NULL;

   StrPtr1 = strstr (Name2, "st");
   StrPtr2 = strstr (Name1, Suche);

   printf ("\n 1. Name: %s \n2. Name:", Name1, Name2);
   printf ("\n 1. Pointer: %s", StrPtr1);
   printf ("\n 2. Pointer: %s", StrPtr2);
}
```

Die Funktion *strstr* sucht im String *s* nach der angegebenen Zeichenkette *t*. Wird eine solche Zeichenkette gefunden, so gibt die Funktion einen Zeiger auf die Position dieses Teilstrings in der

durchsuchten Zeichenkette zurück. Ist die Teilzeichenkette im String *s* nicht enthalten, so wird ein *NULL*-Zeiger zurückgegeben.

16.1.5. ZEICHENMENGE IN ZEICHENKETTE SUCHEN

```
Syntax:
   char *strpbrk (char *s, char *t);
```

```
//=============================================================
// Programm STRING7.CPP
//=============================================================

#include <stdio.h>
#include <string.h>

char Name [40] = "Peanuts und Charlie Brown";

void main (void)
{
   char *StrPtr = NULL;

   StrPtr = strpbrk (Name, "CB");
   printf ("\nPointer: %s", StrPtr);
}
```

Die Funktion *strpbrk* sucht im String *s* nach dem ersten Auftreten eines beliebigen Zeichens aus der Zeichenmenge von *t*. Wird ein solches Zeichen gefunden, so gibt die Funktion einen Zeiger auf die Position dieses Zeichens zurück. Ist kein Zeichen aus *t* in der Zeichenkette *s* enthalten, so wird ein *NULL*-Zeiger zurückgegeben. Sind mehrere Zeichen vorhanden oder ein Zeichen mehrfach, so wird der Zeiger auf das erste Vorkommen *s* zurückgeliefert. Das oben dargestellte Beispiel liefert somit einen Zeiger auf den ersten im String enthaltenen Vokal zurück.

Das folgende Beispiel zeigt, wie man mit Hilfe der Funktion *strpbrk* und einer einfachen Schleife alle Vokale in einem Text durch einen anderen Vokal ersetzen kann.

```
//=============================================================
// Programm CHINESEN.CPP
//=============================================================

#include "stdafx.h"  // VISUAL C++ spezifisch
#include <iostream>
#include <string.h>

using namespace std;
```

```
// int main (void)                    // Andere C++ Compiler
int _tmain(int argc, _TCHAR* argv[]) // VISUAL C++
{
  char sText [] = "Drei Chinesen mit dem Kontrabass";
  char *pVokal   = strpbrk (sText, "aeiou");
  char cErsetzen = 'i';

  cout << sText << endl;

  while (pVokal)
  {
    *pVokal = cErsetzen;
    pVokal  = strpbrk (pVokal+1, "aeiou");
  }
  cout << sText << endl;

  cErsetzen = 'o';
  pVokal = strpbrk (sText, "aeiou");
  while (pVokal)
  {
    *pVokal = cErsetzen;
    pVokal  = strpbrk (pVokal+1, "aeiou");
  }
  cout << sText << endl;
  return 0;
}
```

16.1.6. ZEICHENKETTEN VERGLEICHEN

```
Syntax:
   int strcmp   (char *s, char *t);
   int strcmpi  (char *s, char *t);
   int stricmp  (char *s, char *t);
   int strncmp  (char *s, char *t, Anzahl);
   int strncmpi (char *s, char *t, Anzahl);
   int strnicmp (char *s, char *t, Anzahl);
```

```
//=============================================================
// Programm STRING8.CPP
//=============================================================

#include <stdio.h>
#include <string.h>

char Name1 [40] = "Charlie Brown";
char Name2 [40] = "Woodstock";
int Resultat    = 0;

void main (void)
{
   //-----------------------------------------------------
   // Ergebnis ist negativ, da Name2 > Name1 ist
```

```
   //-------------------------------------------------------
   Resultat = strcmp (Name1, Name2);
   printf ("\n 1. Resultat: %d", Resultat);

   //-------------------------------------------------------
   // Ergebnis Null, beide Strings sind gleich aber 0 ist in
   // C/C++ gleichbedeutend mit FALSE!
   //-------------------------------------------------------
   Resultat = strcmp (Name2, "Woodstock");
   printf ("\n 2. Resultat: %d", Resultat);

   //-------------------------------------------------------
   // Umkehrung der 0 durch NOT(!), Ausdruck ist TRUE
   //-------------------------------------------------------
   if (!strcmp (Name2, "Woodstock"))
   {
      printf ("\nStrings sind gleich");
   }
}
```

Die Funktion *strcmp* vergleicht die angegebenen Zeichenketten miteinander. Sind die Zeichenketten gleich (enthalten die gleichen Zeichen), so ist das Ergebnis der Wert Null (entspricht *NULL* oder *FALSE*! - die Abfragelogik muss also umgekehrt werden, wie im Beispiel oben). Das Ergebnis wird ermittelt indem der Inhalt von *t* von *s* abgezogen wird. Sind die Zeichenketten gleich, so wird am Ende der Wert Null zurückgegeben, ansonsten die erste aufgetretene Differenz (positiv oder negativ). Dies bedeutet aber auch, dass der Rückgabewert negativ ist, wenn der ASCII-Wert der Zeichens in *t* größer ist als der des Zeichens in *s*. Ist der Rückgabewert hingegen positiv, so war der Wert des Zeichens in *s* größer. Sind zwei Zeichenketten von unterschiedlicher Länge, können sie nicht gleich sein, da am Ende der ASCII-Wert der längeren Zeichenkette (bei der Subtraktion mit dem Stringende-Zeichen, Wert ASCII-Null, also *NULL*) übrig bleibt.

Die Funktionen *strcmpi* und *stricmp* sind identisch und nur aus Kompatibilitätsgründen beide deklariert. Sie verhalten sich genau wie *strcmp*, unterscheiden jedoch nicht zwischen Groß- und Kleinschreibung (dies gilt u.U. nicht für die deutschen Umlaute!!).

Die Funktion *strncmp* vergleicht maximal die ersten *Anzahl* Zeichen der beiden Zeichenketten miteinander. Sind die Zeichenketten kürzer oder tritt bereits vorher ein Unterschied auf, so bricht der Vergleich vorher mit einem entsprechenden Ergebnis ab.

Die Funktionen *strncmpi* und *strnicmp* sind identisch und (genau wie *strcmpi* und *stricmp)* nur aus Kompatibilitätsgründen beide deklariert. Sie sind die Kombination von *strncmp* und *stricmp*, unterscheiden also nicht zwischen Groß- und Kleinschreibung (dies gilt u.U. nicht für die

deutschen Umlaute!!) und vergleichen maximal die ersten *Anzahl* Zeichen der beiden Zeichenketten miteinander.

16.1.7. LÄNGE EINER ZEICHENKETTE ERMITTELN

```
Syntax:
   int strlen (char *s);
```

```
//=============================================================
// Programm STRING9.CPP
//=============================================================

#include <stdio.h>
#include <string.h>

char Name [40] = "Charlie Brown";
int Resultat   = 0;

void main (void)
{
   Resultat = strlen (Name);  // Ergebnis ist hier 12!
   printf ("\n Länge ist: %d", Resultat);
}
```

Die Funktion *strlen* ermittelt die Länge des angegebenen Strings *s*. Die Längenermittlung erfolgt durch eine Pointersubtraktion (Adresse des letzten Zeichens Minus Adresse des ersten Zeichens). Die Funktion kopiert dabei den Pointer *s* und verschiebt die Zeigerkopie bis sie ein Stringende Symbol findet, anschließend werden die Pointer voneinander abgezogen, das Ergebnis ist dann die Anzahl der enthaltenen Zeichen.

16.2. ZEICHENKETTEN EIN-/AUSGABE

Bisher erfolgten alle Ein- und Ausgabeanweisungen mit *printf* und *scanf*. Beide Funktionen sind sehr vielfältig in ihren Möglichkeiten, haben jedoch den Nachteil sehr groß oder für Zeichenketten nicht geeignet zu sein. Insbesondere *printf* mit seinen vielen Formatierungsmöglichkeiten ist sehr umfangreich und somit nicht das Mittel der Wahl, um einfach eine Zeichenkette, so wie sie gerade ist, auszugeben.

16.2.1. ZEICHENKETTENAUSGABE MIT PUTS

Die Funktion *puts* schreibt eine Zeichenkette (ASCII-Werte) auf das Standard-Ausgabegerät (Bildschirm). Am Ende wird automatisch ein Zeilenvorschubzeichen ('\n') angehängt. Es wird ein Fehlercode zurückgegeben, der ungleich Null ist, wenn ein Fehler aufgetreten ist. Der Rückgabewert kann ignoriert werden.

```
Syntax :
   error = puts (String_var);
   puts (String_var);
```

```
#include <stdio.h>
#include <string.h>

#define EMPTYSTRING ""

char Zeichenkette [200] = "Bitte Text eingeben: ";

void main (void)
{
   printf ("%s\n", Zeichenkette);

   //-------------------------------------------------------
   // ist identisch mit
   //-------------------------------------------------------
   puts (Zeichenkette);
}
```

16.2.2. ZEICHENKETTENEINGABE MIT GETS

Die Funktion *gets* liest eine Zeichenkette (ASCII-Werte) vom Standard-Eingabegerät (Tastatur) ein. Es wird ein Zeiger auf die eingelesene Stringvariable *String_Var* zurückgegeben oder *NULL* bei einem Fehler. Der Rückgabewert kann ignoriert werden.

```
Syntax :
   String_ptr = gets (String_var);
   gets (String_var);
```

```
//============================================================
// Programm STRING10.CPP
//============================================================

#include <stdio.h>
#include <string.h>

#define EMPTYSTRING ""

char Zeichenkette [200] = EMPTYSTRING;

void main (void)
{
   printf ("Bitte Text eingeben: ");
   gets (Zeichenkette);
   printf ("\n\nUnd hier ist er wieder: \n");
   puts (Zeichenkette);
}
```

Für Zeichenketten ist die Funktion *gets* der Funktion *scanf* in jedem Fall vorzuziehen, denn die *scanf*–Funktion interpretiert alle Leerzeichen als Trennzeichen zwischen mehreren Variablen.

Leider gibt es bei gets *keine Möglichkeit die Anzahl der einzulesenden Zeichen zu bestimmen. Dafür kann aber die Funktion* fgets *verwendet werden, die eigentlich Zeichenketten aus Dateien lesen soll. Da jedoch die Tastatur als „Quasidatei" (Logisches Device) funktioniert, kann – anstelle eines Dateinamens – die Tastatur als Eingabequelle genannt werden (*stdin*).*

```
Syntax :
   String_ptr = fgets (String_var, Anzahl, stdin);
   fgets (String_var, Anzahl, stdin);
```

Die Funktion *fgets* liest maximal Anzahl-1 Zeichen und hängt automatisch ein Stringendesymbol (ASCII-Null) an. Dadurch, dass man die Datei-Version von gets verwendet muss man jedoch mit dem Nachteil leben, dass bei Eingabe kürzerer Zeichenketten als Anzahl Zeichen das *Linefeed*-Zeichen (ASCII 10) Teil der Zeichenkette ist. Dieses Zeichen muss man gesondert herausfiltern, was jedoch nicht besonders schwer ist, da es sich um das letzte Zeichen vor dem Stringendesymbol handelt. Die folgende Funktion erledigt das automatisch und kann anstelle eines *gets*-Aufrufes verwendet werden.

```
//============================================================
// Programm SAVEGETS.CPP
//============================================================

#include <stdio.h>
#include <string.h>

//------------------------------------------------------------
// s ist Zeichenkette auf die einzulesen ist, Anz ist die
// maximale Anzahl von einzulesenden Zeichen inkl. ASCII-Null
//------------------------------------------------------------

char *SaveGets (char *s, int Anz)
{
   int len = 0;

   //---------------------------------------------------------
   // zuerst einmal einlesen und Länge -1 feststellen
   //---------------------------------------------------------
   fgets (s, Anz, stdin);
   len = strlen (s) - 1;

   //---------------------------------------------------------
   // prüfen ob letztes Zeichen ein LF ist, wenn ja mit
```

```
   // ASCII-Null ersetzen
   //-----------------------------------------------------
   if (s [len] == 10) s [len] = 0;
   return (s);
}
```

```
//=========================================================
// Programm SAVEGETS.H
//=========================================================

#ifndef _SAVEGETS_H_
#define _SAVEGETS_H_

char *SaveGets (char *s, int Anz);

#endif
```

Nähere Informationen zur Syntax von *fgets* sind dem Kapitel über Dateien zu entnehmen.

16.3. ZEICHENKETTEN AUFBEREITEN

Die folgenden Funktionen sind dazu gedacht, um Ein-/Ausgaben zwischen Zeichenketten zu ermöglichen und Formatierungen vorzunehmen bzw. formatierte Zeichenketten zu verarbeiten.

16.3.1. FORMATIERT IN EINEN STRING AUSGEBEN

```
Syntax:
   sprintf (Zeichenkette, Formatstring, Varliste);
```

Die Funktion *sprintf* ist in allen Belangen identisch mit der Funktion *printf*, nur dass das Ausgabeziel nicht der Bildschirm sondern der Speicherbereich einer Zeichenkette ist. D.h. alle Formatierungsschalter und Fluchtsymbole die man bei *printf* verwenden kann, sind auch bei *sprintf* gültig.

```
//=========================================================
// Programm STRING11.CPP
//=========================================================

#include <stdio.h>
#include <string.h>
#include <math.h>

#define EMPTYSTRING ""
#define BLANK       32

char String [255] = EMPTYSTRING;
```

```
//-------------------------------------------------------
// Diese Funktion ersetzt alle Leerzeichen durch '*'
//-------------------------------------------------------

void ChangeSpaceToStar (char *s)
{
   int i;
   int len = strlen (s);

   for (i=0; i<len; i++) if (s[i] == BLANK) s[i] = '*';
}

void main (void)
{
   double Betrag = 17.45;

   sprintf (String, "%10.2lf", Betrag);
   ChangeSpaceToStar (String);
   printf ("\nBetrag formatiert: %s", String);
}
```

Die *sprintf*-Funktion ist sehr nützlich, um eigene, zusätzliche Unterprogramme zur Formatierung zu realisieren oder um Datensätze (Strukturen, siehe unten) zu bearbeiten. Wie bei allen Stringfunktionen findet auch hier keine Überprüfung der Anzahl der auf den String zugewiesenen Zeichen statt. Der Programmierer muss selbst für entsprechende Vorsichtsmaßnahmen sorgen.

16.3.2. FORMATIERT AUS EINEM STRING LESEN

```
Syntax:
   sscanf (Zeichenkette, Formatstring, Varliste);
```

Die Funktion *sscanf* ist in allen Belangen identisch mit der Funktion *scanf*, nur dass die Quelle nicht die Tastatur sondern der Speicherbereich einer Zeichenkette ist. D.h. alle Möglichkeiten und Einschränkungen die die Funktion *scanf* hat, sind auch bei *sscanf* gültig.

```
#include <stdio.h>
#include <string.h>

int Tag, Monat, Jahr;
char String [255] = "24 12 2000";

void main (void)
```

```
{
   sscanf (String, "%d%d%d", &Tag, &Monat, &Jahr);
   printf ("\nDatum: %2d.%2d.%4d", Tag, Monat, Jahr);
}
```

Die *sscanf*-Funktion leidet unter den gleichen Unzulänglichkeiten wie die *scanf*-Funktion, sie ist nur im Umgang mit formatierten Datendateien (siehe Kapitel über Dateien) nützlich — und auch dies nur, wenn die Dateien keine Zeichenketten enthalten, da *scanf* und *sscanf* das Leerzeichen als Trennzeichen betrachten.

16.3.3. ZEICHENKETTE IN GROSSBUCHSTABEN UMSETZEN

```
Syntax:
   String_ptr = strupr (Zeichenkette);
```

Die Funktion *strupr* setzt alle Buchstaben in der übergebenen Zeichenkette in Großbuchstaben um. Ziffern und Sonderzeichen (damit leider auch deutsche Sonderzeichen) bleiben unberührt.

```
#include <stdio.h>
#include <string.h>

char String [255] = "Hallo, wie geht's? - äöüß";

void main (void)
{
   puts (String);
   strupr (String);
   puts (String);
}
```

Die folgende Funktion löst das Problem, basieren auf einer eigenen Funktion *gertoupper*, die weiter unten beschrieben wird.

```
//============================================================
// Die deutschen Programmvarianten sind in der Datei
// STRUTIL.CPP zusammengefasst, die zugehörige Headerdatei
// STRUTIL.H deklariert die Prototypen.
// STRUTILH.CPP zeigt Beispielaufrufe der Funktionen
//============================================================

#include <stdio.h>
#include <string.h>

char *gerstrupr (char *s)
{
   char *cp = s;
```

```
   while (*cp)
   {
      *cp = gertoupper (*cp);
      cp++;
   }
   return (s);
}
```

16.3.4. ZEICHENKETTE IN KLEINBUCHSTABEN UMSETZEN

```
Syntax:
   String_ptr = strlwr (Zeichenkette);
```

Die Funktion *strlwr* setzt alle Buchstaben in der übergebenen Zeichenkette in Kleinbuchstaben um. Ziffern und Sonderzeichen (damit leider auch deutsche Sonderzeichen) bleiben unberührt.

```
#include <stdio.h>
#include <string.h>

char String [255] = "HALLO, WIE GEHT'S? - ÄÖÜß";

void main (void)
{
   puts (String);
   strlwr (String);
   puts (String);
}
```

Die folgende Funktion löst das Problem, basieren auf einer eigenen Funktion *gertolower*, die weiter unten beschrieben wird.

```
//============================================================
// Die deutschen Programmvarianten sind in der Datei
// STRUTIL.CPP zusammengefasst, die zugehörige Headerdatei
// STRUTIL.H deklariert die Prototypen.
// STRUTILH.CPP zeigt Beispielaufrufe der Funktionen
//============================================================

#include <stdio.h>
#include <string.h>

char *gerstrlwr (char *s)
{
   char *cp = s;
   while (*cp)
   {
      *cp = gertolower (*cp);
      cp++;
   }
```

```
   return (s);
}
```

16.3.5. ZEICHENKETTE ZERLEGEN

```
Syntax:
   String_ptr = strtok (Zeichenkette, Trennsymbol);
```

Mit der Funktion strtok (String-Token) kann eine Zeichenkette, die mehrere Abschnitte enthält (gegliedert durch ein Trennzeichen) schrittweise in ihre einzelnen Bestandteile zerlegt werden.

Das folgende Beispielprogramm macht die Einsatzmöglichkeit der Funktion deutlich:

```
//============================================================
// Programm STRING12.CPP
//============================================================

#include <stdio.h>
#include <string.h>

char test [] = "hallo,abc,def,ghi";

int main (void)
{
   char *p;
   puts (test);
   p = strtok (test, ",");
   while (p)
   {
      puts (p);
      p = strtok (NULL, ",");
   }
    puts (test);
}
```

Über den Pointer *p* wird nacheinander auf die einzelnen Teile zugegriffen. Die Zeichenkette, in der die Token liegen muss nur beim ersten Zugriff angegeben werden, die Übergabe von *NULL* bei den nächsten Aufrufen gibt der Funktion bekannt, dass man das folgende Token haben möchte. Die Funktion gibt den *NULL*-Zeiger zurück, wenn kein weiteres Token im String enthalten ist.

16.3.6. ZEICHENKETTE IN INT ODER LONG UMWANDELN

```
Syntax:
   int_var  = atoi   (Zeichenkette);
   long_var = atol   (Zeichenkette);
   int_var  = strtol (Zeichenkette, Endptr, Radix);
   long_var = strtoul(Zeichenkette, Endptr, Radix);
```

Die aufgeführten Funktionen setzen eine Zeichenkette in eine Zahl um. Dabei gilt, dass alle Funktionen sofort abbrechen, wenn sie auf ein Zeichen treffen, welches nicht interpretiert werden kann. Die Funktionen *atoi* und *atol* sind die einfachen Versionen, die nur Zahlen im Dezimalformat verarbeiten können.

Demgegenüber können die Funktionen *strtol* und *strtoul* auch andere Zahlensysteme verarbeiten. Das Zahlensystem wird in *Radix* angegeben und kann zwischen 2 und 36 [6] liegen. Dezimal-, Hexadezimal- und Oktalzahlen können auch automatisch erkannt und verarbeitet werden, wenn in *Radix* der Wert 0 angegeben wird. Hexadezimalzahlen werden dann an einem führenden '0x' erkannt - z B. 0x20 und Oktalzahlen an einer führenden Null - z.B. 0127 (alle Zahlen die mit 1-9 beginnen werden als Dezimalzahlen betrachtet). Zudem geben die Funktionen *strtol* und *strtoul* die Adresse des ersten nicht mehr analysierbaren Zeichens in *Endptr* zurück, bzw. den Wert *NULL*, wenn alle Zeichen übersetzt werden konnten.

- Das Ergebnis von `atoi` ist eine Integerzahl (ascii to integer).
- Das Ergebnis von `atol` ist ein long-Wert (ascii to long).
- Das Ergebnis von `strtol` ist ein long-Wert (string to long).
- Das Ergebnis von `strtoul` ist ein unsigned long-Wert (string to unsigned long).

Kann die Zeichenkette nicht interpretiert werden, weil sie keine Ziffern enthält oder zu Beginn gleich einen Buchstaben (Leerzeichen werden ignoriert), so geben diese Funktionen den Wert 0 zurück.

[6] Die Anzahl 26 scheint willkürlich. Da aber Zahlenwerte durch Buchstaben ersetzt werden (beim Hexadezimalformat werden die „Ziffern" für die Zahlenwerte 10 bis 15 durch die Buchstaben A bis F repräsentiert) und das (amerikanische) Alphabet nur 26 Buchstaben (A bis Z) kennt, können maximal 26 „Ziffern" hinzugefügt werden.

```
//=============================================================
// Programm STRING13.CPP
//=============================================================

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char String [255] = "1234";
char Test   [255] = "1234D123";
int x;
char *p;

void main (void)
{
   x = atoi (String);
   printf ("%d\n", x);
   x = strtol (Test, &p, 0);
   printf ("%d\n", x);
}
```

16.3.7. ZEICHENKETTE IN DOUBLE UMWANDELN

```
Syntax:
   double_var = atof   (Zeichenkette);
   double_var = strtod (Zeichenkette, Endptr);
```

Die aufgeführten Funktionen setzen eine Zeichenkette in eine Fließkommazahl um. Dabei gilt, dass alle Funktionen sofort abbrechen, wenn sie auf ein Zeichen treffen, welches nicht interpretiert werden kann. Die Funktion *atof* ist die einfachere Version.

Die Funktion strtod gibt zusätzlich die Adresse des ersten nicht mehr analysierbaren Zeichens in *Endptr* zurück, bzw. den Wert *NULL*, wenn alle Zeichen übersetzt werden konnten.

- Das Ergebnis von `atof` ist eine Fließkommazahl vom Typ double (ascii to float).
- Das Ergebnis von `strtod` ist eine Fließkommazahl vom Typ double (string to double).

Kann die Zeichenkette nicht interpretiert werden, weil sie keine Ziffern enthält oder zu Beginn gleich einen Buchstaben (Leerzeichen werden ignoriert), so geben diese Funktionen den Wert 0 zurück. Die Zeichen für Exponentialformat (e, E) werden von beiden Funktionen übersetzt.

```
//=============================================================
// Programm STRING14.CPP
//=============================================================

#include <stdio.h>
#include <string.h>
#include <float.h>
#include <stdlib.h>

char String [255] = "1234.15";
char Test   [255] = "1234.123e12";
double x;
char *p;

void main (void)
{
   x = atof (String);
   printf ("%lf\n", x);
   x = strtod (Test, &p);
   printf ("%lf\n", x);
}
```

16.4. DIE WICHTIGSTEN CHARACTER-FUNKTIONEN

Die folgenden Funktionen sind ebenfalls in der Standardbibliothek enthalten. Um sie nutzen zu können muss das Headerfile *<ctype.h>* mittels der *#include*-Anweisung in das Programm eingebunden werden. Die deutschen Umlaute und das „ß“ gehören nicht zu den Standardzeichen und sind daher in den Bereichen „a-z“ und/oder „A-Z“ nicht enthalten.

16.4.1. TEST AUF ALPHABETISCHES ZEICHEN

```
Syntax:
   int isalpha (char c);
```

```
#include <stdio.h>
#include <ctype.h>

char Zeichen1 = '1';
char Zeichen2 = 'A';

void main (void)
{
   if (isalpha (Zeichen1) > 0)
      printf ("\nZeichen 1 ist alphabetisch!");

   if (isalpha (Zeichen2) > 0)
      printf ("\nZeichen 2 ist alphabetisch!");
}
```

Die Funktion *isalpha* gibt einen Wert ungleich Null zurück, wenn das Zeichen *c* aus dem alphabetischen Bereich („A-Z“ oder „a-z“) stammt, ansonsten wird der Wert Null zurückgegeben. Um auch die deutschen Umlaute mit einzubeziehen, muss man ein eigenes Unterprogramm schreiben, das auf der *isalpha*-Funktion basiert:

```
//============================================================
// Die deutschen Programmvarianten sind in der Datei
// STRUTIL.CPP zusammengefasst, die zugehörige Headerdatei
// STRUTIL.H deklariert die Prototypen.
// STRUTILH.CPP zeigt Beispielaufrufe der Funktionen
//============================================================

#include <ctype.h>
#include <string.h>

int gerisalpha (int c)
{
   if ((isalpha(c))||(strchr("äöüÄÖÜß",c))) return(c);
   return (0);
}
```

16.4.2. TEST AUF ALPHANUMERISCHES ZEICHEN

```
Syntax:
   int isalnum (char c);
```

```
#include <stdio.h>
#include <ctype.h>

char Zeichen1 = '1';
char Zeichen2 = 'A';

void main (void)
{
   if (isalnum (Zeichen1) > 0)
      printf ("\nZeichen 1 ist alphanumerisch!");
   if (isalnum (Zeichen2) > 0)
      printf ("\nZeichen 2 ist alphanumerisch!");
}
```

Die Funktion *isalnum* gibt einen Wert ungleich Null zurück, wenn das Zeichen *c* aus dem alphanumerischen Bereich („A-Z“, „a-z“ oder „0-9“) stammt, ansonsten wird der Wert Null zurückgegeben. Um auch die deutschen Umlaute mit einzubeziehen, muss man ein eigenes Unterprogramm schreiben, das auf der *isalnum*-Funktion basiert:

```
//=========================================================
// Die deutschen Programmvarianten sind in der Datei
// STRUTIL.CPP zusammengefasst, die zugehörige Headerdatei
// STRUTIL.H deklariert die Prototypen.
// STRUTILH.CPP zeigt Beispielaufrufe der Funktionen
//=========================================================

#include <ctype.h>
#include <string.h>

int gerisalnum (int c)
{
   if ((isalnum(c))||(strchr("äöüÄÖÜß",c))) return(c);
   return (0);
}
```

16.4.3. TEST AUF NUMERISCHES ZEICHEN

```
Syntax:
   int isdigit (char c);
```

```
#include <stdio.h>
#include <ctype.h>

char Zeichen1 = '1';
char Zeichen2 = 'A';

void main (void)
{
   if (isdigit (Zeichen1) > 0)
      printf ("\nZeichen 1 ist numerisch!");
   if (isdigit (Zeichen2) > 0)
     printf ("\nZeichen 2 ist numerisch!");
}
```

Die Funktion *isdigit* gibt einen Wert ungleich Null zurück, wenn das Zeichen *c* aus dem numerischen Bereich („0-9“) stammt, andernfalls wird der Wert Null zurückgegeben.

16.4.4. TEST AUF KLEINGESCHRIEBENES ZEICHEN

```
Syntax:
   int islower (char c);
```

```
#include <stdio.h>
#include <ctype.h>

char Zeichen1 = 'A';
char Zeichen2 = 'a';

void main (void)
{
   if (islower (Zeichen1) > 0)
      printf ("\nZeichen 1 ist klein!");
   if (islower (Zeichen2) > 0)
      printf ("\nZeichen 2 ist klein!");
}
```

Die Funktion *islower* gibt einen Wert ungleich Null zurück, wenn das Zeichen *c* aus dem Bereich der kleinen Buchstaben („a-z“) stammt. Sonst wird der Wert Null zurückgegeben. Um auch die deutschen Umlaute mit einzubeziehen, muss man ein eigenes Unterprogramm schreiben, das auf der *islower*-Funktion basiert:

```
//============================================================
// Die deutschen Programmvarianten sind in der Datei
// STRUTIL.CPP zusammengefasst, die zugehörige Headerdatei
// STRUTIL.H deklariert die Prototypen.
// STRUTILH.CPP zeigt Beispielaufrufe der Funktionen
//============================================================

#include <ctype.h>
#include <string.h>

int gerislower (int c)
{
   if ((islower(c))||(strchr("äöüß",c))) return(c);
   return (0);
}
```

16.4.5. TEST AUF GROSSGESCHRIEBENES ZEICHEN

```
Syntax:
   int isupper (char c);
```

```
#include <stdio.h>
#include <ctype.h>

char Zeichen1 = 'A';
char Zeichen2 = 'a';

void main (void)
{
   if (isupper (Zeichen1) > 0)
```

```
      printf ("\nZeichen 1 ist groß!");
   if (isupper (Zeichen2) > 0)
      printf ("\nZeichen 2 ist groß!");
}
```

Die Funktion *isupper* gibt einen Wert ungleich Null zurück, wenn das Zeichen *c* aus dem Bereich der großen Buchstaben („A-Z“) stammt Sonst wird der Wert Null zurückgegeben. Um auch die deutschen Umlaute mit einzubeziehen, muss man ein eigenes Unterprogramm schreiben, das auf der *isupper*-Funktion basiert:

```
//============================================================
// Die deutschen Programmvarianten sind in der Datei
// STRUTIL.CPP zusammengefasst, die zugehörige Headerdatei
// STRUTIL.H deklariert die Prototypen.
// STRUTILH.CPP zeigt Beispielaufrufe der Funktionen
//============================================================

#include <ctype.h>
#include <string.h>

int gerisupper (int c)
{
   if((isupper(c))||(strchr("ÄÖÜß",c))) return(c);
   return (0);
}
```

16.4.6. TEST AUF DRUCKBARES ZEICHEN

```
Syntax:
   int isprint (char c);
```

```
#include <stdio.h>
#include <ctype.h>

char Zeichen1 = 'ß';
char Zeichen2 = 3;

void main (void)
{
   if (isprint (Zeichen1) > 0)
      printf ("\nZeichen 1 ist druckbar!");
   if (isprint (Zeichen2) > 0)
      printf ("\nZeichen 2 ist druckbar!");
}
```

Die Funktion *isprint* gibt einen Wert ungleich Null zurück, wenn das Zeichen *c* aus dem Bereich der druckbaren Zeichen stammt (Buchstaben,

Zahlen, Satzzeichen, Graphiksymbole), ansonsten wird der Wert Null zurückgegeben.

16.4.7. TEST AUF LEERZEICHEN

```
Syntax:
   int isspace (char c);
```

```
#include <stdio.h>
#include <ctype.h>

char Zeichen1 = ' ';
char Zeichen2 = 'A';

void main (void)
{
   if (isspace (Zeichen1) > 0)
      printf ("\nZeichen 1 ist Leerzeichen!");
   if (isspace (Zeichen2) > 0)
      printf ("\nZeichen 2 ist Leerzeichen!");
}
```

Die Funktion *isspace* gibt einen Wert ungleich Null zurück, wenn das Zeichen *c* aus dem Bereich der Leerzeichen (Leerzeichen, Tabulator, Zeilenvorschub usw.) stammt. Andernfalls wird der Wert Null zurückgegeben.

16.4.8. IN KLEINSCHRIFT UMSETZEN

```
Syntax:
   int tolower (char c);
```

```
#include <stdio.h>
#include <ctype.h>

char Zeichen = "A";

void main (void)
{
   printf ("\nZeichen vorher: %c", Zeichen);
   tolower (Zeichen);
   printf ("\nZeichen nachher: %c", Zeichen);
}
```

Gehört das übergebene Zeichen *c* zu den großen Buchstaben („A-Z“), so wird es mit *tolower* in den entsprechenden kleinen Buchstaben („a-z“)

umgewandelt und zurückgegeben. Sonderzeichen (dazu gehören auch die Umlaute und das „ß“) bleiben unverändert. Die folgende Funktion behebt jedoch dieses Defizit:

```
//============================================================
// Die deutschen Programmvarianten sind in der Datei
// STRUTIL.CPP zusammengefasst, die zugehörige Headerdatei
// STRUTIL.H deklariert die Prototypen.
// STRUTILH.CPP zeigt Beispielaufrufe der Funktionen
//============================================================

#include <ctype.h>

int gertolower (int c)
{
   switch (c)
   {
      case 'Ä': return ('a');
      case 'Ö': return ('ö');
      case 'Ü': return ('ü');
      case 'ß': return ('ß');
      default : return (tolower(c));
   }
}
```

16.4.9. IN GROSSSCHRIFT UMSETZEN

```
Syntax:
   int toupper (char c);
```

```
#include <stdio.h>
#include <ctype.h>

char Zeichen = "a";

void main (void)
{
   printf ("\nZeichen vorher: %c", Zeichen);
   toupper (Zeichen);
   printf ("\nZeichen nachher: %c", Zeichen);
}
```

Gehört das übergebene Zeichen *c* zu den kleinen Buchstaben („a-z“), so wird es mit *toupper* in den entsprechenden großen Buchstaben („A-Z“) umgewandelt und zurückgegeben. Sonderzeichen (dazu gehören auch die Umlaute und das „ß“) bleiben unverändert. Die folgende Funktion behebt jedoch dieses Defizit:

```
//=========================================================
// Die deutschen Programmvarianten sind in der Datei
// STRUTIL.CPP zusammengefasst, die zugehörige Headerdatei
// STRUTIL.H deklariert die Prototypen.
// STRUTILH.CPP zeigt Beispielaufrufe der Funktionen
//=========================================================

#include <ctype.h>

int gertoupper (int c)
{
   switch (c)
   {
      case 'ä': return ('Ä');
      case 'ö': return ('Ö');
      case 'ü': return ('Ü');
      case 'ß': return ('ß');
      default : return (toupper(c));
   }
}
```

17. STRINGS UND STREAMS

Genauso wie, wie es in ANSI-C eine Parallele zwischen *fprintf* und *sprintf* bzw. anderen Ein-/Ausgabefunktionen gibt, existieren in C++ drei Klassen, die einen Stream-Zugriff auf Zeichenketten ermöglichen – *istrstream*, *ostrstream* und *strstream* (analog zu *ifstream*, *ofstream* und *fstream*). Alle drei Klassen werden in der Headerdatei *strstrea.h* deklariert. Zeichenketten, auf die über einen *istrstream* geöffnet werden, sind reine Eingabeströme (genau wie *istream*). Entsprechend sind *ostrstream*-Objekte reine Ausgabeströme (wie *ostream*). Ziel und Quelle sind aber nicht Tastatur und Bildschirm sondern die mit den Streams verbundenen Zeichenketten. Die Klasse *strstream* hingegen vereinigt die Eigenschaften und Methoden beider Streamklassen. Analog zu den Datei- und Standardstreams sind die Stringstreams von der Basisklasse (*ios*) abgeleitet, erben also deren Methoden und implementieren die gleichen Operatoren. So erfolgt auch hier die Ein- bzw. Ausgabe mit den Operatoren >> und <<, jeweils bezogen auf die aktuelle Position innerhalb der Zeichenkette.

Da die Handhabung und die meisten Methoden bereits bekannt sind, stehen daher in den folgenden Abschnitten die Erweiterungen und Abweichungen im Vordergrund.

17.1. VERBINDEN DES STREAMS MIT EINER ZEICHENKETTE

Genau wie ein Dateistreams mit einer Datei, müssen die Stringstreams mit einer Quelle oder einem Ziel (also einer Zeichenkette) verbunden werden. Dies erfolgt meist direkt im Constructor des Streams:

```
Syntax (Constructoren):

ostrstream streamvar (void);
istrstream streamvar (void);
strstream streamvar  (void);

ostrstream streamvar (char *string, int size,
                      int mode);
istrstream streamvar (char *string, int size,
                      int mode);
strstream streamvar  (char *string, int size,
                      int mode);

ostrstream streamvar (char *string, int size);
istrstream streamvar (char *string, int size);
```

Der erste Constructor erzeugt jeweils ein Streamobjekt, ohne dieses mit einer Zeichenkette zu verbinden. Stattdessen wird der benötigte Platz für die Zeichenkette dynamisch verwaltet und belegt immer genau die Größe, die gerade gebraucht wird. Um die Freigabe des dynamisch

belegten Speichers muss sich der Entwickler nicht kümmern, da dies automatisch im Destructor erfolgt.

Der zweite Constructor benötigt als Parameter die Angabe eines Puffers (Zeichenkette) und der Länge des Puffers. Handelt es sich bei dem Stringpuffer um einen dynamischen Speicherbereich (mit *new* erzeugt, siehe dazu auch Abschnitt über dynamische Speicherverwaltung), dann ist der Entwickler selbst dafür verantwortlich, den belegten Speicher (mit *delete*) wieder freizugeben. Zusätzlich müssen/können Modusangaben erfolgen.

Die dritte Constructorform nutzt lediglich die Vorbelegung des Modus für die *ostrstream*- und *istrstream*-Klasse aus. Für die *strstream*-Klasse gibt es keine Vorbelegung, bei ihr muss der Modus immer mit angegeben werden (es sei denn, man verwendet die dynamische, erste Constructorform).

Modusangaben für Stringstreams		
Modus	Flagwert	Bedeutung
ios::in	1	Aus der Zeichenkette wird nur gelesen, Voreinstellung für istrstream-Objekte
ios::out	2	In die Zeichenkette wird nur geschrieben, Voreinstellung für ostrstream-Objekte
ios::ate	4	Streampointer wird zu Beginn an das Ende der Zeichenkette (Nullbyte) positioniert, kann aber später überall stehen
ios::app	8	Beim Schreiben werden alle neuen Daten an das Stringende angehängt (append)

Tabelle 17-1: Stringstream-Modi

Die mit Abstand häufigste Anwendungsform ist die Verwendung von *strstream*-Objekten, da diese Ein- und Ausgaben zulassen. Die folgenden Beispiele beschränken sich daher auf diese Streamvariante:

```
#include <iostream>
#include <strstream>

using namespace std;

void main (void)
{
   int x = 0;
   char buff [300] = "";

   strstream stream1 (buff, 300 ios::in | ios::out);
   strstream stream2;
}
```

17.2. STRINGSTREAM-METHODEN

Um das Stringhandling von ANSI-C abzulösen ist auf den Zeichenketten-Streams eine ganze Reihe von Methoden definiert, die in den folgenden Abschnitten kurz behandelt werden.

17.2.1. FEHLERSTATUS ABFRAGEN

```
Syntax:
   int streamvar.fail (void);
   int streamvar.good (void);
   int streamvar.bad (void);
```

Die Methode *fail* gibt darüber Auskunft, ob eine Streamoperation erfolgreich war. Trat ein Fehler auf, so gibt *fail* einen Wert ungleich Null zurück. Der Zurückgegebene Wert entspricht dann dem Zustand der Bits *ios::failbit*, *ios::badbit* und *ios::hardfail* aus der Membervariable *ios::state* (Member der Basisklasse *ios*).

```
#include <iostream>
#include <strstream>

using namespace std;

void main (void)
{
   strstream mystream;

   if (mystream.fail())
   {
      cout << "Stringbefehl fehlgeschlagen" << endl;
      return;
   }
}
```

Die Methode *bad* gibt einen Wert ungleich Null zurück, falls bei der Abfrage von *ios::badbit* oder *ios::hardfail* in *ios::state* ein Fehler erkannt wird.

```
#include <iostream>
#include <strstream>

using namespace std;

void main (void)
{
   strstream mystream;

   if (mystream.bad ())
   {
```

```
      cout << "Fehler bei Stringoperation" << endl;
      return;
   }
}
```

Die Umkehrung zu *bad*, die Methode *good*, liefert einen Wert ungleich Null zurück, falls keines der Statusbits in *ios::state* gesetzt, also kein Fehler aufgetreten ist.

```
#include <iostream>
#include <strstream>

using namespace std;

void main (void)
{
   strstream mystream;

   if (!mystream.good ())
   {
      cout << "Fehler bei Stringoperation" << endl;
      return;
   }
}
```

17.2.2. BUFFERANWEISUNGEN

Die Methode *flush* schreibt alle noch in den Puffern befindlichen Daten in die zugehörigen Streamobjekte.

```
Syntax:
   char *streamvar.rdbuf (void);
   void streamvar.flush (void);
```

```
//=============================================================
// Programm SSTRBUFF.CPP
//=============================================================

#include <iostream>
#include <strstream>

using namespace std;

void main (void)
{
   strstream mystream;

   cout << "Bufferadresse:" << (long)mystream.rdbuf() << endl;
}
```

17.2.3. FORMATANWEISUNGEN

Für die Stream-IO gibt es eine ganze Reihe von Methoden, die bereits im Abschnitt über die Standard-IO abgehandelt wurden und dort im Detail nachzulesen sind. Es handelt sich dabei um die folgenden Methoden:

```
Syntax:
   long streamvar.flags (long flagstoset);
   long streamvar.flags (void);

   long streamvar.setf (long ios::flagname);
   long streamvar.setf (long ios::flagname, long
                        ios::flaggruppe);

   long streamvar.unsetf (long ios::flagname);

   int streamvar.fill  ();
   int ostreamvar.fill (char Füllzeichen);

   int streamvar.width ();
   int streamvar.width (int Anzahlzeichen);

   int streamvar.flush ();

   int streamvar.precision ();
   int streamvar.precision (int Anzahlzeichen);
```

Diese Methoden beeinflussen die Flags des Streams, mit denen die Ein- und Ausgabeeigenschaften des Streamobjektes festgelegt werden. Eine Liste der beeinflussbaren Flags ist im Kapitel über die IO-Streams zu finden. Die Handhabung der Methoden ist identisch mit dem Handling für die Klassen *ostream* und *istream*.

Natürlich werden nicht nur die gleichen Methoden vererbt, sondern auch die Manipulatoren (u.a. *setprecision*, *setw*, *setfill*, *endl*, *ends*).

```
//=============================================================
// Programm SSTRMANI.CPP
//=============================================================

#include <iostream>
#include <strstream>
#include <iomanip>

using namespace std;

void main (void)
{
   char buff [300] = "";
   strstream mystream;

   // in den Stream schreiben
```

```
   cout << "Zahl eingeben: ";
   cin.getline (buff, 300);

   // Im Stream formatiert zusammensetzen
   mystream << setfill('*') << setw(10) << buff << ends;

   // Aus dem Stream lesen
   cout << mystream.str ();
}
```

17.3. EIN- UND AUSGABEANWEISUNGEN

Neben den Operatoren << und >> existieren bei der Stringstream-IO die gleichen Ein- und Ausgabeanweisungen wie bei der Stream-IO.

17.3.1. AUSGABE MIT STRINGSTREAMS

Die Methode *put* schreibt ein einzelnes Zeichen in den Ausgabestrom, wobei die Methode *put* hat den Vorteil hat, dass mit dieser auch Steuerzeichen an den Ausgabestrom gegeben werden können.

Die Methode *write* hingegen dient zur Ausgabe von Zeichenketten unter Angabe einer Ausgabelänge.

```
Syntax:
   streamtyp& streamvar.put (char Zeichen);
   streamtyp& streamvar.put (signed char Zeichen);
   streamtyp& streamvar.put (unsigned char Zeichen);

   streamtyp& write (const char *Zeichenkette,
                     int Anzahlzeichen);
   streamtyp& write (const signed char *String,
                     int Anzahlzeichen);
   streamtyp& write (const unsigned char *String,
                     int Anzahlzeichen);
```

Besonders anzumerken ist, dass die Funktion *write* die angegebene Ausgabelänge auf jeden Fall einhält. Ist die Zeichenkette länger, dann ist die Sache recht unproblematisch, denn es wird nur der Anfang der Zeichenkette (entsprechend der angegebenen Länge) ausgegeben. Ist die Zeichenkette jedoch kürzer, so kümmert sich der C++-Compiler sich nicht um das Stringendesymbol *'\0'* (siehe auch Abschnitt über Zeichenketten), sondern gibt immer die angegebene Länge an Zeichen aus – d.h. zumeist jede Menge Unsinn.

```
#include <iostream>
#include <strstream>

using namespace std;
```

```
void main (void)
{
   strstream mystream;

   mystream.put ('\t');
   mystream.write ("Hallo", 5);
}
```

17.3.2. EINGABE MIT STRINGSTREAMS

Von der Methode *get* gibt es mehrere Varianten (die Methode ist also mehrfach „überladen“), jede mit einer leicht anderen Handhabung und unterschiedlichen Parameterleisten. Die Methode *get* akzeptiert, im Gegensatz zum Operator >> auch Leerzeichen als Eingaben.

```
Syntax:
   int streamvar.get ();
```

Die einfachste Form der *get*-Methode liest das nächste Zeichen aus dem Eingabestrom und gibt dessen ASCII-Wert zurück. Da ASCII-Wert und Character sich in ANSI-C und C++ nicht unterscheiden, kann man das Ergebnis entweder als Zahl oder als gelesenes Zeichen betrachten.

```
Syntax:
   streamtyp& streamvar.get (char& charvar);
```

Auch diese Form der *get*-Methode liest nur das nächste Zeichen aus dem Eingabestrom. Das Zeichen selbst wird jedoch nicht zurückgegeben, sondern auf die Variable geschrieben, die als Parameter übergeben wurde.

```
Syntax:
   streamtyp& streamvar.get (char* buf, int len,
                             char Delim = '\n');
```

Diese Form des *get* ist dazu geeignet Zeichenketten einzulesen. Dem Eingabestrom werden solange Zeichen entnommen und in der Puffervariablen (*buf*) abgelegt, bis die Methode entweder auf das angegebene Endzeichen (*Delim*) oder ein Dateiendezeichen stößt. Es werden jedoch niemals mehr als *len-1* Zeichen in die Puffervariable übernommen. Die *get*-Methode sorgt zudem dafür, dass stets ein Stringende-Zeichen (*'\0'*) in die Puffervariable geschrieben wird. Der Parameter *Delim* (Delimiter) muss nicht angegeben werden, wenn es sich um die Return-Taste handeln soll, da dies ist die Voreinstellung ist (siehe auch Abschnitt über Default-Parameter).

Die Methode *getline* ist identisch mit der *get*-Variante, die in der Lage ist Zeichenketten zu lesen. Ein Unterschied zwischen *getline* und der angesprochenen *get*-Methode besteht nicht. Die Verwendung von *getline* ist trotzdem vorzuziehen, da sich die Bedeutung im Unterschied zum einfachen *get* leichter erschließt.

```
Syntax:
   streamtyp& istreamvar.getline (char* buf,int len,
                                  char Delim='\n');
```

Die Methode *getline* dient, wie bereits erwähnt, dazu Zeichenketten einzulesen. Dem Eingabestrom werden solange Zeichen entnommen und in der Puffervariablen (*buf*) abgelegt, bis die Methode entweder auf das angegebene Endzeichen (*Delim*) oder ein Dateiendezeichen stößt. Es wird jedoch niemals mehr als *len-1* Zeichen in die Puffervariable übernommen. Die *getline*-Methode sorgt zudem dafür, dass stets ein Stringende-Zeichen (*'\0'*) in die Puffervariable geschrieben wird. Der Parameter *Delim* (Delimiter) muss nicht angegeben werden, wenn es sich um die Returntaste handeln soll, da dies ist die Voreinstellung ist.

Die Methode *peek* dient dazu das nächste Zeichen zu betrachten, ohne es aus dem Datenstrom zu entnehmen, d.h. das Zeichen bleibt im Eingabepuffer (meist der Tastaturpuffer) und wird erst beim nächsten *get* oder *getline* entnommen.

```
Syntax:
   int istreamvar.peek ();
```

Die Methode *putback* hat die Aufgabe ein bereits auf dem Stream gelesenes (oder ein beliebig anderes) Zeichen wieder in diesen zurückzustellen (oder hinzuzufügen).

```
Syntax:
   streamtyp & istreamvar.putback(char);
```

```
//=============================================================
// Programm SSTRPUTB.CPP
//=============================================================

#include <iostream>
#include <strstream>

using namespace std;

void main (void)
{
   int c = 0;
```

```
   char buff [300] = "Hallo";
   strstream mystream;

   mystream << buff;
   c = mystream.get ();
   mystream.putback ((char)c);
   cout << c << endl;
   c = mystream.peek ();
   cout << c << endl;

   mystream.getline (buff, 300);
   cout << buff << endl;
}
```

17.4. POSITIONIERUNG IN STREAMS

Bereits die Klassen *istream* und *ostream* beinhalten Methoden zur Positionierung eines Zeigers im Stream. Diese Positionierungsmethoden werden an die Klassen *istrstream*, *ostrstream* und *strstream* vererbt (siehe auch Abschnitt über Vererbung).

Im Gegensatz zu ANSI-C, dass für diese Aufgabe nur die Funktionen *seek* und *tell* besitzt, beinhalten die Streamobjekte unterschiedliche Methoden für Ein- und Ausgabeströme (*seekg* und *tellg* für die Eingabe – das „*g*“ steht für *get* – und *seekp* und *tellp* für die Ausgabe – das „*p*“ steht für *put*).

Die Unterteilung ist durchaus sinnvoll, da man sich vorstellen kann, dass man den Lese- und die Schreibzeiger unabhängig voneinander im Stream bewegen kann (so z.B. in der Standardklasse *strstream*). Die Methoden *seekg* und *seekp* setzen den Streampointer auf die angegebene, relative Position.

```
Syntax:
   istream& seekg (long position);
   istream& seekg (long offset, int relativzu);

   ostream& seekp (long position);
   ostream& seekp (long offset, int relativzu);
```

Zusätzlich kann man angeben, von wo aus die Verschiebung erfolgen soll (analog zu *seek* in C). Wird kein Positionierungsmodus angegeben, so erfolgt die Verschiebung relativ zum Streamanfang. Im Abschnitt über Dateien sind die erlaubten Schlüsselworte für die Verschiebung aufgeführt.

Die Methoden *tellg* und *tellp* wiederum geben die aktuelle Position (relativ zum Streamanfang) zurück:

```
Syntax:
   long tellg (void);
   long tellp (void);
```

```
//=============================================================
// Programm SSTRTELL.CPP
//=============================================================

#include <iostream>
#include <strstream>

using namespace std;

void main (void)
{
   int  zahl  = 0;
   char b    [100] = "";
   char buff [300] = "";
   strstream mystream (buff, 300, ios::in | ios::out);

   mystream << "ABCDEFGHIJKLMNOP" << ends;

   // erste Zeile lesen
   mystream.seekp (4);
   mystream << "1234";
   cout << buff << endl;

   mystream.seekg (4);
   mystream >> zahl;
   cout << zahl << endl;

   mystream.seekp (10);
   mystream << "12";

   mystream.seekg (10);
   mystream >> zahl;
   cout << zahl << endl;
}
```

Tritt bei der Positionierung ein Fehler auf, so wird der Streamstatus entsprechend gesetzt und kann bei Bedarf über die Methoden *fail*, *bad* und *good* (s.o.) abgefragt werden. Die Handhabung der Positionierung innerhalb eines Stringstreams ist allerdings etwas gewöhnungsbedürftig, da z.B. der Status, ob ein bestimmt5er Teil des Streams bereits gelesen wurde (entscheidend für die *getline*-Methode) nicht von der Position des Streampointers abhängig ist.

17.5. AUFGABENTEIL

Bei allen Aufgaben ist ein Schwierigkeitsgrad angegeben. Der Schwierigkeitsgrad bezieht sich nicht allein auf die Aufgabenstellung sondern ggf. auch auf den Umfang der Ausgabe.

17.5.1. AUFGABE 1 (MITTEL)

Schreiben Sie ein Programm, das 10 Vornamen und 10 Nachnamen auf jeweils einem Feld von Zeichenketten einliest. Sortieren Sie die Zeichenketten nach Nachnamen anschließend in den Feldern. Beachten Sie bitte, dass bei gleichen Nachnamen der Vorname den Ausschlag bei der Reihenfolge gibt. Verwenden Sie weitestgehend Funktionen und geben Sie die Namen am Ende sortiert wieder aus.

17.5.2. AUFGABE 2 (SCHWIERIG)

Schreiben Sie ein Programm, das 10 Vornamen und 10 Nachnamen auf jeweils einem Feld von Zeichenketten einliest. Sortieren Sie die Zeichenketten nach Länge der Nachnamen anschließend in den Feldern. Beachten Sie bitte, dass bei gleichen Nachnamenlängen die alphabetische Reihenfolge den Ausschlag bei der Sortierung gibt. Verwenden Sie weitestgehend Funktionen und geben Sie die Namen am Ende sortiert wieder aus.

18. ZUSAMMENGESETZTE DATENTYPEN (KOMPOSITTYPEN)

In C/C++ besteht, wie in den meisten anderen Programmiersprachen auch, die Möglichkeit, Variablen verschiedener Typen zu einem Datensatz zusammenzufassen.

18.1. STRUKTUREN

Ein aus anderen Datentypen zusammengesetzter Datensatz kann als neuer Variablentyp aufgefasst und entsprechend im Programm verwendet werden. Ein solcher komplexer Datentyp, der sich von den Feldern dadurch unterscheidet, dass er einfache und komplexe Variablen verschiedener Typen enthalten kann, wird in den meisten Programmiersprachen „record“ genannt. In C/C++ hingegen heißt er Struktur und wird im Deklarationsteil mit dem Schlüsselwort *struct* eingeleitet.

```
struct Datum {int Tag; char Monat [4]; int Jahr;}
       EingabeDatum = {1, "Jan", 1994};

struct Datum Heute = {24, "Jan", 1994};

struct {int Tag; char Monat [4]; int Jahr;} Versuch;
```

In der ersten Zeile des Beispiels wird eine Struktur vom Typ *Datum* erzeugt. Die Angabe des Namens *Datum* vor der geschweiften Klammer reserviert aber noch keinen Speicherplatz sondern wird nur dazu benutzt, um den Typ der Struktur (d.h. ihre Zusammensetzung) mit einem Namen umschreiben zu können.

In der geschweiften Klammer sind die Einzelelemente (Komponenten) aufgelistet, aus denen sich die Struktur zusammensetzt. Erst die Angabe eines Variablennamens nach der geschweiften Klammer (hier *EingabeDatum*) reserviert den notwendigen Speicherplatz, so dass auch tatsächlich Werte gesichert werden können. Der speicherinterne Aufbau der oben gezeigten Struktur vom Typ *Datum* wird aus der nachstehenden Tabelle ersichtlich.

Aufbau einer Struktur				
Adresse	Inhalt	Namen	Typ	Offset
keine, nur interner Name	1000	Eingabedatum	Adresse der Struktur vom Typ Datum	keiner
1000	17	EingabeDatum.Tag	Integer	0
1002	"Jan"	EingabeDatum.Monat	Array of char (String)	2 (int)
1006	1996	EingabeDatum.Jahr	Integer	6 (int + string[4])

Tabelle 18-1: Beispiel zum Strukturaufbau

Wie man leicht erkennen kann, werden die einzelnen Komponenten vom C/C++ Compiler in Offsets umgewandelt, die beim Ansprechen der

Strukturteile automatisch zur Anfangsadresse der Struktur (hier 1000) hinzuaddiert werden.

In der zweiten Zeile des Beispiels wird eine weitere Variable vom Typ *Datum* erzeugt (die Variable *Heute*). Da der Typ *Datum* im Programm bereits bekannt ist, braucht seine Zusammensetzung nicht erneut in geschweiften Klammern beschrieben werden.

Sowohl die Angabe der Typbezeichnung wie auch die Vereinbarung des Variablennamens in der ersten Zeile des Beispiels (also *Datum* und *EingabeDatum*) sind optional, nur eines von beiden ist jeweils notwendig. Entfällt in der Deklaration die Typbezeichnung (*Datum*), so würde die zweite Zeile (die Deklaration von *Heute*) des Beispiels einen Fehler erzeugen, da der Typ *Datum* nicht vereinbart wurde. Anders bei der Variablen *Versuch*.

Diese Variable wird korrekt (entsprechend der angegebenen Struktur) angelegt. Syntaktisch jedoch ist *Versuch*, trotz des identischen Aufbaus zu Variablen des Typs *Datum* ein einmaliges Objekt (sie ist ja ausdrücklich nicht vom Typ *Datum*), d.h. keine andere Variable kann vom gleichen Typ sein. Da C/C++ nur typisierte Pointer verarbeiten kann, sind auch Zeiger auf ein solches Objekt nicht möglich.

Soll eine Struktur an eine Funktion weitergereicht werden, so ist eine Typisierung ohnehin unumgänglich. Entfällt im ersten Beispiel der Variablenname (*EingabeDatum*), so bedeutet dies, dass zwar der grundlegende Aufbau der Struktur vereinbart wurde, aber noch kein Speicherplatz für eine Variable dieses Typs angefordert wurde. Letzteres ist der übliche Weg, um eigene Strukturtypen in Headerdateien zu vereinbaren und in Bibliotheken zur Verfügung zu stellen.

Für die Definition von Pointern auf eine Struktur ist, wie bereits angedeutet, eine Typbezeichnung unbedingt notwendig. Dazu einige Beispiele:

```
//-------------------------------------------------------------
// Typische Deklaration: Typname {Struktur} Varname
//-------------------------------------------------------------
struct Datum {int Tag; char Monat [4]; int Jahr;}
       EingabeDatum;

//-------------------------------------------------------------
// Kein Typname, Struktur ist daher einmaliges Objekt
//-------------------------------------------------------------
struct {int xpos; int ypos;} Cursorposition;

//-------------------------------------------------------------
// Keine Strukturangabe, da Typname schon bekannt ist
//-------------------------------------------------------------
struct Datum Morgen;
```

```
//-------------------------------------------------------------
// Datensatzdeklaration, ohne Variablendeklaration Deklaration
// vom Variablentyp KOORDINATEN erfolgt im weiteren Verlauf
// des Programms.
// Es können auch Pointer auf Datensätze vereinbart werden
//-------------------------------------------------------------
struct Koordinaten {float xpos; float ypos;};

struct Koordinaten Position;  // Variablendeklaration
struct Koordinaten *Actual;   // Pointer a. Datensatz

void main (void)
{
}
```

Die doppelte Verwendung der Komponentennamen *xpos* und *ypos* ist in Strukturen ohne Belang, da diese Komponenten nur zusammen mit dem übergeordneten Variablennamen der Struktur angesprochen werden können. Dadurch bleibt die Eindeutigkeit der Namen erhalten. Innerhalb einer Struktur müssen Namen natürlich einmalig sein, um den Zugriff zu erlauben. Als Auswahloperator (Komponentenauswahl) dient in C/C++ der Dezimalpunkt:

```
//=============================================================
// Programm STRUCT1.CPP
//=============================================================

struct Koordinaten {float xpos; float ypos;} Position;
struct {signed int xpos; signed int ypos;} Cursorposition;

void main (void)
{
   Cursorposition.xpos = 17;
   Position.xpos = 181.4;
}
```

Die Strukturen in C/C++ besitzen gegenüber den Feldern und den Strings einen erheblichen Vorteil: man kann sie einfach aufeinander zuweisen.

```
//=============================================================
// Programm STRUCT2.CPP
//=============================================================

struct Koordinaten {float xpos; float ypos;};
struct Koordinaten Neu, Alt;
```

```
void main (void)
{
   Alt.xpos = 17;
   Alt.ypos = 181.4;
   Neu = Alt;
}
```

Beim Zuweisen zweier Strukturen aufeinander werden die Speicherinhalte Bit für Bit über ihre Anfangszeiger aufeinander kopiert (*memcpy*. Die bedeutet, dass auch die ggf. hinter einem Stringendezeichen liegenden Zeichen (bis zur angegebenen Maximallänge der Zeichenkette) mit kopiert werden. Dies gilt natürlich auch für die Parameterübergabe an Funktionen. Es ist zu beachten, dass die Übergabe über einen Pointer auf die Struktur geschehen muss, wenn die Struktur innerhalb der Funktion verändert werden soll. Die Standard-Übergabeform bei Strukturen ist demnach der Call by Value (im Gegensatz zur Übergabe von Arrays und Zeichenketten, die standardmäßig Call by Pointer übergeben werden).

Diese Form der Duplizierung lässt sich durch die Bibliotheksfunktion *memcpy* auch per Hand erreichen (in C++ ist dies für Klassen durch geschicktes Überladen des Zuweisungsoperators – siehe Abschnitt über Klassen – besser gelöst):

```
memcpy (Ziel, Quelle, sizeof(Datum));
```

Ein Vergleich zweier Strukturen über den Vergleichsoperator ist jedoch im ANSI-Standard nicht vorgesehen und daher auch nicht möglich.

```
#include <iostream>

using namespace std;

struct Koordinaten {float xpos; float ypos;};
struct Koordinaten Neu, Alt;

void main (void)
{
   Alt.xpos = 17;
   Alt.ypos = 181.4;
   Neu = Alt;
   if (Neu == Alt)
      cout << "Ja, sind gleich";
}
```

18.1.1. POINTER AUF STRUKTUREN

Der Zugriff über Zeiger auf eine Strukturkomponente gestaltet sich ein wenig anders als der Zugriff über den Komponentenselektor (Dezimalpunkt). Ein Zugriff über Zeiger ist spätestens dann notwendig, wenn eine Struktur an eine Funktion übergeben und dort verändert

werden soll. Da eine Zuweisung von Strukturen aufeinander möglich ist, erfolgt die Übergabe von Strukturen an Unterprogramm (anders als bei Zeichenketten und Feldern) normalerweise über einen Call by Value. Soll also eine Struktur in einem Unterprogramm verändert — z.B. eingelesen — werden, muss ausdrücklich die Adresse der Struktur übergeben werden. Hierbei wird mittels des Operators „->„ der Verweischarakter des Pointers deutlich gemacht:

```
//=============================================================
// Programm STRCTPTR.CPP
//=============================================================

#include <iostream>

float *Verweis_auf_float;
struct Datum {int Tag; char Monat [4]; int Jahr;}
       EingabeDatum;
struct Datum Heute = {24, "Jan", 1994};
struct Koordinaten {float xpos; float ypos;} Position;

void fnDatCopy (struct Datum *Quelle, struct Datum *Ziel)
{
   Ziel->Tag = Quelle->Tag;
   strcpy (Ziel->Monat, Quelle->Monat);
   Ziel->Jahr = Quelle->Jahr;
}

void main (void)
{
   struct Koordinaten *Actual;

   Actual = &Position;
   Actual->ypos = 222.22;
   Verweis_auf_float = &(Actual->ypos);
   cout << "Datum wird kopiert";
   fnDatCopy (&Heute, &EingabeDatum);
}
```

18.1.2. VERSCHACHTELUNG VON STRUKTUREN

Alle Strukturen lassen sich natürlich auch schachteln bzw. zu Vektoren (Feldern, Arrays) zusammenfassen und initialisieren:

```
#include <string.h>

struct Datum {int Tag; char Monat [4]; int Jahr;};

struct Person {char Name [20]; char VName [20];
               struct Datum Geburtstag;} Leute [3] =
              {
                 {"Nachname", "Vorname", {1, "Jan", "00"}};
                 {"Nachname", "Vorname", {1, "Jan", "00"}};
                 {"Nachname", "Vorname", {1, "Jan", "00"}}
              };
```

```
void main (void)
{
}
```

18.1.3. STRUKTURGRÖSSE ERMITTELN

Strukturen werden, wie bereits gezeigt, grundsätzlich über ihre Adresse an Funktionen übergeben (Call by Pointer). Ihre Größe (in Byte) kann mittels des Operators *sizeof ()* festgestellt werden:

```
int groesse = 0;
groesse     = sizeof (Person);
```

Der Operator *sizeof ()* kann, zusammen mit der Möglichkeit verkettete Strukturen zu erzeugen (also Strukturen die einen Pointer auf ihren eigenen Typ beinhalten), dazu benutzt werden, dynamische Listen zu erzeugen. Die Grundlagen der dynamischen Allokation von Speicherplatz werden im Kapitel über „dynamische Listen“ beschrieben.

18.2. UNIONS

Der Datentyp *union* realisiert in C/C++ die „variante“ Struktur. D.h. eine *union* ist eine Variable, die Werte verschiedener Typen auf dem gleichen Speicherbereich unterbringen kann. Eine „variante“ Struktur hat in C/C++ die folgende, allgemeine Form:

```
union u_typ {int ival; float fval;} u_var;
```

Wie sehr leicht zu erkennen ist, ähnelt die *union* im Aufbau einer Struktur (*struct*). Tatsächlich sind beide Formen sehr eng verwandt und sind auch in nahezu identischer Weise zu verwenden. In obigem Beispiel enthält die Variable *u_var* aber nicht zwei Komponenten mit den Namen *ival* und *fval*, wie es bei einer Struktur der Fall wäre, sondern immer nur eine von beiden. Die Variable *u_var* enthält also entweder *ival* oder *fval*. Die Komponenten einer *union* haben daher, jede für sich, die relative Position 0 zur Basisadresse der Variablen. Eine „variante“ Struktur belegt dabei genau den Speicherplatz, der nötig ist, um den größten der angegebenen Variablentypen zu speichern.

Wird eine *union* ausgewertet, d.h. der darin gespeicherte Wert ermittelt, so muss der Programmierer genauestens darauf achten, diesen auf eine Variable des entsprechenden Typs zuzuweisen. Wird im obigen Beispiel eine Integerzahl gespeichert, so kann das dann in der Variable enthaltene Bitmuster dennoch als Fließkommazahl wieder ausgelesen werden. Die Ergebnisse sind dann allerdings eher zufällig, da das Speichern einer Integerzahl nur die ersten Zwei der insgesamt 6 Bytes überschreibt, die restlichen bleiben unberührt.

C/C++ bietet gegen diesen Effekt weder eine Laufzeitüberprüfung noch irgendeine andere Form der Unterstützung. Üblicherweise speichert man daher den Typ des zuletzt an die *union* zugewiesenen Wertes in einer Integervariablen. Für das hier gezeigte Beispiel könnte z.B. eine Variable mit dem Namen *u_inhalt* den Wert Null annehmen, wenn in *u_var* ein Integerwert gespeichert ist (bzw. den Wert Eins wenn es sich um eine Fließkommazahl handelt). Ein solches Vorgehen führt typischerweise zu Konstruktionen wie der folgenden:

```
//============================================================
// Programm UNION.CPP
//============================================================

#include <iostream>
#include <iomanip>
#include <float.h>
#include <conio.h>
#include <ctype.h>
#include <stdlib.h>

using namespace std;

#define UNIONINT 0
#define UNIONFLOAT 1

union u_typ {int ival; float fval;} u_var;

void main (void)
{
   int u_inhalt;
   int Eingabe   = '\0';

   cout << "Integer oder Float speichern (i/f)";
   Eingabe = toupper (getch ());
   cout << endl << "Wert : ";

   switch (Eingabe)
   {
      case 'I' : u_inhalt = UNIONINT;
                 cin >> u_var.ival;
                 break;
      case 'F' : u_inhalt = UNIONFLOAT;
                 cin >> u_var.fval;
                 break;
      default :  cout << "Falsche Eingabe!";
                 exit (10);
                 break;
   }

   if (u_inhalt == UNIONINT)
   {
      cout << "Int-Wert : " << u_var.ival << endl;
   }
```

```
   else
   {
      cout << "Float-Wert : " << u_var.fval << endl;
   }
}
```

Natürlich kann eine *union* über allen Datentypen aufgebaut werden, bzw. auch in komplexen Datentypen einsetzbar (es sind somit auch Vereinigungen von Strukturen definierbar). Es ist also durchaus möglich eine *union* zu definieren, die Teil einer Struktur oder einer anderen *union* ist.

```
union u_typ {int ival; float fval;};
struct uvtyp {int uinhalt; u_typ uvar;} BeidesUVar;
```

18.3. TYPERWEITERUNGEN

Der Befehl *typedef* erlaubt es das Schlüsselwort *struct* bzw. *union* (vor dem Typnamen der Struktur/Union) entfallen zu lassen. Dies ist in C++ nicht notwendig, wird ein Typname vergeben, so definiert der C++ Compiler diesen selbständig als eigenen Typ, so als hätte man eine *typedef*-Anweisung programmiert.

18.3.1. DIE TYPEDEF-ANWEISUNG

Das Schlüsselwort *typedef*, ermöglicht es dem Programmierer, den Programmtext lesbarer zu gestalten. Die *typedef*-Anweisung erlaubt die Zuweisung von zusätzlichen Variablentypnamen an neue oder schon bekannte Typen. So lässt sich z.B. der Verwendungszweck einer Variablen nicht nur an den Namen der Variablen selbst, sondern auch an deren Typ binden:

```
Syntax:
   typedef Typ NeuerTypname
```

```
//============================================================
// Programm TYPNAME.CPP
//============================================================

typedef int COUNTER;
typedef char *STRINGPTR;
typedef struct datumtyp {int Tag; char Monat [4];
                         int Jahr;} DATUM, *DATUMPTR;

void main (void)
{
   COUNTER i, j;
   STRINGPTR Name;
```

```
    DATUM Heute, Morgen, Gestern;
    DATUMPTR EinTag = &Heute;
}
```

Der Typ *COUNTER* ist nun synonym zu einer Integervariablen und *STRINGPTR* steht für einen Pointer auf Character. Die Variablenvereinbarungen werden dadurch wesentlich lesbarer. Mit *typedef* können keine neuen Datentypen geschaffen, sondern nur neue Namen für schon bekannte Datentypen vergeben werden, was z.B. das häufige und lästige Schreiben des Schlüsselwortes *struct* erspart, wenn viel mit Strukturen gearbeitet wird. In diesem Sinne ist *typedef* dem *#define* sehr ähnlich, da hier nur Synonyme, also Ersetzungstexte vereinbart werden. Aufgrund dieser inhaltlichen Verwandtschaft ist es üblich, die neudefinierten Typen, wie die symbolischen Konstanten, groß zu schreiben.

18.3.2. DER UNTERSCHIED ZWISCHEN TYPEDEF UND #DEFINE

Auf den ersten Blick könnte man meinen, *typedef* und *#define* seien in der Bedeutung gleich — schließlich sind die folgenden Anweisungen gleichwertig:

```
#define STRINGPTR char *
STRINGPTR s1;
```

```
typedef char * STRINGPTR
STRINGPTR s1;
```

Dies ist jedoch ein Irrtum, wie sich sehr schnell erweist, wenn nicht nur eine Variable deklariert wird, sondern mehrere:

```
// typedef char * STRINGPTR
STRINGPTR s1, s2; // = char * s1, *s2;
```

Das obere Beispiel wird durch die *#define*-Anweisung umgewandelt in:

```
// #define STRINGPTR char *
char * s1, s2;
```

also in einen Pointer auf char und einen char (!!!) — die typedef-Anweisung wandelt die Zeile korrekt in zwei Pointer auf char um.

19. STANDARD-FILEBEARBEITUNG

Die bisher vorgestellten Ein- und Ausgabeanweisungen haben sich immer auf die Standardeingabe (Tastatur) bzw. Standardausgabe (Bildschirm) bezogen. Um Programme und Daten langfristig speichern zu können ist jedoch die Verwaltung von Texten und Daten auf Festplatten oder anderen Massenspeichern eine notwendige Voraussetzung. C/C++ beinhaltet dementsprechend auch eine Vielzahl von Befehlen für die Einrichtung und Bearbeitung von Dateien.

19.1. ÖFFNEN UND SCHLIESSEN VON DATEIEN

Der erste Schritt bei der Verwendung von Dateien besteht immer darin, dem Programm mitzuteilen, welches *FILE* auf dem Massenspeichers zur Bearbeitung herangezogen werden soll. Dazu muss die entsprechende Datei mit Hilfe der Funktion *fopen* zunächst geöffnet werden. Die Standard-Dateifunktionen wie z.B. *fopen* und *fclose* werden im Headerfile <*stdio.h*> deklariert.

```
//============================================================
// Programm FILE1.CPP
//============================================================

#include <stdio.h>

void main (void)
{
   FILE *fp;
   char String [40];

   fp = fopen ("xyz.dat", "r");
   if (fp)
   {
      fgets (String, 40, fp);
      printf ("Aus Datei gelesen [%s]", String);
      fclose (fp);
   }
   else
   {
      printf ("Datei konnte nicht geöffnet werden");
   }
}
```

In der ersten Zeile des Hauptprogramms wird ein Pointer auf ein *FILE* (Filepointer) vereinbart, über den im weiteren Verlauf auf eine geöffnete Datei zugegriffen werden kann. Der Filepointer (im Beispiel *fp* genannt, er kann aber natürlich auch einen beliebigen anderen Namen haben) stellt einen logischen Dateinamen dar, an den durch die Funktion *fopen* verschiedene physikalische Dateinamen gebunden werden können (hier *"xyz.dat"*).

D.h. nichts anderes, als dass im Verlauf eines Programms verschiedene, auf dem Speichermedium existierende Dateinamen (mittels *fopen*) mit *fp* verbunden werden können. Aber natürlich nur einer zurzeit, bevor eine andere Datei geöffnet werden kann, muss die Datei, mit der *fp* gerade verbunden ist, erst mit *fclose* wieder geschlossen werden. Nach jedem Lesen oder Schreiben wird der Filepointer hinter das zuletzt gelesene bzw. geschriebene Zeichen gesetzt, d.h. er zeigt auf das nächste zu lesende bzw. zu schreibende Position.

Die ANSI-Standardfunktion *fopen* liefert einen Pointer auf *FILE* als Ergebnis zurück, der im Beispiel an den Filepointer *fp* zugewiesen wird. Somit wird die Position der Datei auf einem Datenträger (z.B. Diskette) ermittelt und festgehalten. Kann die angegebene Datei nicht geöffnet werden, z.B. weil es eine Datei des angegebenen Namens nicht gibt oder schon die Datei schon geöffnet ist, so wird ein Pointer auf *NULL* zurückgegeben. Der Standardzeiger *NULL* ist ebenfalls in *<stdio.h>* definiert und hat auch den Zahlenwert Null (Typ *long*), so dass logische Abfragen einfach zu realisieren sind.

Der Datentyp *FILE* ist eine Struktur, die die physikalische Position der Datei auf dem Massenspeicher enthält und die relative Schreib-/Leseposition zum Anfang der Datei verwaltet (den Filepointer). Der Filepointer selbst entspricht -rein technisch- fast immer in der Länge einer *long*-Zahl. Der erste Parameter von *fopen* (hier *"xyz.dat"*) ist der gewünschte physikalische Dateiname, also der Name unter dem die Daten auf dem Datenträger abgespeichert sind.

Der zweite Parameter (hier *"r"*) gibt an, was man mit dem *FILE* zu tun gedenkt (Modus). Die Zeichenkette *"r"* steht dabei für Read (= Lesen). Dies bedeutet, dass in der Datei *"xyz.dat"* zwar gelesen werden darf, aber keine Daten geändert, gelöscht oder hinzugefügt werden können. Neben *"r"* gibt es noch eine ganze Reihe weiterer Parameter, von denen hier nur die wichtigsten genannt werden (zumal diese auch von Betriebssystem zu Betriebssystem leicht unterschiedlich erweitert sein können):

Datei Öffnungsmodi		
"r"	READ	Lesen einer ASCII-Datei, Daten bleiben erhalten
"w"	WRITE	Schreiben einer ASCII-Datei, Daten werden gelöscht
"a"	APPEND	Anhängen von Daten an das Ende einer ASCII-Datei, Daten bleiben erhalten
"r+"	READ+	Lesen und Schreiben einer ASCII-Datei, Daten bleiben erhalten
"w+"	WRITE+	Schreiben und Lesen einer ASCII-Datei, Daten werden gelöscht
"a+"	APPEND+	Anhängen von Daten an das Ende einer ASCII-Datei, Daten bleiben erhalten

Tabelle 19-1: Öffnungsmodi einer Datei

Soll eine Datei zum Lesen geöffnet werden (*"r"*) und diese Datei existiert nicht, so kann *fopen* nicht durchgeführt werden und es wird ein Zeiger auf *NULL* zurückgegeben. Anders hingegen beim Parameter *"w"*. Ist eine Datei vorhanden, so wird diese neu angelegt - die bisher darin enthaltenen Daten gehen verloren. Ist die Datei noch nicht vorhanden, so wird sie erzeugt, d.h. im Anschluss an den Schreibvorgang ist die physikalische Datei auf dem Datenträger vorhanden.

Der Parameter *"a"* ähnelt dem Schreiben, nur wird hier der bereits vorhandene Inhalt der Datei nicht gelöscht. Alle neuen Daten werden an das Ende der Datei angehängt und das *FILE* somit vergrößert, nach dem Öffnen mit *fopen* steht der Filepointer automatisch am Dateiende. Ist die gewünschte Datei noch nicht vorhanden, so wird sie neu angelegt. Die Varianten mit dem angefügten „+“-Zeichen arbeiten ähnlich wie die Parameter ohne Pluszeichen, nur dass nach dem Öffnen sowohl Lesen als auch Schreiben erlaubt sind. Die Voraussetzung, z.B. dass bei *"r+"* eine Datei bereits vorhanden sein muss und dass bei *"w+"* der bisherige Inhalt gelöscht wird, ändern sich jedoch nicht. Der weitaus häufigste Weg ist das Vorhandensein einer Datei zu überprüfen - und wenn sie existiert mit *"r+"* zu öffnen. Existiert die Datei nicht, wird meist *"w+"* verwendet.

Will man *fp* im Verlauf des Programms auf eine andere zu bearbeitende Datei zeigen lassen, die Arbeit mit der Datei abschließen oder das Programm beenden, so muss man die geöffnete Datei (im Beispiel *"xyz.dat"*) zunächst schließen. Dieser Vorgang ist notwendig, da die Transferdaten, also gelesene und geschriebene Daten über Puffer- und Cachespeicher auf den Datenträger geschrieben werden, also über RAM-Speicherbereiche. Wird ein *FILE* nicht ordnungsgemäß geschlossen, so kann es passieren, dass die zuletzt übertragenen Daten noch nicht ordnungsgemäß auf dem Datenträger gespeichert wurden, sondern noch in Pufferbereichen zwischenlagern. Da viele Anwender die Angewohnheit haben, den Rechner direkt nach Beendigung eines Programmes auszuschalten (oder einen Warmstart auszulösen) kann es zu Datenverlusten kommen. Viele C/C++-Compiler schließen alle geöffneten Dateien am Ende des Programms automatisch, es gehört dennoch zum guten Ton unter C/C++-Programmierern, alle Dateien ordnungsgemäß abzuschließen.

Der Befehl *fclose* veranlasst dementsprechend die Übertragung der Daten vom Pufferspeicher auf den Datenträger und schließt die Datei. Wie leicht zu erkennen ist, verwendet auch *fclose* den logischen Dateinamen *fp*, so dass man nicht auf einen bestimmten, physikalischen Dateinamen festgelegt ist. Anschließend kann *fp* dazu genutzt werden um eine weitere Datei zu bearbeiten:

```
//============================================================
// Programm FILE2.CPP
//============================================================

#include <stdio.h>

void main (void)
{
   FILE *fp;
   char String [40];
   char Datname [] = "abc.txt";

   fp = fopen ("xyz.dat", "r");
   if (fp)
   {
      fgets (String, 40, fp);
      printf ("Aus Datei xyz.dat gelesen [%s]\n", String);
      fclose (fp);
   }
   else
   {
      printf ("Datei xyz.dat konnte nicht geöffnet werden");
   }

   fp = fopen (Datname, "a");
   if (fp)
   {
      fgets (String, 40, fp);
      printf ("Aus Datei %s gelesen [%s]", Datname, String);
      fclose (fp);
   }
   else
   {
      printf ("Datei %s konnte nicht geöffnet werden",
              Datname);
   }
}
```

Die folgende Funktion ist ein nützliches Programm, welches lediglich feststellt, ob eine angegebene Datei auf dem Massenspeicher existiert oder nicht. Es gibt zwar unter fast allen Betriebssystemen entsprechende DOS-Aufrufe (über Interrupts), diese sind jedoch logischerweise nicht unabhängig von der Hardware und dem Betriebssystem.

```
//============================================================
// Programm FILETEST.CPP
//============================================================

#include <stdio.h>

#define FALSE 0
#define TRUE  1

int Filetest (char *Dateiname)
```

```
{
   FILE *Datei;

   if ((Datei = fopen (Dateiname, "r")) == NULL)
   {
      return (FALSE);
   }
   else
   {
      fclose (Datei);
      return (TRUE);
   }
}

void main (void)
{
   if (Filetest ("c:\\autoexec.bat"))
   {
      printf ("Autoexec auf Platte c: gefunden!");
   }
   else
   {
      printf ("Autoexec nicht auf Platte c:!!!!!");
   }
}
```

Wie leicht zu erkennen ist, kann *fopen* auch einen String als Parameter verarbeiten, wodurch das Programm für beliebige Dateinamen verwendet werden kann. Im Allgemeinen ist es üblich, den Dateinamen einzulesen (z.B. mit *gets*), anstatt ihn im Programm fest vorzuschreiben.

19.2. DIE WICHTIGSTEN DATEIFUNKTIONEN

Auch die Ein- und Ausgabe von Daten auf einem Datenträger wird mit Hilfe von Funktionen gesteuert, die (fast alle) in <stdio.h> vereinbart sind. Die dazu verwendeten Standardfunktionen tragen fast die gleichen Namen, wie ihre Bildschirm- und Tastaturäquivalente, allerdings mit einem „f“ davor und einem Filepointer als zusätzlichem Parameter. Die Funktion *fprintf* hat also die gleichen Parameter wie *printf*, abgesehen von einem zusätzlichen Zeiger auf die gewünschte Datei.

Die Verwendung der wichtigsten Dateifunktionen *fopen* und *fclose* ist bereits beschrieben worden. Die Unterprogramme sollen aber dennoch in dieser Auflistung der wichtigsten Dateifunktionen nicht fehlen, sie sind, wie fast alle Dateifunktionen, in der Headerdatei <stdio.h> definiert.

Im folgenden Text bezieht sich der Begriff „Filepointer“ immer auf eine Variable vom Typ Pointer auf *FILE* (wie die im obigen Beispiel verwendeten Variablen *fp* und *Datei*), also auf eine geöffnete Datei. Neben den (wie bisher) selbstdefinierten Pointern gibt es auch eine Reihe

von vorbesetzten Standardfilepointern, auf die am Ende des Kapitels eingegangen wird.

19.2.1. DATEIEN ÖFFNEN MIT FOPEN

```
Syntax:
   FILE *fopen (char *String, char *String);
```

```
//============================================================
// Programm FILEOPEN.CPP
//============================================================

#include <stdio.h>

FILE *Datei = NULL;

void main (void)
{
   if ((Datei = fopen ("c:\\abc.dat","r"))==NULL)
   {
      puts ("Datei war nicht zu öffnen");
   }
   else
   {
      puts ("Datei ist offen");
      fclose (Datei);
   }
}
```

Die Funktion *fopen* öffnet eine Datei zum Lesen und/oder Schreiben. *fopen* liefert einen Zeiger auf den Dateiinhalt zurück, bzw. *NULL*, wenn die Datei nicht geöffnet werden konnte.

19.2.2. DATEIEN SCHLIESSEN MIT FCLOSE

```
Syntax:
   fclose (FILE *Filepointer);
```

Die Funktion *fclose* schließt eine geöffnete Datei. Sie muss durchgeführt werden, bevor mit einem Filepointer eine andere Datei geöffnet und bearbeitet werden kann.

19.2.3. AUSGABE IN EINE DATEI MIT FPRINTF

```
Syntax:
   int fprintf (FILE *Filepointer,
                "Text/Platzhalter", Variablenliste);
```

```
//=============================================================
// Programm FILEFPRI.CPP
//=============================================================

#include <stdio.h>

FILE *Datei      = NULL;
char Eingabe [40]   = "";

void main (void)
{
   fgets (Eingabe, 40, stdin);
   if ((Datei = fopen ("c:\\abc.dat","w")) != NULL)
   {
      fprintf (Datei, "Datei offen, Eingabe %s", Eingabe);
      fclose (Datei);
   }
}
```

Die Funktion *fprintf* entspricht der bekannten *printf*-Funktion, mit dem Unterschied, dass die formatierte Ausgabe nicht auf dem Bildschirm erfolgt, sondern in die mit dem Filepointer verbundene physikalische Datei. Die Angaben in *"Text und Platzhalter"* sowie die *"Variablenliste"* entsprechen exakt dem Format der *printf*-Funktion (Einschließlich Fluchtsymbole und Formatierungsschalter).

19.2.4. EINLESEN AUS EINER DATEI MIT FSCANF

```
Syntax:
   int fscanf (FILE *Filepointer, "Platzhalter",
               Variablenliste)
```

```
//=============================================================
// Programm FILEFPRI.CPP
//=============================================================

#include <stdio.h>

FILE *Datei      = NULL;
char Eingabe [40]   = "";
```

```
void main (void)
{
   if ((Datei = fopen ("c:\\abc.dat","r")) != NULL)
   {
      fscanf (Datei, "%s", Eingabe);
      puts (Eingabe);
      fclose (Datei);
   }
}
```

Die Funktion *fscanf* entspricht der bereits bekannten *scanf*-Funktion, mit dem Unterschied, dass die formatierte Eingabe nicht von der Tastatur erfolgt, sondern statt dessen aus der mit dem *Filepointer* verbundenen, physikalischen Datei gelesen wird. Die Platzhalterliste ist mit der Liste der Funktion *scanf* identisch.

19.2.5. EINLESEN AUS EINER DATEI MIT FGETS

```
Syntax:
   char *fgets (char *String, int MaxAnzahl,
                FILE *Filepointer);
```

```
//=============================================================
// Programm FILEFGET.CPP
//=============================================================

#include <stdio.h>

FILE *Datei      = NULL;
char Eingabe [40]   = "";

void main (void)
{
   if ((Datei = fopen ("c:\\abc.dat","r")) != NULL)
   {
      fgets (Eingabe, 40, Datei);
      puts (Eingabe);
      fclose (Datei);
   }
}
```

Die Funktion *fgets* liest maximal eine Zeile (also alle Zeichen, bis entweder die Datei beendet ist oder ein Zeilenvorschub gefunden wurde) aus der mit *Filepointer* verbundenen Datei. Die gelesenen Zeichen werden in der Zeichenkette *String* (hier *Einlesen*) abgelegt.

Maximal werden aber MaxAnzahl-1 Zeichen gelesen und ein abschließendes Stringende-Zeichen hinzugefügt, um einen Zeichenketten-

überlauf zu verhindern. Dadurch können (bei entsprechender Programmierung) die Grenzen von *String* nicht überschritten werden. Ist die Funktion erfolgreich, so wird der Zeiger auf die Zeichenkette *String* zurückgegeben, ansonsten der Standardpointer *NULL*.

19.2.6. AUSGABE IN EINE DATEI MIT FPUTS

```
Syntax:
   int fputs (char *String, FILE *Filepointer);
```

```
//=============================================================
// Programm FILEFPUT.CPP
//=============================================================

#include <stdio.h>

FILE *Datei      = NULL;
char Eingabe [40]   = "";

void main (void)
{
   gets (Eingabe);
   if ((Datei = fopen ("c:\\abc.dat","w")) != NULL)
   {
      fputs (Eingabe, Datei);
      fclose (Datei);
   }
}
```

Die Funktion *fputs* schreibt eine Zeichenkette (alle Zeichen bis auf das Stringende-Zeichen „*\0*“) in die mit *Filepointer* verbundene Datei. Wenn ein Fehler auftritt wird ein Wert ungleich Null, ansonsten Null zurückgegeben.

19.2.7. DATEIENDE ERKENNEN MIT FEOF

```
Syntax:
   int feof (FILE *Filepointer);
```

```
//=============================================================
// Programm FILEFEOF.CPP
//=============================================================

#include <stdio.h>

FILE *Datei = NULL;
char Ausgeben [40];
```

```
void main (void)
{
   if ((Datei = fopen ("c:\\abc.dat","r"))==NULL)
   {
      return;
   }
   else
   {
      while (!feof (Datei))
      {
         fgets (Ausgeben, 40, Datei);
         puts (Ausgeben);
      }
      fclose (Datei);
   }
}
```

Die Funktion *feof* testet, ob das Ende der mit *Filepointer* verbundenen Datei erreicht worden ist oder nicht. Die Funktion gibt den Wert Null zurück, solange das Ende noch nicht erreicht wurde, bzw. einen Wert ungleich Null wenn das Dateiende erreicht wurde.

19.2.8. DATEIFEHLER ERKENNEN MIT FERROR

```
Syntax :
   int ferror (FILE *Filepointer);
```

```
//=============================================================
// Programm FILEFERR.CPP
//=============================================================

#include <stdio.h>

FILE *Datei = NULL;
char Ausgeben [40];
int Zahl = 0;

void main (void)
{
   if ((Datei = fopen ("c:\\abc.dat","r")) != NULL)
   {
      fgets (Ausgeben, 40, Datei);
      Zahl = ferror (Datei);
      printf ("Fehler %s", Zahl);
      fclose (Datei);
   }
}
```

Die Funktion *ferror* testet, ob im Zusammenhang mit einer Ein- oder Ausgabe auf der mit *Filepointer* verbundenen Datei ein Fehler aufgetreten ist oder nicht (z.B. durch zerstörte Daten oder defekte Disketten).

Die Funktion gibt den Wert Null zurück, wenn kein Fehler, bzw. einen Wert ungleich Null wenn ein Fehler vorlag. Im letzteren Fall sperrt die Funktion alle weiteren Zugriffe auf die Datei, um der Zerstörung weiterer Daten vorzubeugen. Die Fehlercodes haben symbolische Namen, die in der Headerdatei <error.h> nachzulesen sind.

19.2.9. PUFFER LEEREN MIT FFLUSH

```
Syntax:
   int fflush (FILE *Filepointer);
```

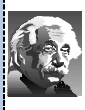
```
#include <stdio.h>

FILE *Datei = NULL;
char Ausgeben [40] = "Hallo Welt";

void main (void)
{
   if ((Datei = fopen ("c:\\abc.dat", "w")) !=NULL)
   {
      fputs (Ausgeben, Datei);
      fflush (Datei);
      fclose (Datei);
   }
}
```

Die Funktion *fflush* erzwingt die Leerung des Filebuffers und schreibt alle noch im Schreibpuffer lagernden Daten, auf den Datenträger. Die Funktion gibt den Wert Null zurück, wenn dabei kein Fehler auftrat, ansonsten wird ein Wert ungleich Null zurückgegeben.

Nach Leseoperationen von der Tastatur (z.B. mit fscanf() *oder* getch()*) bleibt häufig ein CR/LF (Returntaste) im Tastaturpuffer stehen. Dadurch wird die nächste Abfrage automatisch, ohne Eingabe des Anwenders, beantwortet.*



Das gleiche Problem tritt auf, wenn man mit scanf() *eine Zahl einlesen möchte, der Anwender aber Buchstaben eingibt. Um die restlichen, überflüssigen Zeichen nach einer Eingabe zu löschen empfiehlt es sich, nach jeder Eingabe durch den Anwender den folgenden Befehl aufzurufen:*

```
fflush (stdin);
```

19.2.10. AUSGABE IN EINE DATEI MIT FPUTC

```
Syntax:
   int fputc (char Zeichen, FILE *Filepointer);
```

```
#include <stdio.h>

FILE *Datei   = NULL;
char Eingabe   = 'H';

void main (void)
{
   if ((Datei = fopen ("c:\\abc.dat", "w"))!= NULL)
   {
      fputc (Eingabe, Datei);
      fclose (Datei);
   }
}
```

Das Unterprogramm *fputc* entspricht der bereits bekannten Funktion *putchar*, mit dem Unterschied, dass die Ausgabe von *Zeichen* nicht auf dem Bildschirm erfolgt, sondern in die mit *Filepointer* verbundene, physikalische Datei.

19.2.11. EINLESEN AUS EINER DATEI MIT FGETC

```
Syntax:
   int *fgetc (FILE *Filepointer);
```

```
#include <stdio.h>

FILE *Datei   = NULL;
char Eingabe   = '';

void main (void)
{
   if ((Datei = fopen ("c:\\abc.dat", "r+"))!=NULL)
   {
```

```
      Eingabe = fgetc (Datei);
      fclose (Datei);
   }
}
```

Das Unterprogramm *fgetc* entspricht der bereits bekannten *getchar*-Funktion, mit dem Unterschied, dass die Eingabe nicht von der Tastatur erfolgt, sondern aus der mit *Filepointer* verbundenen, physikalischen Datei.

19.3. DATENSATZFUNKTIONEN FÜR DATEIEN

Neben einfachen Textdateien, die Informationen zumeist sehr unstrukturiert enthalten, gibt es Datendateien, die nur Datensätze einer festen Größe beinhalten.

Kennzeichen solcher Dateien ist die Berechenbarkeit der relativen Position eines Datensatzes in einer Datei. D.h. der Abstand eines Datensatzes entspricht dem Wert des Dateizeigers) lässt sich mathematisch bestimmen. Durch einfache Mathematik lässt sich z.B. der Filepointer vor dem achten Datensatz positionieren, ohne dass die anderen Datensätze gelesen werden müssen. Die Funktionen, die dieses Vorgehen in C realisieren heißen *fseek* und *ftell*.

19.3.1. POSITIONIEREN IN DATENSÄTZEN MIT FSEEK

```
Syntax
   int fseek (FILE *Filepointer, long Offset,
              int Origin);
```

```
//=============================================================
// Programm FILESEEK.CPP
//=============================================================

#include <stdio.h>

FILE *Datei = NULL;
char Ausgeben [40];

void main (void)
{
   if ((Datei = fopen ("c:\\abc.dat","r+"))!= NULL)
   {
      fseek (Datei, 5, SEEK_SET);
      fgets (Ausgeben, 40, Datei);
      puts (Ausgeben);
      fclose (Datei);
   }
}
```

Die Funktion *fseek* verschiebt den Filepointer in der Datei um den in *Offset* („Verschiebung“) genannten Wert (in Bytes), ausgehend von der Position *Origin* („Ausgangspunkt“). Dabei gibt *Offset* die Richtung (positiver oder negativer Wert) und die Entfernung an. *Origin* muss eine der drei folgenden, symbolischen Konstanten sein:

```
"SEEK_SET"   relativ zum Dateianfang.
"SEEK_END"   relativ zum Dateiende.
"SEEK_CUR"   relativ zur aktuellen Position.
```

Konnte die neue Position eingestellt werden, so wird der Wert Null, ansonsten ein Wert ungleich Null zurückgegeben. Soll z.B. der sechste Datensatz gelesen werden, so wäre der Wert 100 im obigen Beispiel zu ersetzen durch eine auf der Datensatzlänge basierenden Berechnung:

```
fseek (Datei, 6 * DATENSATZLAENGE, SEEK_SET);
```

Anschließend kann der Datensatz gelesen (z.B. mit *fgets*) und auf eine entsprechende Struktur zugewiesen werden.

19.3.2. POSITIONIEREN AUF DEN DATEIANFANG MIT REWIND

```
Syntax:
   int rewind (FILE *Filepointer);
```

Die Funktion *rewind* setzt den Datensatzzeiger (*Filepointer*) wieder an den Anfang der Datei zurück, so als wäre diese gerade geöffnet worden. Steht die Funktion *rewind* nicht zur Verfügung, so kann sie leicht durch einen *fseek*-Befehl ersetzt werden.

```
fseek (Datei,0,SEEK_SET);
```

19.3.3. POSITION ABFRAGEN MIT FTELL

```
Syntax:
   int long ftell (FILE *Filepointer);
```

```
//============================================================
// Programm FILETELL.CPP
//============================================================

#include <stdio.h>
```

```
FILE *Datei      = NULL;
long Position   = 0;

void main (void)
{
   if ((Datei = fopen ("c:\\abc.dat","r+"))!= NULL)
   {
      fseek (Datei, 5, SEEK_SET);
      Position = ftell (Datei);
      fclose (Datei);
      printf ("Position Filepointer: %ld", Position);
   }
}
```

Die Funktion *ftell* bildet das Gegenstück zur Funktion *fseek*, sie ermittelt die aktuelle Position des Filepointers in der Datei relativ zum Dateianfang (in Bytes). Tritt bei der Ermittlung ein Fehler auf (z.B. weil die Datei geschlossen ist), so wird der Wert -1L (L=long) zurückgegeben, ansonsten die Positionsangabe, d.h. der Inhalt des Filepointers.

19.4. DATEIVERWALTUNG

Neben den Funktionen zur Bearbeitung von Dateiinhalten gibt es auch Unterprogramme, die Dateien verwalten können. Die Dateiverwaltung umfasst zumindest so wichtige Funktionen wie das Löschen und Umbenennen von Dateien auf dem Speichermedium.

19.4.1. DATEIEN LÖSCHEN MIT REMOVE

```
Syntax:
   int remove (char *Filename);
```

```
//============================================================
// Programm FILERMVE.CPP
//============================================================

#include <stdio.h>

void main (void)
{
   remove ("c:\\abc.dat");
}
```

Die Funktion *remove* löscht die angegebene physikalische Datei. Die zu löschende Datei darf nicht geöffnet sein, da sonst der Filepointer einen nicht definierten Zustand einnehmen würde. Die Funktion gibt Null zurück wenn kein Fehler auftrat, sonst einen Wert ungleich Null.

19.4.2. DATEIEN UMBENENNEN MIT RENAME

```
Syntax:
   int rename (char *Oldname, char *Newname);
```

```
//============================================================
// Programm FILERNME.CPP
//============================================================

#include <stdio.h>

void main (void)
{
   rename ("c:\\abc.dat", "c:\\neu.dat");
}
```

Die Funktion *rename* benennt die in angegebene physikalische Datei (*Oldname*) auf den in *Newname* angegebenen Namen um. Die Funktion gibt Null zurück wenn kein Fehler auftrat, sonst einen Wert ungleich Null.

Wird nicht der Dateiname geändert, sondern zusätzlich oder ausschließlich der Pfad, so wird die Datei, ggf. unter dem angegebenen, neuen Namen, in das entsprechende Verzeichnis verschoben:

```
#include <stdio.h>

void main (void)
{
   // Nur verschieben
   rename ("c:\\def.dat", "c:\\windows\\def.dat");

   // Verschieben und umbenennen
   rename ("c:\\ghi.dat", "c:\\windows\\def2.dat");
}
```

Leider hat die Funktion rename() die unangenehme Eigenschaft nicht über Laufwerke hinweg verschieben zu können. Das Verschieben einer Datei auf ein anderes physikalisches oder logisches Laufwerk ist daher nicht möglich.

Es ist allerdings nicht schwierig einen entsprechenden Befehl selbst zu schreiben. Die folgenden beiden Funktionen realisieren ein beliebiges Kopieren und Verschieden von Dateien:

```
//=============================================================
// Programm MYFILE.CPP
//=============================================================

#include <io.h>         // Lowlevel Input/Output
#include <stdio.h>      // Standard Input/Output
#include <string.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <errno.h>

//-------------------------------------------------------------
// int fnCopyFile (LPCTSTR lpFile, LPCTSTR lpNewFile);
//-------------------------------------------------------------
// Parameter
// lpFileName – Name der existierenden Datei
// lpNewFile  - Name der Zieldatei
//-------------------------------------------------------------
// Return-Werte
//    0 = Operation war erfolgreich
//-------------------------------------------------------------
// -10 = Quelldatei ist nicht zu öffnen  (ERROR NOT SPECIFIED)
// -11 = Quelldatei ist nicht zu öffnen  (ACCESS DENIED)
// -12 = Quelldatei ist nicht zu öffnen  (INVALID ACCESS CODE)
// -13 = Quelldatei ist nicht zu öffnen  (TOO MANY OPEN FILES)
// -14 = Quelldatei ist nicht zu öffnen  (NO SUCH FILE OR DIR)
//-------------------------------------------------------------
// -20 = Zieldatei ist nicht zu öffnen  (ERROR NOT SPECIFIED)
// -21 = Zieldatei ist nicht zu öffnen  (ACCESS DENIED)
// -22 = Zieldatei ist nicht zu öffnen  (INVALID ACCESS CODE)
// -23 = Zieldatei ist nicht zu öffnen  (TOO MANY OPEN FILES)
// -24 = Zieldatei ist nicht zu öffnen  (NO SUCH FILE OR DIR)
//-------------------------------------------------------------
// -30 = Buffer kann nicht angelegt werden
//-------------------------------------------------------------
// -40 = Lesezugriff verweigert
// -41 = Bad Filenumber
//-------------------------------------------------------------
// -50 = Schreibzugriff verweigert
// -51 = Bad Filenumber
//-------------------------------------------------------------
// -60 = allgemeiner Kopierfehler
//-------------------------------------------------------------
// -70 = Quell und Zieldatei sind identisch
//-------------------------------------------------------------
int fnCopyFile (char *lpFile, char *lpNewFile)
{
   #define MYBUFFERSIZE 32000
   extern int errno;
   int   nRHandle, nWHandle; // Handle Quell- und Zieldatei
   int   nBytes = 1;         // zurückgemeldete Buffergröße
   int   nResult;            // Anzahl geschriebene Zeichen
   int   nOk  = 1;           // Schreiboperation ok
   void *buff = NULL;        // Datenpuffer
   int   nError = 0;         // Fehlernummer
```

```
   if (!stricmp (lpFile, lpNewFile))
      return (-70);

   //-------------------------------------------------------
   // Quell- und Zieldatei öffnen
   //-------------------------------------------------------
   nRHandle = open (lpFile, O_RDONLY|O_BINARY,
                    S_IWRITE|S_IREAD);

   if(-1 == nRHandle)
   {
      switch (errno)
       {
         default:      nError = -10; break;
         case EACCES:  nError = -11; break;
         case EINVACC: nError = -12; break;
         case EMFILE:  nError = -13; break;
         case ENOENT:  nError = -14; break;
      }
      return (nError);
   }

   nWHandle = open (lpNewFile, O_CREAT|O_WRONLY|O_BINARY,
                    S_IWRITE|S_IREAD);

   if(-1 == nWHandle)
   {
      close (nRHandle);
      switch (errno)
      {
         default:      nError = -20; break;
         case EACCES:  nError = -21; break;
         case EINVACC: nError = -22; break;
         case EMFILE:  nError = -23; break;
         case ENOENT:  nError = -24; break;
      }
   }

   //-------------------------------------------------------
   // Beide Dateien sind offen, jetzt Bufferbereich anlegen
   // Wenn Buffer nicht angelegt werden kann, dann return –30
   //-------------------------------------------------------
   buff = new unsigned char [MYBUFFERSIZE];

   if (!buff)
   {
      close (nRHandle);
      close (nWHandle);
      return (-30);
   }

   //-------------------------------------------------------
   // Beide Dateien sind offen und Puffer existiert
   // Kopierprozess kann beginnen
   //-------------------------------------------------------
   while ((nBytes) && (nOk) && (!nError))
```

```
   {
      nBytes  = read  (nRHandle, buff, MYBUFFERSIZE);
      if (nBytes == -1)
      {
         switch (errno)
         {
            case EACCES: nError = -40; break;
            case EBADF:  nError = -41; break;
         }
         nOk = 0;
      }
      nResult = write (nWHandle, buff, nBytes);
      if (nResult == -1)
      {
         switch (errno)
         {
            case EACCES: nError = -50; break;
            case EBADF:  nError = -51; break;
         }
         nOk = 0;
      }

      if (nBytes != nResult)
      {
         nOk = 0;
         if (!nError)
            nError = -60;
      }
   }

   //-------------------------------------------------------
   // Alles aufräumen
   //-------------------------------------------------------
   close (nRHandle);
   close (nWHandle);
   delete [] buff;

   //-------------------------------------------------------
   // Fehler beim Kopieren, dann Zielfile wieder entfernen
   //-------------------------------------------------------
   if (nError < 0)
   {
      remove (lpNewFile);
      return (nError);
   }

   //-------------------------------------------------------
   // Alles ok
   //-------------------------------------------------------
   return (0);
   #undef MYBUFFERSIZE
}

//-------------------------------------------------------------
// int fnMoveFile (LPCTSTR lpFile, LPCTSTR lpNewFile);
//-------------------------------------------------------------
```

```
// Parameter
// lpFile     – Name der zu kopierenden Datei
// lpNewFile  - Name der Zieldatei
//-------------------------------------------------------------
// Return-Werte
//    0 = Operation war erfolgreich
//-------------------------------------------------------------
// -10 = Quelldatei ist nicht zu öffnen  (ERROR NOT SPECIFIED)
// -11 = Quelldatei ist nicht zu öffnen  (ACCESS DENIED)
// -12 = Quelldatei ist nicht zu öffnen  (INVALID ACCESS CODE)
// -13 = Quelldatei ist nicht zu öffnen  (TOO MANY OPEN FILES)
// -14 = Quelldatei ist nicht zu öffnen  (NO SUCH FILE OR DIR)
//-------------------------------------------------------------
// -20 = Zieldatei ist nicht zu öffnen  (ERROR NOT SPECIFIED)
// -21 = Zieldatei ist nicht zu öffnen  (ACCESS DENIED)
// -22 = Zieldatei ist nicht zu öffnen  (INVALID ACCESS CODE)
// -23 = Zieldatei ist nicht zu öffnen  (TOO MANY OPEN FILES)
// -24 = Zieldatei ist nicht zu öffnen  (NO SUCH FILE OR DIR)
//-------------------------------------------------------------
// -30 = Buffer kann nicht angelegt werden
//-------------------------------------------------------------
// -40 = Lesezugriff verweigert
// -41 = Bad Filenumber
//-------------------------------------------------------------
// -50 = Schreibzugriff verweigert
// -51 = Bad Filenumber
//-------------------------------------------------------------
// -60 = allgemeiner Kopierfehler
//-------------------------------------------------------------
// -70 = Quell und Zieldatei sind identisch
//-------------------------------------------------------------
int fnMoveFile (char *lpFile, char *lpNewFile)
{
   int nResult = fnCopyFile (lpFile, lpNewFile);

   if (!nResult)
      remove (lpFile);

   return (nResult);
}
```

```
//=============================================================
// Programm MYFILE.H
//=============================================================

#ifndef _MYFILE_H_
   #define _MYFILE_H_

   int fnCopyFile (char *lpFile, char *lpNewFile);
   int fnMoveFile (char *lpFile, char *lpNewFile);

#endif
```

```
//=============================================================
// Programm MYFILEMN.CPP
//=============================================================

#include <iostream>
#include <iomanip>
#include "myfile.h"

using namespace std;

void main (void)
{
   int result;
   result = fnCopyFile ("c:\\autoexec.bat","d:\\test.tst");
   cout << result << endl;

   result = fnMoveFile ("d:\\test.tst", "c:\\test.tst");
   cout << result << endl;
}
```

19.5. STANDARDDEVICES

Die Ein- und Ausgabe für Dateien kann auch dazu benutzt werden, um Texte auf den Drucker auszugeben. Die Funktion *printf* ist im Grunde nur eine Abkürzung für *fprintf*, wobei der Filepointer mit *stdout* (Standard-Ausgabekanal) vorbelegt ist. Die meisten C-Compiler kennen neben *stdout* und *stdin* (Tastatur) auch einen Standarddrucker (*stdprn*). Diese Standardgeräte (Devices) werden automatisch bei Programmstart geöffnet und brauchen daher nicht erst mit *fopen* initialisiert werden. Mit einfachen Dateiausgabeanweisung kann so auch auf den Drucker ausgegeben werden:

```
//=============================================================
// Programm PRINT.CPP
//=============================================================

#include <stdio.h>

char Ausgeben [40] = "Hallo Weltn\f";

void main (void)
{
   fputs (Ausgeben, stdprn);
}
```

20. SORTIEREN UNTER C UND C++

Das Sortieren von Arrays ist eine häufig wiederkehrende Aufgabe in der Programmierung. Dabei ist es von entscheidender Bedeutung, welche Art von Datentyp im Array enthalten ist, denn die Methodik des Sortierens ist von diesem nicht unabhängig. Selbst bei den scheinbar einfachen Grunddatentypen wie short, int, long, float, double oder long double bleibt die Frage zu klären, ob die Daten auf- oder absteigend sortiert werden soll.

Bei komplexeren Datentypen stellt sich zudem die Frage, nach welchen Kriterien überhaupt sortiert werden soll, so z.B. bei zusammengesetzten Datentypen, den Strukturen. Bei anderen, Pointer basierten Datentypen hingegen, z.B. den auf char-Pointern beruhenden C-Strings, ist zu überlegen, wie die Elemente sauber vertauscht werden können.

20.1. SORTIEREN UNTER ANSI-C

Unter C gibt es grundsätzlich zwei Möglichkeiten, Sortierprobleme zu lösen. Neben der Implementation eines eigenen Sortieralgorithmus kann man alternativ auf den bereits in ANSI-C enthaltenen und vorimplementierten Quicksort[7] zurückgreifen.

20.1.1. IMPLEMENTATION EINES EINFACHEN SORTIERALGORITHMUS UNTER ANSI-C

Der einfachste Sortieralgorithmus, den man implementieren kann ist der sogenannte Bubblesort. Das Verfahren beruht darauf, die Elemente einer Liste nacheinander zu vergleichen und zu vertauschen, wenn die Vergleichsbedingung anzeigt, dass die Reihenfolge der Elemente nicht der angestrebten Sortierreihenfolge genügt. Weil die korrekt sortierten Elemente dabei im Array aufsteigen wie die Bläschen in einer Sodaflasche erhielt der Algorithmus den Namen „Bubblesort“.

Ein Vorteil des Bubblesort ist es, dass der Algorithmus praktisch keinen zusätzlichen Platzbedarf hat — abgesehen von einem zusätzlichen Element, welches temporär zum Vertauschen benötigt wird.

Die Nachteile hingegen überwiegen, denn das Verfahren ist vom zeitlichen Aufwand her sehr ineffizient, da es vergleichsweise viele Vergleichs- und Vertauschungsaufrufe benötigt, bis das Array am Ende sortiert ist. Dies ist dem nachstehenden Pseudocode nicht direkt zu entnehmen, wird aber in den folgenden Beispiel-Implementationen sehr deutlich.

[7] Eine umfassende Darstellung verschiedener Sortier- und Mischalgorithmen findet sich z.B. in „Algorithmen und Datenstrukturen“ von Niklaus Wirth

Formal kann man den Algorithmus folgendermaßen in Pseudocode beschreiben:

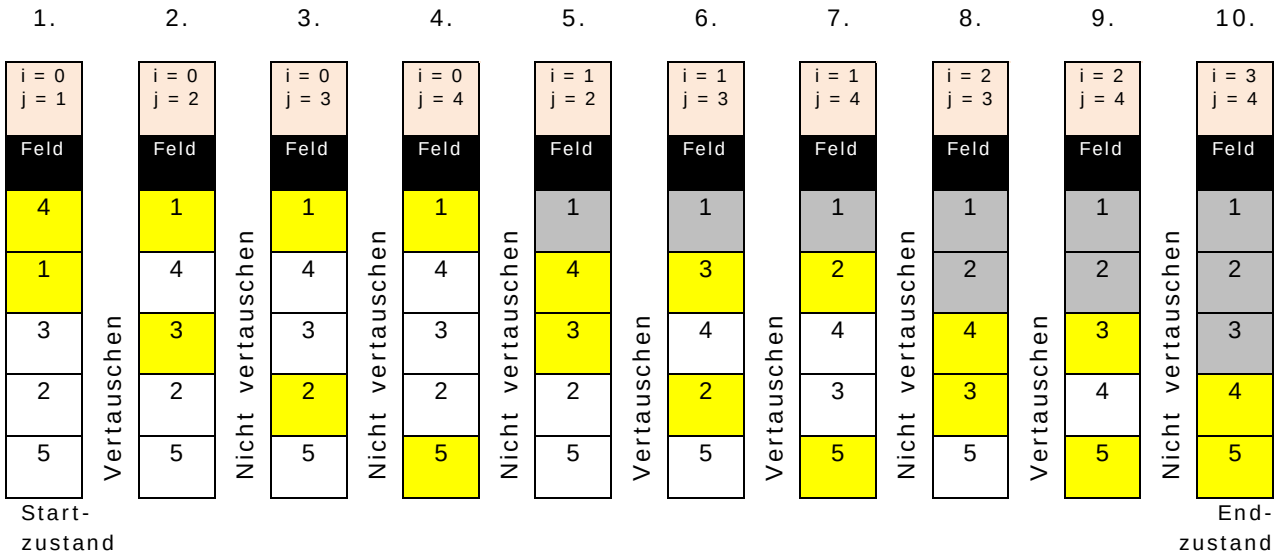
```
prozedur bubbleSort( A : Liste sortierbarer Elemente )
definiert als:
  n := Länge( A ) - 1
  wiederhole
    vertauscht := falsch
    für jedes i von 0 bis n wiederhole:
      falls A[ i ] > A[ i + 1 ] dann
        vertausche( A[ i ], A[ i + 1 ] )
        vertauscht := wahr
      falls_ende
    nächstes
    n := n - 1
  solange vertauscht
prozedur_ende
```

20.1.1.1. BUBBLESORT FÜR INT

Für ein Integer-Feld sieht die konkrete Ausprägung unter ANSI-C dann wie folgt aus:

```
void bubblesort_int (int *pArray, const int pSize)
{
   int i, j;
   int nBuffer;
   for (i=0; i<pSize-1; i++)
   {
      for (j=i+1; j<pSize; j++)
      {
         if (pArray [j] < pArray[i])
         {
             nBuffer   = pArray[i];
             pArray[i] = pArray[j];
             pArray[j] = nBuffer;
         }
      }
   }
}
```

Betrachtet man sich den Verlauf der Zustände, die ein Array im Zuge der Ausführung annimmt, dann erkennt man, wie die beiden Schleifen ineinandergreifen. Hier dargestellt an einem Array mit den Inhalten 4, 1, 3, 2 und 5. Der Parameter pSize hat dementsprechend die Länge Fünf:



Ein Hauptprogramm, welches die Funktion bubblesort_int benutzt, könnte beispielsweise so aussehen:

```
#include <iostream>
#include <iomanip>

using namespace std;

#define ARRAYSIZE 5

// Prototype
void bubblesort_int (int *pArray, const int pSize);
int main (int argc, char *argv[])
{
   int aMeinArray [ARRAYSIZE] = { 17, 3, -3, 5, 99 };

   bubblesort_int (aMeinArray, ARRAYSIZE);

   for (int i=0; i<ARRAYSIZE; i++)
   {
      cout << aMeinArray[i] << endl;
   }
   return 0;
}
```

Unter reinem ANSI-C, wenn die Stream-IO nicht zur Verfügung steht, muss das Ausgabeobjekt cout durch die Standardfunktion printf ersetzt werden. Auf die Angabe des Namespace ist dann ebenfalls zu verzichten.

```
#include <stdio.h>

#define ARRAYSIZE 5

void bubblesort_int (int *pArray, const int pSize);
```

```
int main (int argc, char *argv[])
{
   int aMeinArray [ARRAYSIZE] = {33, 22, 11, 55, 44};

   bubblesort_double (aMeinArray, ARRAYSIZE);

   for (int i=0; i<ARRAYSIZE; i++)
   {
      printf ("%d\n", aMeinArray[i]);
   }
   return 0;
}
```

Wie man sieht, erhält die Funktion zwei Parameter (pArray und pSize). Bei pArray handelt es sich um einen Integer-Pointer, also den Zeiger auf das erste Element des Arrays. Wie der nachstehenden Tabellendarstellung zu entnehmen ist, sind die Adressen des Arrays (aMeinArray) und der Inhalt des Parameters pArray identisch.

Rechts in den beiden Spalten neben den Werten ist aufgezeigt, wie sich der derefrenzierte Pointerzugriff *(pArray + x) und die Arrayschreibweise aMeinArray [x] entsprechen.

Parameter (Adressberechnung)	Adresse (Zeiger)	Wert	Variablenname (Zugriff auf Inhalt)	Parametername (Zugriff auf Inhalt)
pArray ≙ aMeinArray	10000	17	aMeinArray [0]	*pArray
pArray + 1	10004	3	aMeinArray [1]	*(pArray + 1)
pArray + 2	10008	-3	aMeinArray [2]	*(pArray + 2)
pArray + 3	10012	5	aMeinArray [3]	*(pArray + 3)
pArray + 4	10016	99	aMeinArray [4]	*(pArray + 4)

Die Übergabe an die Funktion bubblesort_int verwendet nur den Anfangszeiger des Arrays, um die Funktion auch mit anderen Arraylängen — als durch ARRAYSIZE definiert — verwenden zu können. Die Länge, also die Anzahl der im Array gespeicherten Elemente wird im Parameter pSize übergeben.

Es wäre auch möglich, die Funktion ohne Aufteilung in Pointer und Größe zu schreiben, dann hätte der Prototyp die Form:

```
void bubblesort_int (int pArray[ARRAYSIZE]);
```

Wie man leicht erkennen kann, ist dieser Sortieralgorithmus jetzt nur noch für Integer-Arrays der Länge ARRAYSIZE (im Beispiel Fünf) geeignet. Für jede andere Arraylänge wäre jetzt eine weitere Funktion zu schreiben. Da die Konstante ARRAYSIZE zum Zeitpunkt der Übersetzung festgeschrieben wird, ist die Funktion auch kaum wiederverwendbar.

20.1.1.2. BUBBLESORT FÜR DOUBLE

Da es unter einfachem ANSI-C keine Möglichkeit des Function-Overloading gibt, muss für alle weiteren Datentypen jeweils eine eigene Funktion mit eindeutigem Namen geschrieben werden.

```
#include <stdio.h>

#define ARRAYSIZE 5

void bubblesort_double (double *pArray, const int pSize);

int main (int argc, char *argv[])
{
   double aMeinArray [ARRAYSIZE] = {33.3, 22.2, 11.1, 55.5,
44.4};

   bubblesort_double (aMeinArray, ARRAYSIZE);

   for (int i=0; i<ARRAYSIZE; i++)
   {
      printf ("%lf\n", aMeinArray[i]);
   }
   return 0;
}

void bubblesort_double (double *pArray, const int pSize)
{
   int i, j;
   double fBuffer;
   for (i=0; i<pSize-1; i++)
   {
      for (j=i+1; j<pSize; j++)
      {
         if (pArray[j] < pArray[i])
         {
             fBuffer   = pArray[i];
             pArray[i] = pArray[j];
             pArray[j] = fBuffer;
         }
      }
   }
}
```

Wie leicht zu erkennen ist, stimmen die Algorithmen für die Sortierverfahren von `int` und `double` bis auf den Datentyp völlig überein. Die Darstellung gilt letztlich für alle einfachen Datentypen, für die ein Vergleich über die Vergleichsoperatoren (`<`, `>`, `==`, `!=`, `<=` und `>=`) definiert ist. Ist eine Reihung mit Hilfe dieser Operatoren nicht möglich, so muss auf Vergleichsfunktionen zurückgegriffen werden, wie es z.B. bei `char`-Arrays mit der Funktion `strcmp` üblich ist.

20.1.1.3. BUBBLESORT FÜR STRUCT

Wie bereits beschrieben, ist bei Strukturen hauptsächlich zu klären, wie eine Sortierung vorzunehmen ist. Ein Vergleich mit Hilfe der Vergleichsoperatoren ist nur beim Vergleich von Strukturkomponenten, die dieses unterstützen möglich, aber nicht zwischen kompletten Strukturen.

```
#include <stdio.h>
#include <string.h>

#define ARRAYSIZE 5
#define NAMESIZE  51

struct TAdresse {char Nachname [NAMESIZE]; char Vorname
[NAMESIZE]; int Gebjahr;};

void bubblesort_adresse (TAdresse *pArray, const int pSize);

int main (int argc, char *argv[])
{
   TAdresse aMeinArray [ARRAYSIZE];

   strcpy (aMeinArray[0].Nachname, "Merkel");
   strcpy (aMeinArray[0].Vorname,  "Angela");
   aMeinArray[0].Gebjahr = 1954;

   strcpy (aMeinArray[1].Nachname, "Kohl");
   strcpy (aMeinArray[1].Vorname,  "Helmut");
   aMeinArray[1].Gebjahr = 1930;

   strcpy (aMeinArray[2].Nachname, "Kohl");
   strcpy (aMeinArray[2].Vorname,  "Hannelore");
   aMeinArray[2].Gebjahr = 1933;

   strcpy (aMeinArray[3].Nachname, "Brandt");
   strcpy (aMeinArray[3].Vorname,  "Willy");
   aMeinArray[3].Gebjahr = 1913;

   strcpy (aMeinArray[4].Nachname, "Schröder");
   strcpy (aMeinArray[4].Vorname,  "Gerhard");
   aMeinArray[4].Gebjahr = 1944;

   bubblesort_adresse (aMeinArray, ARRAYSIZE);

   for (int i=0; i<ARRAYSIZE; i++)
   {
      printf ("%s, %s, %d\n", aMeinArray[i].Nachname,
aMeinArray[i].Vorname,

aMeinArray[i].Gebjahr);
   }
   return 0;
}

void bubblesort_adresse (TAdresse *pArray, const int pSize)
```

```
{
   int i, j;
   TAdresse rBuffer;
   for (i=0; i<pSize-1; i++)
   {
      for (j=i+1; j<pSize; j++)
      {
         if (strcmp(pArray[j].Nachname, pArray[i].Nachname) <
0)
         {
             rBuffer   = pArray[i];
             pArray[i] = pArray[j];
             pArray[j] = rBuffer;
         }
         else if (strcmp(pArray[j].Nachname,
pArray[i].Nachname) == 0)
         {
            if (strcmp(pArray[j].Vorname, pArray[i].Vorname) <
0)
            {
               rBuffer   = pArray[i];
               pArray[i] = pArray[j];
               pArray[j] = rBuffer;
            }
            else if (strcmp(pArray[j].Vorname,
pArray[i].Vorname) == 0)
            {
               if (pArray[j].Gebjahr < pArray[i].Gebjahr)
               {
                  rBuffer   = pArray[i];
                  pArray[i] = pArray[j];
                  pArray[j] = rBuffer;
               }
            }
         }
      }
   }
}
```

Die Sortierfunktion bubblesort_adresse zeigt, dass schon die Bewertung einer relativ einfachen Struktur zu aufwändigen, verschachtelten Vergleichsfunktionen führt.

20.1.1.4. BUBBLESORT FÜR CHAR-ARRAYS (C-STRINGS)

Noch etwas diffiziler ist Sortierung von einfachen C-Zeichenketten (char-Arrays), da es sich letztlich um zweidimensionale Arrays handelt.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define ARRAYSIZE 5
#define NAMESIZE  51
```

```
void bubblesort_chararray (char *pArray, const int pNamesize,
const int pSize);

int main (int argc, char *argv[])
{
   char aMeinArray [ARRAYSIZE] [NAMESIZE];

   strcpy (aMeinArray[0], "Donald");
   strcpy (aMeinArray[1], "Daisy");
   strcpy (aMeinArray[2], "Dagobert");
   strcpy (aMeinArray[3], "Gustav");
   strcpy (aMeinArray[4], "Daniel");

   bubblesort_chararray (aMeinArray[0], NAMESIZE, ARRAYSIZE);

   for (int i=0; i<ARRAYSIZE; i++)
   {
      printf ("%s\n",aMeinArray[i]);
   }
   return 0;
}

void bubblesort_chararray (char *pArray, const int pNamesize,
const int pSize)
{
   int i, j;
   char *cBuffer = (char *) malloc(pNamesize);
   for (i=0; i<pSize-1; i++)
   {
      for (j=i+1; j<pSize; j++)
      {
         if (strcmp(pArray+j*pNamesize, pArray+i*pNamesize) <
0)
         {
             strcpy (cBuffer, pArray+i*pNamesize);
             strcpy (pArray+i*pNamesize, pArray+j*pNamesize);
             strcpy (pArray+j*pNamesize, cBuffer);
         }
      }
   }
   free (cBuffer);
}
```

Folgende Besonderheiten sind zu beachten. Da es sich um ein zweidimensionales Arrays handelt, wird der Zeiger auf das erste Element gebildet, indem man noch die erste Dimension mit angibt:

```
bubblesort_chararray (aMeinArray[0], NAMESIZE,
ARRAYSIZE);
```

Außerdem ist der Sortierfunktion die Länge der zweiten Dimension mitzugeben, da in der Sortierfunktion die zweite Dimension von Hand in

die Zeigerfortschaltung einbezogen werden muss. Sonst wäre die Funktion nur in der Lage Zeichenketten-Arrays von Zeichenketten der Länge NAMESIZE zu sortieren.

```
if (strcmp(pArray+j*pNamesize, pArray+i*pNamesize) < 0)
```

20.1.2. VERWENDUNG DES QUICKSORT-ALGORITHMUS UNTER ANSI-C

Der Quicksort-Algorithmus ist in der Regel einer der effizientesten Sortieralgorithmen die es gibt. Nur in einigen Ausnahmefällen, wie z.B. bei bereits vorsortierten Listen gibt es Algorithmen, die unter Umständen effizienter arbeiten.

Diese Effizienz hat natürlich einen Preis — der Quicksort baut zur Sortierung einen komplexen Baum auf, benötigt also temporär noch einmal den Speicherplatz der zu sortierenden Elemente plus den Platz der für die Baumstruktur selbst benötigt wird.

Der Quicksort-Algorithmus ist nicht einfach zu implementieren, daher gibt es in ANSI-C eine vorimplementierte Variante mit dem Funktionsnamen qsort. Da bei der Implementierung natürlich nicht bekannt sein konnte, welche benutzerdefinierten Datentypen (struct) sortiert werden sollen, muss der Anwendungsentwickler eine sogenannte Bewertungsfunktion schreiben, welche von der qsort-Funktion bei Bedarf aufgerufen wird.

Um die Art und Weise zu verstehen, wie die qsort-Funktion funktioniert, muss man sich die Parameterleiste der Bewertungsfunktion und der qsort-Funktion selbst ansehen. Die folgenden Ausführungen beziehen sich dazu auf das nachstehende Beispiel zur Sortierung eines Integer-Arrays.

```
int sortIntFunction (const void *a, const void *b);
```

```
qsort ((void *)nZahlenArray, ARRAYSIZE, sizeof(int),
sortIntFunction);
```

In beiden Parameterleisten werden void-Pointer verwendet. Die void-Pointer dienen dazu, die Datentypisierung der Zeiger aufzuheben. Da alle Zeiger, gleichgültig auf was sie zeigen, die gleiche Länge haben (abhängig vom Speichermodell des Betriebssystems, derzeit sind das üblicherweise 32 Bit), kann man jeden typisierten Pointer (hier int *) in einen nicht typisierten Pointer (void *) umwandeln. Damit geht aber auch die Implizite Längeninformation des Datentyps verloren. Der Compiler „weiß“ natürlich, aus wie vielen Bytes ein Integer besteht (vier in einem 32 Bit Datenmodell), also weiß er auch, dass ein int * auf ein

Speicherstück von vier Byte zeigt. Durch die Umwandlung in void * wird diese Information entzogen, der void-Pointer zeigt demnach nur auf ein einzelnes Byte.

Damit die Funktion qsort mitbekommt, wie groß die einzelnen Speicherstücke sind, muss diese Information als eigenständiger Parameter übergeben werden. Dies geschieht mit Hilfe des sizeof() Operators. Zudem wird die Anzahl der zu sortierenden Arrayelemente benötigt, genau wie zuvor bei der selbstgeschriebenen Sortierfunktion. Ein Umstand, welcher der Tatsache geschuldet ist, dass C / C++ am Array selbst keine Information über die Länge des Feldes hält.

Interessant ist auch der letzte Parameter, der durch den Namen der Sortierfunktion gebildet wird. Hier wird der Funktion qsort ein *Funktionspointer* auf eine Bewertungsfunktion übergeben (ein sogenannter Callback). Von einer gültigen Bewertungsfunktion für qsort wird zwingend verlangt, die gleiche Parameterleiste und den gleichen Rückgabetyp wie die Funktion sortIntFunction zu besitzen. Die Sortierung selbst wird von der qsort-Funktion durchgeführt, der Bewertungsfunktion obliegt es nur auf Anforderung zu entscheiden, in welcher Reihenfolge zwei zu vergleichende Elemente stehen. Dies wird der qsort-Funktion durch ein Integer-Ergebnis signalisiert (Minus Eins bedeutet, der erste Parameter ist im Sinne der Sortierung kleiner als der zweite. Eins bedeutet das Gegenteil und Null bedeutet, dass beide Elemente als gleich zu betrachten sind).

20.1.2.1. QUICKSORT FÜR INT

Das nachstehende Beispiel zeigt den Einsatz der qsort-Funktion für ein einfaches Integer-Array.

```
#include <stdio.h>
#include <stdlib.h>

#define ARRAYSIZE 5

int sortIntFunction (const void *a, const void *b);

int main(int argc, char* argv[])
{
  int nZahlen [ARRAYSIZE] = {33, 22, 11, 55, 44};

  qsort((void *)nZahlen, ARRAYSIZE, sizeof(int),
sortIntFunction);
  for (int i=0; i<ARRAYSIZE; i++)
  {
     printf ("%d\n", nZahlen[i]);
  }
  return 0;
}
```

```
int sortIntFunction (const void *a, const void *b)
{
  int x = *((int *)a);
  int y = *((int *)b);
  if (x<y) return -1;
  if (x>y) return  1;
  return 0;
}
```

20.1.2.2. QUICKSORT FÜR DOUBLE

Das nachstehende Beispiel zeigt den Einsatz der qsort-Funktion für ein einfaches Double-Array.

```
#include <stdio.h>
#include <stdlib.h>

#define ARRAYSIZE 5

int sortDoubleFunction (const void *a, const void *b);

int main(int argc, char* argv[])
{
  double nZahlen [ARRAYSIZE] = {33.3, 22.2, 11.1, 55.5, 44.4};

  qsort((void *)nZahlen, ARRAYSIZE, sizeof(double),
sortDoubleFunction);
  for (int i=0; i<ARRAYSIZE; i++)
  {
     printf ("%lf\n", nZahlen[i]);
  }
  return 0;
}

int sortDoubleFunction (const void *a, const void *b)
{
  double x = *((double *)a);
  double y = *((double *)b);
  if (x<y) return -1;
  if (x>y) return  1;
  return 0;
}
```

20.1.2.3. QUICKSORT FÜR STRUCT

Durch die Tatsache, dass man das Vertauschen/Sortieren der Elemente nicht selbst durchführen muss, wird die Komplexität der Bewertungsfunktion gegenüber dem selbstgeschriebenen Bubblesort-Algorithmus drastisch reduziert.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define ARRAYSIZE 5
```

```
#define NAMESIZE  51

struct TAdresse {char Nachname [NAMESIZE]; char Vorname
[NAMESIZE]; int Gebjahr;};

int sortAdressFunction (const void *a, const void *b);

int main(int argc, char* argv[])
{
   TAdresse aMeinArray [ARRAYSIZE];

   strcpy (aMeinArray[0].Nachname, "Merkel");
   strcpy (aMeinArray[0].Vorname,  "Angela");
   aMeinArray[0].Gebjahr = 1954;

   strcpy (aMeinArray[1].Nachname, "Kohl");
   strcpy (aMeinArray[1].Vorname,  "Helmut");
   aMeinArray[1].Gebjahr = 1930;

   strcpy (aMeinArray[2].Nachname, "Kohl");
   strcpy (aMeinArray[2].Vorname,  "Hannelore");
   aMeinArray[2].Gebjahr = 1933;

   strcpy (aMeinArray[3].Nachname, "Brandt");
   strcpy (aMeinArray[3].Vorname,  "Willy");
   aMeinArray[3].Gebjahr = 1913;

   strcpy (aMeinArray[4].Nachname, "Schröder");
   strcpy (aMeinArray[4].Vorname,  "Gerhard");
   aMeinArray[4].Gebjahr = 1944;

   qsort((void *)aMeinArray, ARRAYSIZE, sizeof(TAdresse),
sortAdressFunction);

   for (int i=0; i<ARRAYSIZE; i++)
   {
      printf ("%s, %s, %d\n", aMeinArray[i].Nachname,
aMeinArray[i].Vorname,

aMeinArray[i].Gebjahr);
   }
   return 0;
}

int sortAdressFunction (const void *a, const void *b)
{
  TAdresse x = *((TAdresse *)a);
  TAdresse y = *((TAdresse *)b);

  int nErgebnis = strcmp(x.Nachname, y.Nachname);
  if (!nErgebnis)
  {
     nErgebnis = strcmp(x.Vorname, y.Vorname);
     if (!nErgebnis)
     {
        if (x.Gebjahr < y.Gebjahr) nErgebnis = -1;
```

```
        if (x.Gebjahr > y.Gebjahr) nErgebnis =  1;
     }
  }
  return nErgebnis;
}
```

20.1.2.4. QUICKSORT FÜR CHAR-ARRAYS (C-STRINGS)

Nur marginal schwieriger ist Sortierung von einfachen C-Zeichenketten (char-Arrays).

Da es sich letztlich um zweidimensionale Arrays handelt, muss hier die maximale Größe der Strings angegeben werden.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define ARRAYSIZE 5
#define NAMESIZE  51

int sortCStringFunction (const void *a, const void *b);

int main(int argc, char* argv[])
{
   char aMeinArray [ARRAYSIZE] [NAMESIZE];

   strcpy (aMeinArray[0], "Donald");
   strcpy (aMeinArray[1], "Daisy");
   strcpy (aMeinArray[2], "Dagobert");
   strcpy (aMeinArray[3], "Gustav");
   strcpy (aMeinArray[4], "Daniel");

   qsort((void *)aMeinArray, ARRAYSIZE, sizeof(char)*NAMESIZE,
sortCStringFunction);

   for (int i=0; i<ARRAYSIZE; i++)
   {
      printf ("%s\n", aMeinArray[i]);
   }
   return 0;
}

int sortCStringFunction (const void *a, const void *b)
{
  char *x = (char *)a;
  char *y = (char *)b;
  return strcmp(x, y);
}
```

20.2. SORTIEREN UNTER C++

Alle Verfahren, die oben unter ANSI-C beschrieben wurden, sind natürlich auch unter C++ lauffähig, hinsichtlich der Ausgabe sind natürlich einige

Anpassungen durchzuführen (Verwendung von cout anstelle von printf) und auch die Angabe des Namespace ist bei einigen Compilern dann Pflicht.

20.2.1. IMPLEMENTATION EINES EINFACHEN SORTIERALGORITHMUS UNTER C++

Da die Implementation der einfachen Datentypen bereits oben behandelt wurde, werden im weiteren Verlauf nur die besonderen Datentypen von C++ behandelt.

20.2.1.1. BUBBLESORT FÜR C++-STRINGS

Die Verwendung von STL [8] -Strings macht die Sortierung von Zeichenketten erheblich einfacher, als die der einfachen C-Zeichenketten (char-Arrays) — hauptsächlich, weil die STL-Strings den Vergleich über die Vergleichsoperatoren ermöglichen.

```
#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

#define ARRAYSIZE 5

void bubblesort_string (string *pArray, const int pSize);

int main (int argc, char *argv[])
{
   string aMeinArray [ARRAYSIZE];

   aMeinArray[0] = "Donald";
   aMeinArray[1] = "Daisy";
   aMeinArray[2] = "Dagobert";
   aMeinArray[3] = "Gustav";
   aMeinArray[4] = "Daniel";

   bubblesort_string (aMeinArray, ARRAYSIZE);

   for (int i=0; i<ARRAYSIZE; i++)
   {
      cout << aMeinArray[i] << endl;
   }
   return 0;
}

void bubblesort_string (string *pArray, const int pSize)
{
   int i, j;
   string sBuffer;
```

[8] STL = Standard Template Library

```
   for (i=0; i<pSize-1; i++)
   {
      for (j=i+1; j<pSize; j++)
      {
         if (pArray[j] < pArray[i])
         {
             sBuffer   = pArray[i];
             pArray[i] = pArray[j];
             pArray[j] = sBuffer;
         }
      }
   }
}
```

20.2.1.2. DER BUBBLESORT-ALGORITHMUS ALS TEMPLATE

Betrachtet man die vorstehenden Implementationen für den Bubblesort-Algorithmus, so ist schnell ersichtlich, dass sich die verschiedenen Funktionen (bubblesort_string, bubblesort_int und bubblesort_double) nur im Datentyp des zu sortierenden Elementes unterscheiden. Ermöglicht wird dies allein durch den Umstand, dass die zu vergleichenden Elemente mit Hilfe der Vergleichsoperatoren bewertet werden können. Funktionen, die sich allein durch Datentypen unterscheiden, die in der Parameterleiste übergeben werden, sind ideale Kandidaten für eine Template-Implementierung.

Damit ein Algorithmus als Template funktionieren kann, ist er – anders als die Funktionen bubblesort_string, bubblesort_int und bubblesort_double sinnvollerweise in einer Headerdatei zu schreiben:

```
#ifndef _BUBBLESORT_H_
  #define _BUBBLESORT_H_

  template <class T> void bubblesort (T *pArray, const int
pSize)
  {
    int i, j;
    T tBuffer;
    for (i=0; i<pSize-1; i++)
    {
      for (j=i+1; j<pSize; j++)
      {
        if (pArray[j] < pArray[i])
        {
           tBuffer   = pArray[i];
           pArray[i] = pArray[j];
           pArray[j] = tBuffer;
        }
      }
    }
  }
```

```
#endif
```

Ein so implementierter Sortieralgorithmus funktioniert für alle Datentypen, die sich über die Vergleichsoperatoren definieren (int, long, short, double, float, long double und einfache char, sowie deren unsigned Varianten, aber nicht für Arrays!) , also auch für selbst geschriebene Klassen, wenn für zwei Elemente dieser Klasse die Zuweisung und die Vergleichsoperatoren per Operator-Overloading festgelegt wurden.

*Weil sich gemäß der oben dargelegten Definitionen und Algorithmen die Bubblesort-Versionen für C-Strings (char *) und Strukturen (struct) nicht mit den Vergleichsoperatoren (C-Strings) bzw. nicht durch einen einfachen Vergleich (Strukturen benötigen Vergleiche über ihre Komponenteninhalte) bestimmen lassen, kann auf diese Fälle das Template nicht sinnvoll angewendet werden.*

20.2.2. VERWENDUNG DES QUICKSORT-ALGORITHMUS UNTER C++

Da der Qsort-Algorithmus für einfache Datentypen bereits oben behandelt wurde, werden hier nur besondere Datentypen von C++ behandelt.

QUICKSORT FÜR C++-STRINGS

Die Vereinfachung die für den Bubblesort gilt, trägt natürlich auch im Quicksort.

```
#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

#define ARRAYSIZE 5

int sortStringFunction (const void *a, const void *b);

int main(int argc, char* argv[])
{
   string aMeinArray [ARRAYSIZE];

   aMeinArray[0] = "Donald";
   aMeinArray[1] = "Daisy";
   aMeinArray[2] = "Dagobert";
   aMeinArray[3] = "Gustav";
   aMeinArray[4] = "Daniel";
```

```
   qsort((void *)aMeinArray, ARRAYSIZE, sizeof(string),
sortStringFunction);

   for (int i=0; i<ARRAYSIZE; i++)
   {
      cout << aMeinArray[i] << endl;
   }
   return 0;
}

int sortStringFunction (const void *a, const void *b)
{
  string x = *(string *)a;
  string y = *(string *)b;
  if (x<y) return -1;
  if (x>y) return  1;
  return 0;
}
```

Die Bewertungsfunktion für qsort kann leider nicht über ein template-Makro automatisiert werden, da der Callback-Mechanismus zwingend die immer gleiche Funktionssignatur [9] (= Datentypen der Parameter) erwartet. Ein Template hingegen braucht mindestens einen flexiblen Datentyp in der Übergabe, anhand dessen er die zu verwendenden Datentypen (quasi als Vorlage) ermitteln kann.

[9] Unter der Signatur einer Funktion versteht man die Anzahl und Datentypen der Übergabeparameter.

21. PARAMETERÜBERNAHME VON DER DOS-EBENE

Wenn bei Programmstart Daten von der DOS-Ebene übergeben werden sollen, wie es z.B. bei DOS-Befehlen üblich ist, so ist dies in C recht einfach zu realisieren.

Der Compiler übergibt die hinter dem Programmnamen aufgeführten Parameter mittels der Variablen *argc* (= Argument Count) und *argv* (= Argument Value) an die Funktion *main* (also das Hauptprogramm). Die beiden Namen *argc* und *argv* sind natürlich nicht verbindlich, haben sich aber allgemein eingebürgert und werden daher fast ausschließlich für diesen Zweck verwendet. Beide Variablen müssen als Parameter von *main* auch entsprechend deklariert werden:

```
//============================================================
// Programm DOSPARAM.CPP
//============================================================

#include <iostream>

using namespace std;

void main (int argc, char *argv [])
{
   int i;

   if (argc == 1)
   {
      cout << "Nur ein Parameter !" << endl;
   }
   for (i=0; i<argc; i++)
   {
      cout << "Parameter: %s" << argv [i] << endl;
   }
}
```

Nachdem Programmstart enthält *argc* die Anzahl der angegebenen DOS-Parameter. Das Feld von Strings *argv* hingegen enthält die eigentlichen Parameter als Texte. Zu beachten ist hierbei, dass *argc* immer mindestens den Wert Eins hat, da der Programmstart zumindest ein Argument liefert (und zwar den Namen des gestarteten Programms selbst). D.h. erst wenn *argc* größer als Eins ist, wurden auch tatsächlich weitere Parameter übergeben. Die dem Programmnamen folgende Parameterliste kann vom Programmierer nach Belieben weiterverarbeitet werden.

Die Variable *argv* ist vom Typ her ein Pointer auf ein Feld von Zeigern auf Strings. Den Inhalt und die sich ergebende Struktur muss man sich, für den Beispielaufruf unten, wie in nachstehender Tabelle vorstellen.

```
Programmaufruf:
   c:> PROGRAM PARM1 xyz
```

Übergabe von DOS-Parametern			
Variable	Adresse	Inhalt	Typ
argc	1000	3	Integer
argv	1002	1006	Zeiger auf Zeiger auf Char
argv[0]	1006	1018	Zeiger auf Char
argv[1]	1010	1040	Zeiger auf Char
argv[2]	1014	1046	Zeiger auf Char
	1018	"C:\\PROG\\PROGRAM.EXE"	String
	1040	"PARM1"	String
	1046	"xyz"	String

Tabelle 21-1: Übergabe von DOS-Parametern

Sollen Werte, d.h. Zahlen übergeben werden, so müssen die Zeichenketten erst umgewandelt werden. Dazu eignen sich z.B. die Funktionen *atoi*, *atol*, *atof* und *sscanf*.

Die Funktion *atoi* wandelt einen String, der einen Integerwert enthält (*signed int*) in den entsprechenden Zahlenwert um. Das Unterprogramm *atol* realisiert die Umwandlung auf *signed long int* und *atof* gibt eine Fließkommazahl vom Typ *double* zurück.

Das Unterprogramm *sscanf* ist identisch mit den Funktionen *scanf* und *fscanf*, nur dass hier die Daten nicht von der Tastatur bzw. aus einer Datei eingelesen werden, sondern aus einer Zeichenkette.

```
Syntax :
   Integervar = atoi (String);
   Longvar    = atol (String);
   Doublevar  = atof (String);

   sscanf (String, "%d%f%s", Integervar, Floatvar,
           Stringvar);
```

Zu beachten ist, dass beim Einlesen von der DOS-Ebene alle Leerzeichen als Trennzeichen zwischen Parametern betrachtet werden. Soll ein String mit darin enthaltenen Leerzeichen eingelesen werden, so ist der Text bei der Übergabe in Hochkommata zu setzen:

```
PROGRAM "PARM1 xyz"
```

Das Beispiel erzeugt nur einen Eingabeparameter, die Variable *argc* enthält entsprechend den Wert Zwei.

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS