Softwareentwicklung in C

Markus Zuber Harald Glock

3. November 2002

Inhaltsverzeichnis

1. Vorwort **6**

2. Einführung **7**
2.1. Programmiersprachen . . . 7
2.1.1. Klassifizierung durch Hardwarenähe . . . 7
2.1.2. Entwicklung der Programmiertechnik . . . 7
2.2. Entwicklungsgeschichte von C . . . 8
2.3. Programmerstellung . . . 10
2.3.1. Source-Code . . . 10
2.3.2. Compiler/Interpreter/Assembler . . . 11
2.3.3. Object-Code . . . 11
2.3.4. Library . . . 12
2.3.5. Linker . . . 12
2.3.6. Preprocessor . . . 12
2.3.7. Debugger . . . 12
2.4. Algorithmen . . . 13
2.4.1. Ablaufstrukturen . . . 15
2.4.2. Struktogramme . . . 16
2.4.3. Ablaufplan . . . 17
2.4.4. Datenflußplan . . . 18
2.4.5. Vom Problem zum Programm . . . 20
2.5. DV-Grundlagen . . . 23
2.5.1. Bit und Byte . . . 24
2.5.2. Binär-Zahlensystem . . . 24
2.5.3. Hexadezimal-Zahlensystem . . . 26
2.5.4. Negation von Zahlen durch Zweierkomplement . . . 26

3. Aller Anfang ist schwer... **28**
3.1. Das erste C-Programm . . . 28
3.1.1. Grundstruktur . . . 28
3.1.2. Kommentare . . . 29
3.1.3. Layout . . . 29
3.1.4. Bildschirmausgabe . . . 29

4. Datentypen, Variablen, Konstanten **31**
4.1. elementare Datentypen . . . 31

4.1.1. Wertebereiche 33
4.2. Variablen 35
4.2.1. Deklaration von Variablen 35
4.2.2. Wertzuweisung 36
4.2.3. Datentypumwandlung 37
4.2.4. Bildschirmausgabe 40
4.2.5. Tastatureingabe 45
4.3. Konstanten 48
4.3.1. Zeichenkonstanten 48
4.3.2. Ganzzahlige Konstanten 49
4.3.3. Gleitpunkt Konstanten 50
4.3.4. symbolische Konstanten 50
4.3.5. Aufzählungstypen 51

5. Ausdrücke und Operatoren **54**
5.1. Ausdruck und Anweisung 54
5.2. Zuweisungsoperator 54
5.3. Arithmetische Operatoren 54
5.3.1. binäre arithmetische Operatoren 55
5.3.2. Unäre arithmetische Operatoren 56
5.4. Logische- und Vergleichsoperatoren 57
5.5. Bitoperatoren 59
5.5.1. Setzen von Bits 61
5.5.2. Löschen/Prüfen von Bits 61
5.5.3. Invertieren von Bits 62
5.6. Zusammengesetzte Zuweisungsoperatoren 63
5.7. Priorität und Anwendbarkeit von Operatoren 63
5.8. lvalue 64

6. Kontrollstrukturen **65**
6.1. Blöcke 65
6.2. Verzweigungen 66
6.2.1. Einfache Verzweigung 66
6.2.2. mehrfache Verzweigung 71
6.3. Wiederholungen 73
6.3.1. Zählergesteuerte Schleife 73
6.3.2. kopfgesteuerte Schleife 76
6.3.3. fußgesteuerte Schleife 78
6.3.4. break und continue 79
6.3.5. Sprünge mit goto 81
6.3.6. Endlosschleifen 81

7. Felder **83**
7.1. eindimensionale Felder 83
7.2. mehrdimensionale Felder 84

7.3. Initialisierung von Feldern . . . 84
7.3.1. Initialisierung eindimensionaler Felder . . . 85
7.3.2. Initialisierung mehrdimensionaler Felder . . . 85
7.4. Zugriff auf Elemente . . . 86
7.5. Zeichenketten . . . 88
7.5.1. Besonderheiten von C-Zeichenketten . . . 88
7.5.2. Deklaration von Zeichenketten (Strings) . . . 89
7.5.3. Initialisierung von Zeichenketten . . . 89
7.5.4. Eingabe von Zeichenketten über die Tastatur . . . 90
7.5.5. Die Headerdatei <string.h> . . . 91

8. Funktionen **96**
8.1. Aufbau . . . 96
8.1.1. Parameter . . . 96
8.2. Funktionsprototypen . . . 97
8.2.1. Header-Dateien . . . 98
8.3. Gültigkeit von Variablen . . . 98
8.3.1. Globale Variablen . . . 98
8.3.2. Lokale Variablen . . . 99
8.3.3. Static Variablen . . . 100
8.4. Aufruf von Funktionen . . . 100
8.4.1. Call-by-value . . . 100
8.4.2. Übergabe mit C++ Referenzen . . . 101
8.4.3. Auswertung der Parameter . . . 102
8.5. Modulare Programmierung in C . . . 103

9. Strukturen **105**
9.1. Definition . . . 105
9.2. Darstellung im Datenhierarchie Diagramm . . . 106
9.3. Verwendung . . . 106
9.3.1. Deklarieren und Initialisieren von Strukturvariablen . . . 106
9.3.2. Zugriff auf Strukturelemente . . . 107
9.3.3. Felder von Strukturvariablen . . . 108

A. ASCII-Tabelle **110**

B. Arbeiten mit Microsoft Visual C++ **111**
B.1. Programmerstellung . . . 112
B.2. Fehlersuche . . . 116
B.3. Debugging . . . 116

C. Übungen **117**
C.1. Übungen zu Zahlensystemen . . . 117
C.2. Übungen zum Einstieg . . . 118
C.3. Übungen zu Datentypen . . . 120

C.4. Übungen zu Operatoren 121
C.5. Übungen zu Verzweigungen 129
C.6. Übungen zu Wiederholungen 134
C.7. Übungen zu Feldern 138
C.8. Übung zu Zeichenketten 139
C.9. Übungen zu Funktionen 140
C.10. Übungen zu Strukturen 143

1. Vorwort

Dieses Skript dient als Unterrichtsgrundlage für die Vorlesung „Informatik“ an der Berufsakademie. Der Stoff beinhaltet eine Einführung in die Programmierung mit der Hochsprache C.
Das Skript soll dabei als Ergänzung und Nachschlagewerk für die ersten beiden Semester dienen. Es beinhaltet viele Übungen, jedoch wenig Quelltext. Aus eigener Erfahrung und Rückmeldungen von vorigen Semestern zeigte sich, dass stupides Abtippen oder Herunterladen fertiger Quelltexte zwar Zeit spart, jedoch nicht den gewünschten Lerneffekt bietet. Allein das selbständige Entwerfen der Programme und damit alle Fehler in der Denk- und Schreibweise führen zu einer sicheren Programmierung in jeder Progammiersprache.
Die Übungen in den einzelnen Kapiteln sind zur Vertiefung des behandelten Stoffes gedacht. Sie werden zum größten Teil nicht während der Vorlesung behandelt und sollen als Anregung für weitere Programmierübungen außerhalb der Vorlesung dienen. Programmieren erlernt man nicht durch Lesen vieler dicker Schmöker, sondern allein durch die Praxis!! Dabei müssen nicht immer große Ziele im Vordergrund stehen, denn oft liegen schon in kleinen Programmen viele Grundlagen für größere Projekte verborgen.

2. Einführung

2.1. Programmiersprachen

Der Begriff Programmiersprache kennzeichnet ganz allgemein eine Sprache zur Formulierung von Rechenvorschriften für einen Computer. Eine Programmiersprache ist durch ihre Syntax[1] und Semantik[2] eindeutig definiert.
Genau wie im wirklichen Leben gibt es viele verschiedene Programmiersprachen, die in bestimmte Klassen eingeteilt werden:

2.1.1. Klassifizierung durch Hardwarenähe

Ein Kriterium ist die Nähe der Sprache zur Hardware. Es ist dabei entscheidend, ob der Sprachumfang so ausgelegt ist, dass er genau einen bestimmten Prozessor wiederspiegelt oder nicht. Entsprechend wird zwischen den **niederen** und den **höheren** Programmiersprachen unterschieden.

- *niedere Programmiersprachen*
  Diese Sprachen werden oft auch als Maschinen- oder Assemblerspachen bezeichnet. Diese Sprachen erlauben das direkte Programmierung der Zielhardware. Entsprechend ist auch der Befehlssatz exakt an die Möglichkeiten der Hardware angepasst. Der Assembler überträgt die eingegebenen Befehle direkt in die Bitfolgen für den Prozessor.

- *höhere Programmiersprachen*
  Diese Sprachen sind im Befehlsatz nicht mehr von einer bestimmten Hardware abhängig. Die Befehle werden nicht mehr 1:1 in Bitfolgen für den Prozessor übersetzt. Viel mehr arbeiten hier ein Interpreter oder ein Compiler daran, den eingegebenen Quellcode in die entsprechenden Bitfolgen der Zielhardware zu übersetzen.

2.1.2. Entwicklung der Programmiertechnik

Mit den Sprachen und der Komplexität der Rechner und Programme, haben sich auch die Programmiertechniken verändert.

[1] Definiert den Aufbau von Wörtern, Sätzen... ohne den einzelnen Konstrukten eine Bedeutung zu geben
[2] Ordnet den einzelnen Wörtern und Zeichen eine Bedeutung zu

- *unstrukturierte Programmierung*
  Diese Art der Programmierung ist besonders im Assembler-Bereich anzutreffen. Hier findet man im ganzen Code verteilt immer wieder Sprünge („GOTO“) an andere Stellen. So wird kreuz und quer durch den Programmcode gesprungen und es ist schwierig bis unmöglich, den Programmablauf zu durchschauen.

- *strukturierte Programmierung*
  Ein erster Besserungsschritt bestand in der Einführung der strukturierten Programmierung. Hier wird der Code in verschiedene Grundstrukturen wie Anweisungsblock, Verzweigung und Wiederholung eingeteilt. Durch Einführung von Unterprogrammen (Funktionen, Prozeduren) wurde das unkontrollierte Springen im Programmcode („GOTO“) abgelöst. Damit ist eine bessere Strukturierung und Übersicht des Codes möglich. Die wichtigsten höheren Programmiersprachen basieren auf diesem Prinzip.

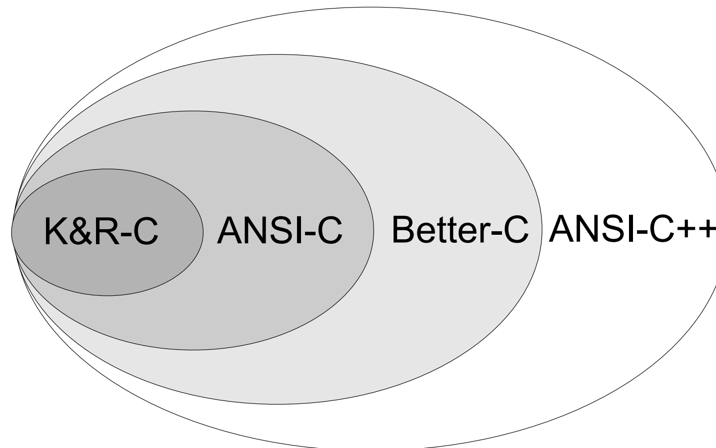
- *modulare Programmierung*
  Ein weiterer Schritt in eine bessere Strukturierung des Programms, war die Einführung der Modultechnik. Dabei werden bestimmte zusammengehörige Codeteile in andere Module ausgelagert. Dies erlaubt zum einen eine Wiederverwendung in anderen Projekten sowie eine bessere Kapselung zusammengehöriger Codeteile.

- *objektorientierte Programmierung*
  Diese Technik erweitert die modulare Programmierung dahingehend, dass jetzt Code und Daten zu so genannten Klassen zusammengefasst werden. Diese Stufe ist der bisher letzte Entwicklungsschritt. Da die objektorientierte Programmierung zu großen Teilen auf der strukturierten Programmierung basiert, soll in diesem Skript die strukturierte Programmierung erlernt werden. Außerdem ist der Einstieg hier etwas einfacher und ein späterer Umstieg zur Objektorientierung einfach machbar.

2.2. Entwicklungsgeschichte von C

Im Jahr 1969 erstellte Ken Thomson (Bell Laboratories) die erste Version von UNIX in Assembler. Das System war nicht portabel auf andere Hardwaresysteme!
Im Jahr 1970 entwickelte Ken Thomson die Sprache B als Weiterentwicklung der Sprache BCPL. B ist eine typlose Sprache, sie kennt nur Maschinenworte.
Die Programmiersprache C wurde im Jahr 1972 von Dennis Ritchie entworfen und erstmals implementiert. Die anfängliche Weiterentwicklung von C war stark an die des Betriebssystems UNIX gekoppelt.
Im Jahr 1978 folgte die Veröffentlichung des „Manuals“: „The C–Programming Language“ von B.W. Kernighan und D.M. Ritchie, welche den De–facto–Standard der damaligen UNIX–Implementierungen für C darstellte („Kernighan/Ritchie–C“).
Im Jahr 1988 folgte eine Überarbeitung des Standards durch das American National Standards Institute („ANSI–C“).
Im Jahr 1998 entwickelt Bjarne Stroustrup C hin zu einer objektorientierten Sprache. Dies

war die Geburt von C++. Auch hier gibt es ein Buch vom Entwickler, das als Standardwerk genannt werden soll: „The C++ Programming Language“ von Bjarne Stroustrup.
Inzwischen vermischen sich ANSI-C und C++ immer mehr. Das liegt auch daran, dass die heutigen Compiler meist beide Spracherweiterung kennen und somit nicht mehr genau unterschieden wird, ob der Code jetzt reines ANSI-C oder C++ ist.



Dem vorliegenden Skript ist der ANSI–Standard mit einigen üblichen C++-Elementen zugrunde gelegt.

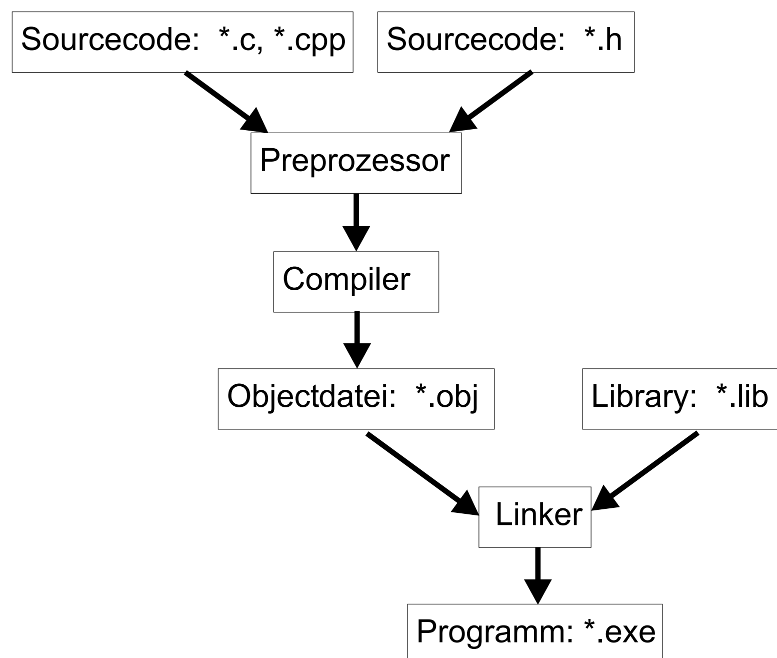
C nimmt eine Zwischenstellung zwischen Assembler und Hochsprache ein. Genau genommen, lassen sich sogar beide Sprache mischen. So vereint C zwei an sich widersprüchliche Eigenschaften: gute Anpassung an Hardware und Portabilität.
C-Compiler erzeugen sehr effizienten Code (sowohl bzgl. Laufzeit als auch bzgl. Programmgröße), daher kann für die meisten Anwendungsfälle auf den Einsatz eines Assemblers verzichtet werden. Die einfachere Programmierung bedingt kurze Entwicklungszeiten für die Software und die hohe Portabilität einfachere und vor allem schnellere Anpassungen an eine andere Hardware.
C ist eine vergleichsweise einfache Sprache (Daher nicht ganz einfach anzuwenden!!!). Sie kennt nur vier grundlegende Datentypen aber auch zusammengefaßte Datentypen wie Vektoren(Felder) und Strukturen sind möglich, allerdings mit weniger eingebauten Kontrollen als z.B. in PASCAL.
Es gibt einfache Kontrollstrukturen: Entscheidungen, Schleifen, Zusammenfasssungen von Anweisungen (Blöcke) und Unterprogramme.

Die Programmiersprache C findet heutzutage eine immer weitere Verbreitung, da sie einerseits fast so flexibel wie ein Assembler ist und andererseits über viele Möglichkeiten heutiger moderner Hochsprachen verfügt. Das hat dazu geführt, dass sich C besonders als Sprache für Betriebssystementwicklungen eignet (für die sie ja auch entwickelt wurde). Weiterhin lassen sich numerische Verfahren, Textverarbeitung und Datenbanken effizient in C realisieren.

C gehört zu UNIX in dem Sinne, dass UNIX ohne C kaum vorstellbar ist (wohl aber C ohne UNIX). So ist z.B. die Syntax des UNIX Dienstprogrammes awk in vielen Dingen identisch mit der von C. Selbst das Lesen und Verstehen der UNIX-Dokumentation ist an vielen Stellen ohne C-Kenntnisse nicht möglich.
Eine Programmierung in C ist immer dann angebracht, wenn sehr portabler Code erzeugt werden soll, da keine andere Sprache (trotz Normung) dazu in dem Maße in der Lage ist wie C.
Mittlerweile gibt es viele verschieden C-Implementationen für nahezu jedes Betriebssystem und Mikrocontroller. Daher wird die Verbreitung immer weiter fortschreiten.

2.3. Programmerstellung

Die Erstellung eines ablauffähigen Programms in einer Hochsprache ist ein komplexer Vorgang. Das gesamte Programm besteht aus verschiedenen Teilen, die nun im Einzelnen näher betrachtet werden sollen.



2.3.1. Source-Code

Der Begriff Source-Code bezeichnet den Programmcode in einer bestimmten Programmiersprache. Es ist nicht von Belang, um welche Programmiersprache es sich hierbei handelt. Der Source-Code wird nicht direkt vom Computer verstanden, sondern muss erst über einen oder mehrere Schritte für den Computer verständlich gemacht werden. Eine Datei mit Source-Code ist zumeist eine Textdatei, die mit jedem beliebigen Texteditor erstellt werden kann. Allerdings gibt es viele hilfreiche Editoren, die den Programmierer

schon beim Erstellen des Codes mit kleinen Hilfen unterstützen.
In der Programmiersprache C gibt es hauptsächlich zwei Source-Code Dateitypen mit den Endungen C bzw. CPP und H. Die genauen Unterschiede werden noch behandelt.

2.3.2. Compiler/Interpreter/Assembler

Wie schon erwähnt, ist der reine Source-Code für einen Prozessor nicht verständlich. Er muss noch in die binäre Maschinensprache übersetzt werden. Dies kann auf unterschiedlichen Wegen geschehen:

1. *Interpreter*
   Als Interpreter wird ein Programm bezeichnet, das Source-Code Stück für Stück (meistens Zeile für Zeile) übersetzt und auch sofort ausführt. Aus diesem Grund sind Interpretersprachen (z.B. BASIC, Java) auch in der Ausführung langsamer als Sprachen, die einen Compiler benutzen, da die Übersetzung sich direkt auf die Laufzeit niederschlägt. Ein Vorteil ist die Möglichkeit, einen hardwareunabhängigen Code zu erzeugen und erst zur Laufzeit, über den Interpreter, das Programm auszuführen. Nach diesem Prinzip arbeitet z.B. Java und hat sich deshalb im Internet schnell verbreitet.

2. *Compiler*
   Als Compiler wird ein Übersetzer bezeichnet, der Source-Code in Object-Code (siehe 2.3.3) transferiert. Im üblichen Fall wird je eine Datei mit Source-Code vom Compiler in je eine Datei mit Object-Code übersetzt. Dementsprechend können bei einem Compilerlauf viele Object-Files entstehen. Der Vorteil des Compilers ist die Möglichkeit, sehr schnellen, optimierten und kleinen Code für eine bestimmte Zielhardware zu erzeugen.

3. *Assembler*
   Im Prinzip ist die Aufgabe eines Assemblers dieselbe wie die eines Compilers. Auch er übersetzt Source-Code in Object-Code. Der einzige Unterschied zwischen einem Compiler und einem Assembler ist, dass ein Compiler Source-Code aus höheren Programmiersprachen übersetzt. Ein Assembler übersetzt Source-Code, der in Maschinensprache (siehe niedrige Programmiersprachen) geschrieben ist. Oft erzeugt heute noch ein Compiler aus der Hochsprache den Source-Code für einen Assembler.

2.3.3. Object-Code

Der Begriff Object-Code bezeichnet Maschinencode, der noch nicht direkt ausführbar ist, da er teilweise noch aus dem Source-Code übernommene symbolische Namen (z.B. Funktionsaufrufe) enthält. Der Computer versteht aber intern keine symbolischen Namen, sondern nur Adressen. Daher stellt der Object-Code auch nur ein Zwischenprodukt dar, das erst in einem weiteren Verarbeitungsschritt (durch den Linker) zu einem ausführbaren Programm wird. Object-Code entsteht als Resultat der Übersetzung von Source-Code

durch einen Compiler oder Assembler.
Dateien mit Object-Code erhalten üblicherweise die Endung OBJ.

2.3.4. Library

Als Library wird eine Zusammenfassung von nützlichen Funktionen, Prozeduren etc. bezeichnet, die in verschiedenen Programmen verwendet werden können. Im Prinzip liegt eine Library selbst als Object-Code vor. Und natürlich ist auch hier wieder der Linker verantwortlich, dass eine verwendete Library zum ausführbaren Programm dazu gebunden wird.
Prinzipiell gibt es statische und dynamische Libraries. Statische Libraries werden zum Zeitpunkt des Linkens in das ausführbare Programm übernommen, machen dieses also dementsprechend größer. Dynamische Libraries werden nicht direkt zum Programm gelinkt, sondern es werden nur die entsprechenden Einstiegspunkte im Programm vermerkt. Dynamische Libraries müssen immer zur Laufzeit eines Programms am entsprechenden Zielrechner vorhanden sein, da es sonst nicht ausführbar ist.
Dateien mit Bibliotheksfunktionen erhalten üblicherweise die Endung LIB (statisch) oder DLL (dynamisch).

2.3.5. Linker

Der Linker ist dafür verantwortlich, ein oder mehrere Files mit Object-Code und benötigte Libraries zu einem tatsächlich am Computer ausführbaren Programm zu binden. Hierbei muss er die noch im Object-Code vorhandenen symbolischen Namen auflösen und durch entsprechende Adressen ersetzen. Auch muss der Linker die korrekten Einstiegspunkte bei Verwendung von dynamischen Libraries erzeugen.

2.3.6. Preprocessor

In einigen Sprachen, wie z.B. auch in C, wird der Source-Code noch von einem Preprocessor behandelt, bevor er dem Compiler zur tatsächlichen Übersetzung übergeben wird. Der Preprocessor reagiert auf bestimmte Textsequenzen im Source-Code und ersetzt diese textuell nach bestimmten Regeln. Diese Sequenzen stellen also eine Art von Macros dar, durch deren Auswertung neuer bzw. veränderter Source-Code erzeugt wird. Das Resultat, das aus dem Preprocessor kommt, ist selbst wieder Source-Code, nur in veränderter Form. Dieser veränderte Source-Code wird dem Compiler zur Übersetzung übergeben.

2.3.7. Debugger

Eines von vielen Werkzeugen, die den Prozess des Testens erleichtern, ist der Debugger. Er gestattet es, Programme schrittweise (z.B. Zeile für Zeile) auszuführen, eigene Breakpoints im Programm zu setzen (das Programm wird bis dorthin ausgeführt, dann wird die Ausführung unterbrochen), Variableninhalte zu inspizieren u.v.m. Was genau ein Debugger kann, ist durch die jeweilige Implementation bestimmt.

Bereits in Vergessenheit geraten ist der Ursprung des Namens BUG für einen Programmfehler (von dem sich auch der Name Debugger ableitet): Zu Zeiten, als Computer noch raumfüllende Monster waren, die vollständig aus einzelnen Transistoren aufgebaut waren, kam es des Öfteren vor, dass sich darin Ungeziefer einnistete. Die dadurch entstehenden Kurzschlüsse waren der Korrektheit der ausgeführten Programme nicht unbedingt förderlich, wie man sich leicht vorstellen kann. Die Fehlerbehebung war in diesem Fall mit einer Entfernung des Ungeziefers aus dem Rechner verbunden (echtes Debugging eben...)

2.4. Algorithmen

Ein Computerprogramm wird immer, egal in welcher Programmiersprache, Schritt für Schritt abgearbeitet. Einer solchen Abarbeitung liegt ein Konzept zugrunde, das die Problemlösung beschreibt (=Algorithmus), und eine Formulierung dieses Konzepts in einer Form, die durch die gewählte Programmiersprache vorgegeben ist. Bei programmiersprachlichen Formulierungen gibt es im Prinzip immer zumindest ein Konzept, wie man eine Reihe von Anweisungen zu einem logischen Ganzen zusammenfassen kann.

Der Begriff des Algorithmus bezeichnet eine Verarbeitungsvorschrift, die die Problemlösung beschreibt. Diese Verarbeitungsvorschrift ist so präzise formuliert,dass sie direkt in ein Computerprogramm umgesetzt werden kann. Das bedeutet: Alle möglichen Fälle, die dabei durchzuführenden Schritte und die Randbedingungen bei der Ausführung müssen genau beschrieben sein.
Allen Verfahren, wie z.B. Suchverfahren, Sortierverfahren oder auch der Durchführung von mathematischen Operationen, liegen Algorithmen, also genaue Verarbeitungsvorschriften, zugrunde. Algorithmen sind nicht in einer bestimmten Programmiersprache formuliert, sondern, je nach Problemstellung, mathematisch, andersartig formal oder auch umgangssprachlich.

Beispiele für Algorithmen:

- aus dem täglichen Leben:
  1. Bedienungsanleitungen
  2. Bauanleitungen
  3. Kochrezepte
  4. Spielregeln
- aus dem „Schulwissen“:
  1. die Berechnung der Zahl e=2,7182...
  2. der Test, ob eine Zahl eine Primzahl ist
  3. die Bestimmung eines Schaltjahres
  4. die Addition bzw. Multiplikation von Dualzahlen

Solch ein Verfahren muss bestimmte Eigenschaften besitzen, damit ein Bearbeiter, ob Mensch oder Computer, auch immer das gewünschte Ergebnis erzielt:

- Die Verfahrensbeschreibung muss an jeder Stelle eindeutig festlegen, welcher Bearbeitungsschritt als nächstfolgender auszuführen ist.
- Jeder Bearbeitungsschritt in der Folge muss auch ausführbar sein.
- Die Verfahrensbeschreibung muss eine endliche Länge besitzen.
- Jeder Bearbeitungsschritt muss festlegen, was zu tun ist und womit (mit welchen Daten) der Schritt durchzuführen ist.
- Vollständigkeit der Beschreibung
  Es muss eine komplette Anweisungsfolge vorliegen. Eine Bezugnahme auf unbekannte Information darf nicht enthalten sein.
- Wiederholbarkeit des Algorithmus
  Im Sinne eines physikalischen Experiments muss man die Ausführungen eines Algorithmus beliebig oft wiederholen können und jede Wiederholung muss bei gleichen Eingabedaten das gleiche Resultat liefern (Reproduzierbarkeit). Ist ein Algorithmus endlich und definiert so ergibt sich diese Forderung automatisch.
- Korrektheit des Algorithmus
  Diese Forderung ist zwar selbstverständlich, aber in der Praxis ist die Korrektheit nur sehr schwer nachzuweisen. Man bedient sich daher Tests, bei denen für die vorgegebenen Eingabedaten das Ergebnis bereits bekannt ist, und vergleicht dann die erzeugten Ausgabedaten mit den Ergebnissen. (Ein solcher Test ist insofern problematisch, da alle möglichen Fälle abgedeckt werden müssen. Im Extremfall muss dieser Test sogar für jede mögliche Eingabe durchgeführt werden.)

Zusammenfassen lassen sich alle diese Eigenschaften in einer kurzen aber präzisen Definition des Begriffs Algorithmus.

Algorithmus
Eine Bearbeitungsvorschrift heißt Algorithmus, wenn sie folgende Eigenschaften erfüllt:

1. Die Vorschrift ist mit endlichen Mitteln beschreibbar.
2. Sie liefert auf eine eindeutig festgelegte Weise zu einer vorgegebenen Eingabe in endlich vielen Schritten genau eine Ausgabe.

Der Begriff Algorithmus ist ein zentraler Begriff der Informatik. Er baut immer auf einer Menge von Grundoperationen auf. Als Beispiel soll ein Algorithmus aus der Mathematik betrachtet werden: Die näherungsweise Berechnung des Kreisumfangs.

Algorithmus “Kreisumfang“:

(K1) Nimm den Radius aus der Eingabe.

(K2) Multipliziere den Radius mit 2,0 und nenne das Ergebnis ’Durchmesser’.

(K3) Multipliziere den Durchmesser mit 3,1415926 und bringe das Ergebnis in die Ausgabe.

Dieser Algorithmus liefert beispielsweise zur Eingabe 7,0 die Ausgabe 43,982296. Wenn der Computer das Multiplizieren nicht beherrscht, dann müsste man ihm noch einen Algorithmus, ein Verfahren, für deren Durchführung geben.

Algorithmus ”Multiplikation”:

(M1) Schreibe die beiden Zahlen aus der Eingabe nebeneinander.

(M2) Nimm die erste Ziffer der zweiten Zahl.

(M3) Schreibe die erste Zahl sooft untereinander, wie die Ziffer der zweiten Zahl angibt, und zwar so, dass die letzte Ziffer unter der betrachteten Ziffer der zweiten Zahl steht. Für den Fall, dass besagte Ziffer 0 ist, schreibe 0.

(M4) Falls noch Ziffern in der zweiten Zahl vorhanden sind, nimm die nächste Ziffer und gehe nach M3.

(M5) Addiere alle mit M3 erzeugten Zahlen.

(M6) Zähle bei dem Ergebnis so viele Stellen von rechts ab, wie beide Eingabewerte zusammen an Stellen hinter dem Komma besitzen. Setze hier das Komma im Ergebnis.

(M7) Bringe dies Endergebnis in die Ausgabe.

2.4.1. Ablaufstrukturen

Seit der Einführung der strukturierten Programmierung wurden die Programme und die Algorithmen in verschiedene Strukturen unterteilt. Eine Aneinanderreihung dieser weniger Strukturen führt schließlich zum fertigen Algorithmus. Dabei dienen nur wenige Strukturen zur Darstellung des Algorithmus:

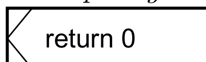
- Sequenz oder Block
- einfache oder mehrfache Verzweigung
- kopf-, fuss- oder zählergesteuerte Schleife

2.4.2. Struktogramme

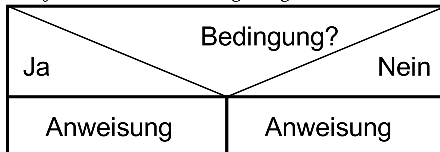
Die Darstellung des Algorithmus in einem Struktogramm nach DIN66261 ist eine Möglichkeit. Diese Darstellungsform unterstützt die Forderung der strukturierten Programmierung, dass Programme von „oben nach unten" ablaufen, optimal. Durch folgende Elemente werden die einzelnen Ablaufstrukturen dargestellt:

- *Block/Sequenz*

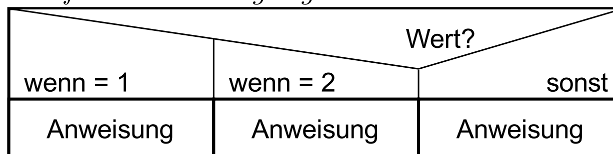
- *Rücksprung*



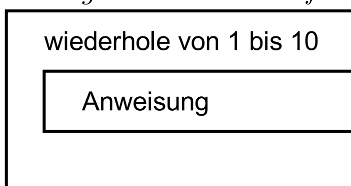
- *einfache Verzweigung*



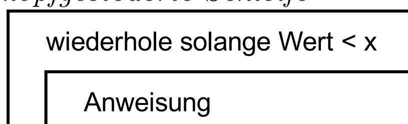
- *mehrfache Verzweigung*



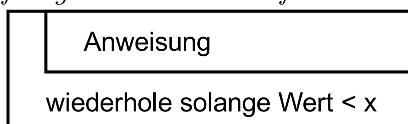
- *zählergesteuerte Schleife*



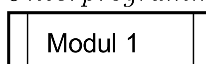
- *kopfgesteuerte Schleife*



- *fussgesteuerte Schleife*



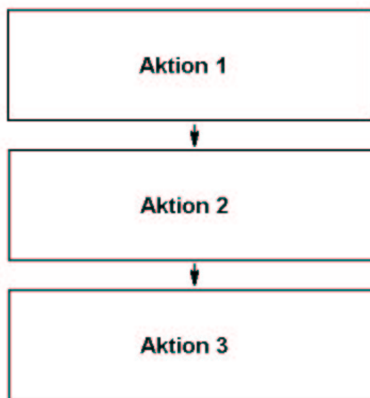
- *Unterprogrammaufruf*



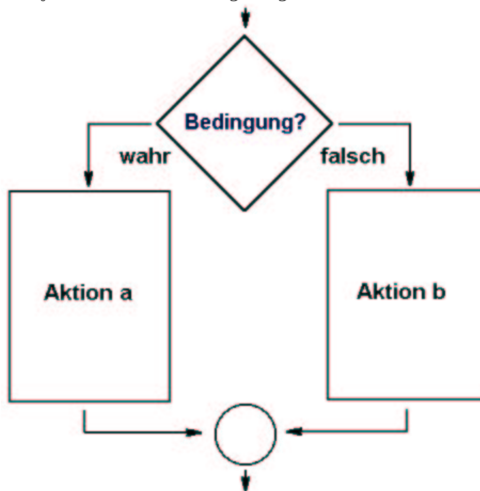
2.4.3. Ablaufplan

Eine andere Möglichkeit zur Darstellung ist der Ablaufplan nach DIN66001. Durch die Einführung des Struktogramms hat die Bedeutung des Ablaufplans abgenommen. Er besteht aus verschiedenen geometrische Formen, die über Pfeile miteinander verbunden sind. Folgende Elemente stehen zur Verfügung:

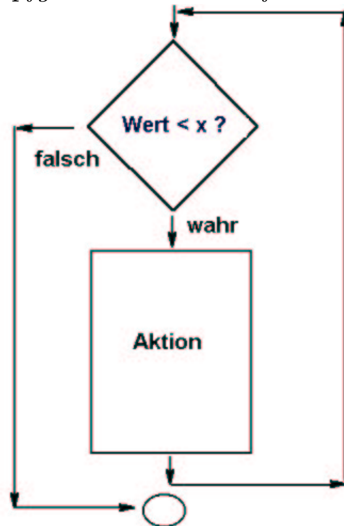
- *Block/Sequenz*



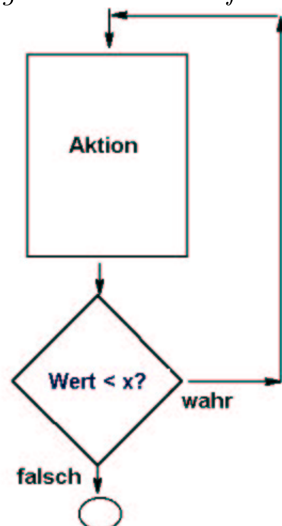
- *einfache Verzweigung*



- *kopfgesteuerte Schleife*

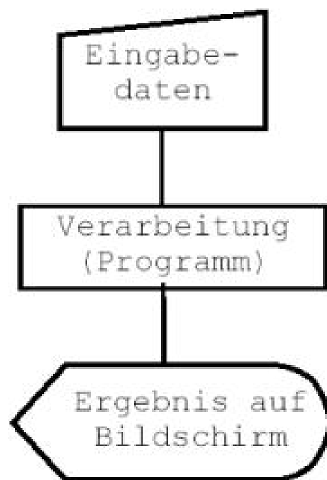


- *fussgesteuerte Schleife*



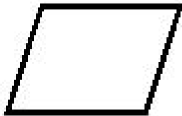
2.4.4. Datenflußplan

Ein Datenflussplan ist eine grafische Übersicht, welche die Programme und Daten, die zu einer Gesamtaufgabe gehören, miteinander verbindet. Er zeigt, welche Teile eines Programms von den Daten durchlaufen werden und welche Art der Bearbeitung innerhalb der Programme vorgenommen wird. Ein Datenflussplan besitzt ähnliche Symbole wie ein Programmablaufplan. Zusätzliche Sinnbilder werden vor allem für die Daten eingeführt.



Folgende Elemente stehen zur Verfügung:

- *Daten allgemein*



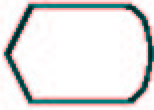
- *Daten auf einem Schriftstück (z.B. Druckliste)*

- *Daten auf einer Karte (z.B. Loch- oder Magnetkarte)*

- *Daten auf einem Speicher mit ausschließlich nacheinander zu verarbeitendem Zugriff (sequenziell), z.B. ein Magnetband*

- *Daten auf einem Speicher mit direktem Zugriff (wahlweise), z.B. eine Festplatte*

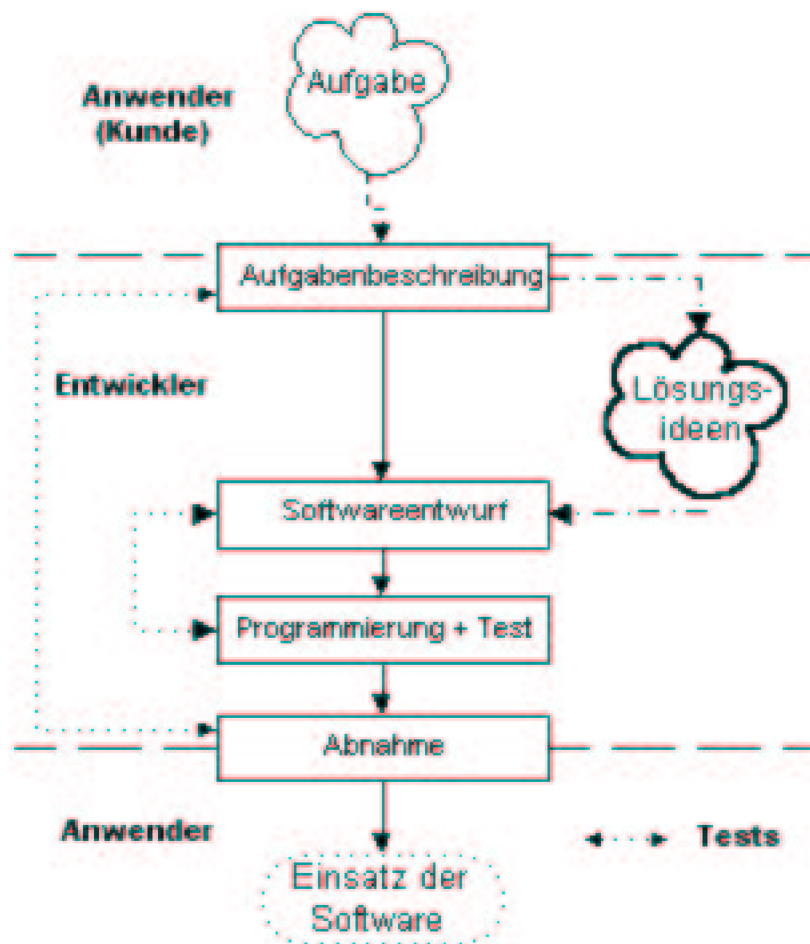
- *Maschinell erzeugte optische oder akustische Daten, z.B. die Bildschirmausgabe*



- *Manuelle Eingabe*

2.4.5. Vom Problem zum Programm

Bevor der Softwareerstellungsprozeß beginnt, ist das Wichtigste das Ziel festzulegen: Wofür und warum muss Software erstellt werden und was muss sie können, um diese Aufgabe zu erledigen. Im folgenden Bild ist der Softwareprozess stark vereinfacht in der Übersicht dargestellt:



Problemanalyse

Die wichtigste Phase beim Erstellen eines Algorithmus ist die Analyse der Problemstellung und das Nachdenken über die Lösung. Dieser Teil, die Problemanalyse und die nachfolgende Skizze des Lösungswegs, ist wesentlich wichtiger und zeitaufwendiger als die eigentliche Kodierung in irgendeiner Programmiersprache. Welche Phasen beim Entwickeln eines Algorithmus durchlaufen werden, soll an einem Beispiel gezeigt werden:

Gesucht wird ein Algorithmus zur Jahrestagberechnung. Zu einem eingegebenen Datum (Tag, Monat, Jahr) soll ausgerechnet werden, um den wievielten Tag im Jahr es sich handelt.

Der erste Schritt ist natürlich immer das Nachdenken über die Problemlösung. Diese Phase sollte nicht zu kurz sein, denn oft findet sich eine optimale Lösung erst nach dem zweiten oder dritten Ansatz. Und kürzere Programme sind nicht nur schneller, sondern haben (rein statistisch betrachtet) auch weniger Fehler. Dann sollte man den Lösungsweg skizzieren und dabei auch überlegen, ob auch alle Randbedingungen und Ausnahmesituationen berücksichtigt wurden.

Oft hilft es, den Algorithmus zuerst anhand eines Beispiels durch zuspielen. Dabei werden einem oft Zusammenhänge und Fallstricke klar. Versuchen wir das gleich mit der Problemstellung von oben:

- Datum = 7.7.99
- Jahrestag (JT) = 31 + 28 + 31 + 30 + 31 + 30 + 7 = 188

Das Verfahren addiert also bis zum Juni die Monatslängen und zählt dann noch die 7 Tage des Juli hinzu. Spätestens hier sollte einem einfallen, dass der Februar auch 29 Tage haben kann. Der Programmierer sollte sich hier eine Notiz machen: "Nachsehen, wann ein Jahr ein Schaltjahr ist".
Danach sollte eine problemnahe Darstellung des Algorithmus folgen. Diese sieht beispielsweise so aus:

1. Eingabe von Tag, Monat, Jahr.
2. Ermittle die Monatslänge des Februar (Schaltjahr?).
3. Addiere die Monatslängen der Vormonate (also bis Monat-1) und den Tageswert des eingegebenen Monats.
4. Gib das Ergebnis aus.

Dabei wird davon ausgegangen, dass die Eingabe korrekt ist (also nicht z. B. der 35.13.19999 eingegeben wird.
Für die Schaltjahresberechnung brauchen wir noch einen Teilalgorithmus. Selbst bei diesem recht einfachen Beispiel zeigt sich also schon ein Grundsatz der strukturierten Programmierung, die Zerlegung einer Aufgabe in kleinere und damit überschaubare Teilaufgaben.

Nach dem Gregorianischen Kalender handelt es sich um ein Schaltjahr,

- wenn die Jahreszahl ohne Rest durch 4 teilbar ist.
- nicht aber, wenn die Jahreszahl ohne Rest durch 100 teilbar ist.
- jedoch schon, wenn die Jahreszahl ohne Rest durch 400 teilbar ist.

Das Jahr 2000 ist also ein Schaltjahr (Erinnerung an den Jahreswechsel nach 2000 werden hier vielleicht wach). Der Algorithmus lautet also:

1. Eingabe Jahr.
2. Falls (Jahr mod 4) != 0: Ausgabe “NEIN“ und ENDE.
3. Falls (Jahr mod 100) != 0: Ausgabe “JA“ und ENDE.
4. Falls (Jahr mod 400) = 0: Ausgabe “JA“ sonst: Ausgabe ”NEIN”.

Anmerkungen: mod sei der Modulo-Operator = Rest der Ganzzahligen Division. != bedeutet „ungleich“.
Nun steht der Jahreszahlberechnung eigentlich nichts mehr im Weg. Die einzige Schwierigkeit liegt noch darin, dass die Monatslängen unterschiedlich sind:

Jan.	Feb.	März	April	Mai	Juni	Juli	Aug.	Sept.	Okt.	Nov.
31	28/29	31	30	31	30	31	31	30	31	30

Nun ließe sich ein Algorithmus niederschreiben, der 11 WENN-DANN-Abfragen enthält, welche die Tagessumme der Vormonate berechnen:

1. Eingabe Tag, Monat, Jahr.
2. Wenn Monat = 1 dann Jahrestag = Tag und ENDE.
3. Wenn Monat = 2 dann Jahrestag = 31 + Tag und ENDE.
4. Wenn Monat = 3 dann Jahrestag = 31 + 28 + Tag.
   Wenn Schaltjahr, erhöhe Jahrestag um 1.
   ENDE.
5. Wenn Monat = 4 dann Jahrestag = 31 + 28 + 31 + Tag.
   Wenn Schaltjahr, erhöhe Jahrestag um 1.
   ENDE.
6. ...

Damit ist die Aufgabe sicher zufriedenstellend gelöst.

Untersuchen wir einen zweiten Fall:

Gesucht wird ein Algorithmus, der feststellt ob eine gegebene Zahl eine Primzahl ist.

2.5. DV-Grundlagen

Im EDV-Bereich gibt es verschiedene Systeme zur Darstellung einer Zahl. Allen Zahlensysteme liegt ein Positionssystem zugrunde. Dies bedeutet, dass jeder Position in einer Zahl ein bestimmter Wert zugeordnet ist, der eine Potenz der Grundzahl ist.
Für das im alltäglichen Gebrauch verwendete Zehnersystem ist die Basis die 10. Es gibt daher Ziffern von 0 bis 9, aus welchen alle Zahlen aufgebaut sind. Die Zahl 574 ist eine Kurzform für $5 \cdot 100 + 7 \cdot 10 + 4 \cdot 1$ oder auch $5 \cdot 10^2 + 7 \cdot 10^1 + 4 \cdot 10^0$. Die am weitesten rechte Ziffer ist immer als Vielfaches von x^0, die davon linke Ziffer als x^1 usw. zu verstehen. x steht für die Basis des Zahlensystems, also z. B. die 10.
Die zehn verschiedenen Ziffern von 0 bis 9 sind im Rechner, der ja nur die Zustände 0 und 1 (ein oder aus...) kennt, schwer zu realisieren. Daher werden im Prozessor alle Zahlen im Dualsystem mit der Basis 2 dargestellt (siehe 2.5.2).

2.5.1. Bit und Byte

Obwohl es lächerlich erscheinen mag, die Begriffe Bit und Byte hier zu erwähnen, gibt es doch einen Grund dafür. Einerseits scheint es sich noch nicht überall herumgesprochen zu haben, dass ein Bit die kleinste Informationseinheit darstellt, mit der man nur noch 0 bzw. 1 darstellen kann, und dass ein Byte aus (üblicherweise) 8 Bits besteht. Andererseits, und das ist viel häufiger, werden nur zu gerne die falschen Buchstaben für ihre Bezeichnung herangezogen. Ein Bit wird mit dem Buchstaben b bezeichnet, und ein Byte mit dem Buchstaben B. Wenn man also im Kontext von Modems irgendwo den Begriff „bps“ liest, dann ist damit Bits pro Sekunde gemeint, niemals Bytes pro Sekunde, auch wenn das auf Prospekten nur allzu gerne verwechselt wird.

Lächerlich genug, dass Bit und Byte hier erklärt werden, jetzt auch noch der Begriff Kilobyte... Nun, es gibt da leider noch größere Verwirrung als bei Bits und Bytes. Einerseits würde ein kB (=Kilobyte) technisch gesehen genau 1000 Bytes bezeichnen, andererseits hat sich in der Computerwelt eingebürgert, dass ein kB 1024 Bytes bezeichnet, weil dies die nächstgelegene Zweierpotenz ist. Diese ist interessant, weil man es computerintern immer mit Binärzahlen zu tun hat. Dementsprechend werden auch oft für ein Megabyte 1024*1024 Bytes angenommen, das entspricht 1048576 Bytes.
Gleich noch verworrener wird es bei der Bezeichnung von Megabytes (MB): Manchmal werden 1000 kB als 1 MB bezeichnet, manchmal auch 1024 kB. Wenn man jetzt noch die beiden verschiedenen Definitionen des Kilobytes in die Betrachtung mit einbezieht, dann kann 1 MB alles zwischen 1000 * 1000 B, 1000 * 1024 B und 1024 * 1024 B sein. Zumeist ist der Unterschied zwar klein genug, aber unter gewissen Umständen sollte man doch im Vorhinein abklären, mit welcher Definition man es gerade zu tun hat.

2.5.2. Binär-Zahlensystem

Wie schon erwähnt arbeitet der Prozessor intern im Binär- oder Dualsystem. Hier liegt jeder Zahl die Basis 2 zugrunde. Daraus folgt, dass es nur die zwei Ziffern 0 oder 1 zur Darstellung einer Zahl gibt. Die Folge daraus ist, dass eine Zahl sehr schnell viele einzelne Ziffern erhält. So wird z.B. die Zahl 19_{10} im Dualsystem als 10011_2 dargestellt.

Umwandlungen zwischen Binärsystem und Dezimalsystem

Zur Umwandlung vom Dezimalsystem ins Binärsystem ist folgende Vorgehensweise zu wählen:

1. $x/2 = y$ Rest z
2. Mache y zu neuem x und fahre wieder mit Schritt 1 fort, wenn dieses neue x ungleich 0 ist, ansonsten mache mit Schritt 3 weiter.
3. Die ermittelten Reste z von unten nach oben nebeneinander geschrieben ergeben dann die entsprechende Dualzahl.

Beispiel:
$30_{10} = ?_2$

Die Umwandlung vom Binärsystem ins Dezimalsystem ist wesentlich einfacher und nach obiger Übung recht einfach zu machen. Hierzu müssen einfach alle Ziffern entsprechend ihres Wertes addiert werden. Beispiel von oben:

$10011_2 = ?_{10} : 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 1 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 = 19$

binäre Verknüpfungen

Im Binärsystem gibt es drei grundlegende Verknüpfungen bzw. Rechenregeln. Diese und noch viel mehr sind in der so genannten Boolschen Algebra zu finden. Hier soll nicht auf die gesamten Regeln eingegangen werden, sonder ganz speziell auf drei binäre Verknüpfungen, die für das spätere Verständnis so mancher Konstrukte in C notwendig sind.

- **UND-Verknüpfung**
  Eine UND-Verknüpfung zweier binärer Ziffern ist immer dann erfüllt (1), wenn beide Ziffern den Zustand 1 haben:

  0 UND 0 = 0
  0 UND 1 = 0
  1 UND 0 = 0
  1 UND 1 = 1

- **ODER-Verknüpfung**
  Eine ODER-Verknüpfung zweier binärer Ziffern ist immer dann erfüllt (1), wenn eine der beiden Ziffern den Zustand 1 hat:

  0 ODER 0 = 0
  0 ODER 1 = 1
  1 ODER 0 = 1
  1 ODER 1 = 1

- **XOR-Verknüpfung**
  Eine XOR-Verknüpfung (Entweder-oder) zweier binärer Ziffern ist immer dann erfüllt (1), wenn beide Ziffern einen unterschiedlichen Zustand haben:

  0 XOR 0 = 0
  0 XOR 1 = 1
  1 XOR 0 = 1
  1 XOR 1 = 0

2.5.3. Hexadezimal-Zahlensystem

Die doch recht umfangreiche Schreibweise des Binärsystems führte zur Einführung des Hexadezimalsystems. Wie der Name schon sagt, liegt diesem Zahlensystem die Zahl 16 als Basis zugrunde. Dementsprechend würde es die Ziffern 0 bis 15 geben. Da hiermit eine eindeutige Identifikation der einzelnen Ziffern nicht mehr möglich ist, wurde vereinbart, die Ziffern 0 bis F zu verwenden. Die Buchstaben A bis F haben die Wertigkeit von 10 bis 15 im Zehnersystem.

Umwandlung der Zahlensysteme

Die Umwandlung vom Binärsystem ins Hexadezimalsystem ist sehr einfach. Die Zahlen 0 bis 15 können im Binärsystem mit jeweils 4 Ziffern dargestellt werden (0000_2 - 1111_2). Daher wird bei der Umwandlung die Dualzahl von rechts her in Vierergruppen zerteilt. Jeder Vierergruppe wird nun die jeweilige Ziffer aus dem Hexadezimalsystem zugeordnet.
Die Umwandlung vom Hexadezimalsystem ins Dezimalsystem funktioniert analog zur Umwandlung aus dem Binärsystem. Nur ist als Basis hier die 16 einzusetzen.

Beispiel:
$0100001101010111_2 = ?_{16} = ?_{10}$

2.5.4. Negation von Zahlen durch Zweierkomplement

Wir nehmen hier einmal an, dass es einen vorzeichenbehafteten Datentyp **kurz** gäbe, der durch 4 Bits dargestellt wird, wobei das 1. Bit das Vorzeichenbit ist (übliche Konvention

bei Ganzzahligen Datentypen). Für diese Datentyp wären dann folgende Bitkombinationen möglich:

0000 = 0
0001 = 1
0010 = 2
0011 = 3
0100 = 4
0101 = 5
0110 = 6
0111 = 7

1000 = -8
1001 = -7
1010 = -6
1011 = -5
1100 = -4
1101 = -3
1110 = -2
1111 = -1

In unserem Datentyp **kurz** können wir also Zahlen aus dem Wertebereich -8..7 darstellen. Hier drängt sich jetzt nur noch die Frage auf, nach welchem Prinzip die einzelnen negativen Zahlen den entsprechenden Bitkombinationen zugeordnet werden.

Regel für die Bildung des Zweierkomplements

1. Ist das 1. Bit mit 1 besetzt, so handelt es sich um einen negative Zahl.
2. Der Wert einer negativen Zahl wird dabei im Zweierkomplement dargestellt. Die bedeutet dabei, dass zunächst jedes einzelne Bit invertiert (umgedreht) wird und dann auf die so entstandene Bitkombination 1 aufaddiert wird.

Beispiel:

Zweierkomplement zu 5:		Zweierkomplement zu -5:	
Dualdarstellung von 5:	0101	Dualdarstellung von -5:	1011
Negieren von 5:	1010	Negieren von -5:	0100
+1:	0001	+1:	0001
= -5:	1011	= 5:	0101

Der Vorteil einer solchen Komplementdarstellung ist, dass eine Maschine nicht subtrahieren können muss, sondern jede Subtraktion a - b durch eine Addition a+ -b realisieren kann.

3. Aller Anfang ist schwer...

Programmieren lernen ist fast so wie eine Fremdsprache lernen. Nur durch viel Übung wird man ein Profi und man „denkt“ schon fast in der Sprache. Auch wir wollen hier klein anfangen und mit kleinen Programmen Schritt für Schritt immer umfangreichere Aufgaben erledigen.

3.1. Das erste C-Programm

Zuerst wollen wir einen Arbeitsbereich und ein Projekt anlegen. Die Source-Code Datei soll hier jetzt „main.cpp“ heißen. Im folgenden kleinen Programm sind schon die wichtigsten Merkmale eines ANSI-C Programms enthalten.

Listing 3.1: Das erste C-Programm

```
/***************************************
* Hello World - Unser erstes C-Programm *
***************************************/

#include <stdio.h>            // Header-Datei einfuegen

int main()
{
  printf("Hello World\n");    // Textausgabe auf Bildschirm
  return 0;
}
```

3.1.1. Grundstruktur

Die Grundstruktur eines Konsolenprogramms muss immer folgende Elemente enthalten:

- Vorläufig müssen alle unsere Programme die Zeichenfolge **void main()** enthalten. Es handelt sich dabei um die Startfunktion, die vom Betriebssystem beim Programmstart aufgerufen wird.
- Der eigentliche Programmteil von main() ist in geschweiften Klammern eingebettet. Dabei bezeichnet { den Anfang und } das Ende des entsprechenden Programmteils.
- Bildschirmausgaben sind mit **printf("Text")** möglich. Der Text muss dabei mit doppelten Anführungszeichen geklammert sein. Die Zeichenfolge "\n" erzeugt einen Zeilenumbruch.

- Um einzelne C-Anweisungen voneinander zu trennen, muss immer ein Semikolon als Trennzeichen angegeben werden. Vor und nach einem Semikolon können beliebig viele Leerzeichen oder -zeilen stehen.
- Die Zeile **#include <stdio.h>** sollte zunächst immer am Anfang (vor main()) stehen. Damit werden die Funktionen der Bibliothek zur Ein-/Ausgabe mit den Standardelementen Tastatur und Bildschirm zur Verfügung gestellt.

3.1.2. Kommentare

Kommentare werden mit /* und */ geklammert und können an jeder beliebigen Stelle des Programms stehen. Kommentare sind Informationen für den Programmieren und werden vom Compiler ignoriert. Um das Programm lesbar zu machen, sollte es gut aber nicht zu ausführlich kommentiert sein. Seit C++ gibt es noch den einzeiligen Kommentar mit //. Dieser gilt nur für die folgenden Zeichen in derselben Zeile. Kommentare sollten möglichst nicht geschachtelt werden, da nicht alle C-Compiler damit problemlos zurecht kommen.
Einige Tipps zur Verwendung der Kommentare:

- Kommentare sollten zusätzliche Informationen enthalten und nicht nur den Programmcode in anderen Worten wiedergeben.
- Es sollte nicht zu viel Code auf einmal kommentiert werden. 3 Seite Kommentar und dann 3 Seiten Code dient nicht unbedingt der besseren Verständlichkeit des Programms.
- Ein falscher Kommentar ist schlimmer als überhaupt keiner. Kommentare sollten bei Codeänderungen immer mit angepasst werden!
- Kurze Kommentare und keine Romane schreiben.

3.1.3. Layout

Viele Anfänger unterschätzen das Layout des Programmcodes. In Visual C++ unterstützt uns der Editor beim Einrücken von zusammengehörigen Programmblöcken. Dies ist sinnvoll, denn damit ist sofort zu erkennen, wenn verschiedenen Blöcke verschachtelt werden oder wenn eine geschweifte Klammer vergessen wurde. Jeder Block sollte mit mindestens 2 Leerzeichen eingerückt werden.
Jede Anweisung wird mit einem Semikolon abgeschlossen. Es ist dann auch üblich einen Zeilenumbruch zu machen. C erlaubt es ein ganzes Programm in einen einzelne Zeile zu schreiben. Leider ist dieses Programm dann wahrscheinlich „etwas“ schwerer verständlich als ein sauber strukturiert geschriebenes.

3.1.4. Bildschirmausgabe

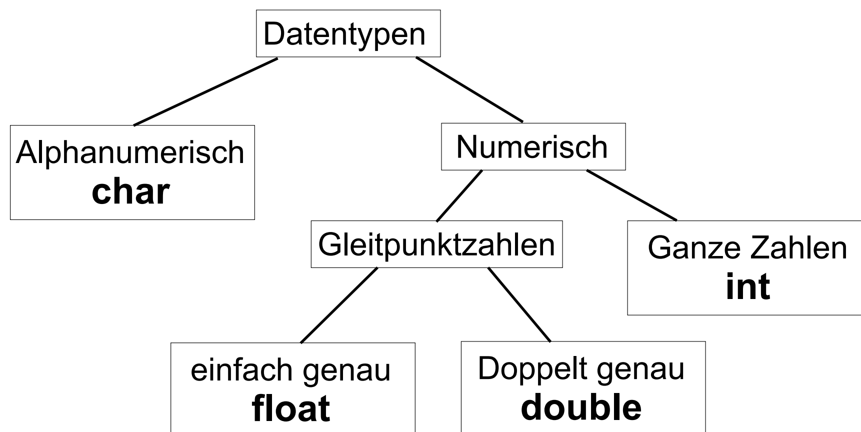
Wie schon gesehen können mit dem Befehl *printf()* Texte auf dem Bildschirm ausgegeben werden. Der Befehl *printf()* ist eine Funktion zur Bildschirmausgabe, die wir noch sehr

oft verwenden werden. Es erlaubt nicht nur die Ausgabe von Text sondern kann noch allerhand mehr. *printf()* ist kein Befehl des Sprachumfangs von C sondern eine Funktion die sich in der Bibliothek „stdio“ befindet. Somit ist es möglich, diese Bibliothek den gegebenen Hardwareanforderungen anzupassen, ohne am C-Compiler etwas zu ändern!

4. Datentypen, Variablen, Konstanten

4.1. elementare Datentypen

In C existieren vier unterschiedliche Datentypen:



Da ein Computer Zeichen wie z.B. Buchstaben anders behandelt als Gleitpunktzahlen (Bruchzahlen) wurde eine Klassifizierung notwendig. Ordnet man im Programm Daten bestimmten Klassen wie *Zeichen*, *ganze Zahl*, *Gleitpunktzahl* zu, dann teilt man dem Rechner deren **Datentyp** mit. In C existieren die 4 Grunddatentypen **char**, **int**, **float** und **double**:

- **char**
  Daten dieses Typs belegen 1 Byte Speicherplatz und repräsentieren „Zeichen". In 1 Byte kann genau 1 Zeichen des ASCII-Zeichenvorrats[1] gespeichert werden. Vor char darf eines der beiden folgenden Schlüsselwörter stehen: **signed**: 1. Bit ist Vorzeichenbit **unsigned**: 1. Bit ist kein Vorzeichenbit

- **int**
  Dieser Datentyp repräsentiert „ganze Zahlen". Der Speicherverbrauch dieses Datentyps richtet sich nach dem Betriebssystem. So werden bei 16Bit Betriebssystemen (z.B. DOS) 2Byte belegt, bei 32Bit Betriebssystemen (z.B. Windows) aber 4Byte. Entsprechend größer sind dann auch die Wertebereiche (siehe unten). Vor int können noch folgende Schlüsselwörter stehen:

[1]ASCII = American Standard for Code Information Interchange

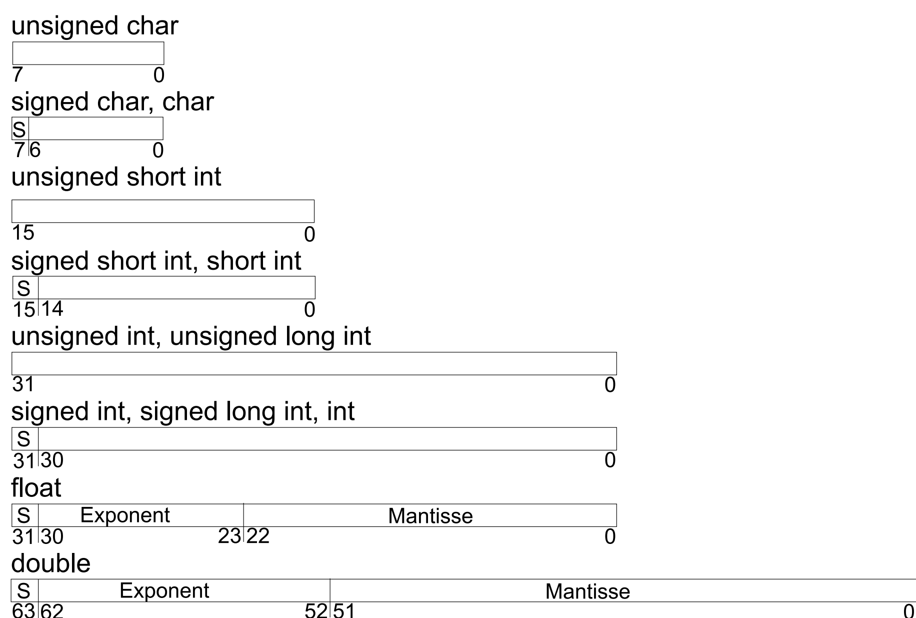
  - **short**
    Bei 32Bit Betriebssystemen bewirkt dies, dass nur 2Byte Speicherplatz belegt werden.
  - **long**
    Bei 16Bit Betriebssystemen bewirkt dies, dass 4Byte Speicherplatz belegt werden.
  - **unsigned**
    Bewirkt, dass das 1.Bit nicht als Vorzeichenbit verwendet und interpretiert wird. Als Standard wird bei int dieses immer als Vorzeichenbit ausgewertet

  Die obigen Schlüsselwörter können alleine oder auch in Kombination verwendet werden.

- **float**
  Dieser Datentyp ist für Gleitpunktzahlen mit einfacher Genauigkeit vorgesehen; dazu werden 4Byte Speicher reserviert.

- **double**
  Dieser Datentyp ist für Gleitpunktzahlen mit doppelter Genauigkeit vorgesehen; dazu werden 8Byte Speicher reserviert. Wird **long double** angegeben, so bedeutet dies meist, dass ein Speicherplatz von 80Bits reserviert wird.

Der Prozessor arbeitet mit den verschiedenen Typen im Speicher unterschiedlich. Dies führt oft bei Anfängern zu einigen vermeidbaren Fehlern bei Rechnungen, wenn verschiedenen Datentypen vermischt werden. Deshalb hier anhand einiger Skizzen den prinzipiellen Aufbau der verschiedenen elementaren Datentypen:



4.1.1. Wertebereiche

Die Aufteilung in verschiedene Datentypen bringt gewissen Probleme mit sich. Jeder Datentyp hat einen gewissen Wertebereich, der in folgender Tabelle zu sehen ist, angegeben hier für ein 32Bit-Betriebssystem.

Datentyp	Speicherbedarf	Wertebereich
unsigned char	1	0..255
char, signed char	1	-128..127
unsigned short int, unsigend short	2	0..65535
signed short int, short, short int	2	-32768..32767
int, signed, signed int, long, long int, signed long int	4	–2147483648..2147483647
unsigned int, unsigned, unsigned long, unsigned long int	4	0..4294967295
float	4	3.4E+/-38 (7 Stellen)
double	8	1.7E+/-308 (15 Stellen)
long double	10	1.2E+/-4932 (19 Stellen)

Der Speicherbedarf der Datentypen kann auch in einem C-Programm selbst ermittelt werden (siehe Listing 4.1).

Listing 4.1: Ermitteln des Speicherverbrauchs

```
/*********************************************
* Varsize - Zeigt Speichbedarf der Datentypen *
*********************************************/

#include <stdio.h>

int main()
{
  printf("Speicherbedarf der C-Datentypen\n\n");
  printf("%12s : %d\n", "char",        sizeof(char));
  printf("%12s : %d\n", "short int",   sizeof(short int));
  printf("%12s : %d\n", "int",         sizeof(int));
  printf("%12s : %d\n", "long int",    sizeof(long int));
  printf("%12s : %d\n", "float",       sizeof(float));
  printf("%12s : %d\n", "double",      sizeof(double));
  printf("%12s : %d\n", "long double", sizeof(long double));
  return 0;
}
```

Der Operator **sizeof()** gibt den verbrauchten Speicherplatz für ein Objekt in Bytes an. Er kann des öfteren auch für komplexere Datentypen verwendet werden.

Ein Überschreiten des Wertebereichs zur Laufzeit des Programms (z.B. bei Eingaben oder Berechnungen) ist möglich! Dadurch ergeben sich aber meistens unvorhersehbare Folgen im Programmablauf. Wird z.B. bei Ganzzahlen der Wertebereich überschritten, so werden die überstehenden Bits im Dualsystem einfach abgeschnitten und der Wert beginnt vorne am Wertebereich. Am folgenden Beispiel soll das kurz erläutert werden:

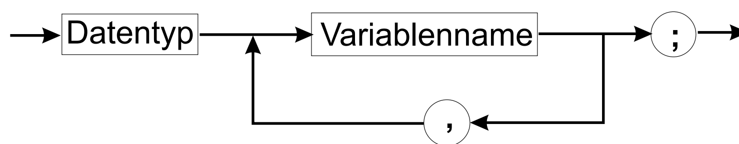
Es soll versucht werden die Zahl 50 im Datentyp **kurz** (mit 4 Bit) darzustellen:
$50_{10} = 110010_2$
Da nur für 4 Bits Platz ist, werden die ersten beiden Ziffern einfach weggeworfen, so dass schließlich folgende Bitkombination in **kurz** gespeichert wird:
$0010_2 = 2_{10}$
Man erhält also nicht die Zahl 5 sondern stattdessen die Zahl 2. Dieses Abschneiden von vorne überhängenden Ziffern bei Zahlen, die außerhalb des Wertebereichs eines Datentyps liegen, kann sogar dazu führen, dass aus positiven Zahlen dann negative Zahlen werden (siehe auch Kapitel 2.5.4):
Es soll versucht werden die Zahl 43 im Datentyp **kurz** (mit 4 Bit) darzustellen:
$43_{10} = 101011_2$
Da nur für 4 Ziffern Platz ist gehen die ersten beiden Ziffern verloren, so dass man schließlich folgende Bitkombination erhält:
$1010_2 = -5_{10}$

Natürlich werden auch bei nicht vorzeichenbehafteten Datentypen (**unsigned**) vorne über hängende Ziffern abgeschnitten. In diesem Fall kann aber niemals eine negative Zahl aus dem Abschneiden resultieren, da der Wertebereich immer als positiv betrachtet wird. Dies heißt, bei unsigned-Variablen interpretiert der Computer das erste Bit nicht als Vorzeichen.
Das Überlaufen von Datentypen wird deshalb so betont, da C beim Abspeichern von Zahlen, die außerhalb des Wertebereichs eines Datentyps liegen, kein Fehler meldet, sondern einfach die über hängenden Ziffern abschneidet. Mit diesem falschen Wert wird dann einfach weiter gearbeitet, was schließlich zu falschen Ergebnissen führt. Beim Entwurf eines Programms sollte also genau darauf geachtet werden, dass die während des Programmablaufs zu erwartenden Zahlen niemals außerhalb der Wertebereiche liegen.

Eine weitere Anmerkung noch kurz zu den Gleitkommazahlen. Oft denken die Programmierer, dass mit den Gleitkommazahlen aufgrund ihres hohen Wertebereichs alles zu meistern ist. Das ist aber ein Trugschluss. Betrachtet man die Genauigkeit z.B. des Datentyps float, so sieht man, dass dieser nur auf 7 Nachkommastellen genau rechnet. Das bedeutet aber, dass schon ab Zahlen größer als 100Millionen die letzten Stellen verloren gehen. Dasselbe gilt bei Zahlen kleiner als 0.0000001, die dann immer als Null ausgewertet werden!

4.2. Variablen

Eine Variable ist ein Datenobjekt, das einen symbolischen Namen hat, und dessen Inhalt vom Programm manipuliert werden kann. Damit der Compiler weiß, wie viel Platz für eine Variable im Speicher reserviert werden muss, muss jede Variable einen bestimmten Typ haben. Man muss dem Compiler also sagen, was in der Variablen zu speichern ist (Zeichen, ganze Zahl, Gleikommazahl usw.). Diese so genannte **Deklaration** der Variablen muss immer erfolgen, bevor mit ihnen gearbeitet wird. Der Aufbau solch eine Variablendeklaration lässt sich mit dem folgendem Syntaxdiagramm zeigen:



Hier noch einige Beispiele für solche Variablendeklarationen:

char ein_zeichen, antwort;	Für die beiden Variablen *ein_zeichen* und *antwort* wird je 1 Byte reserviert.
int zaehler;	Für die Variable *zaehler* werde 2 bzw. 4Bytes reserviert.
float einfach;	Für die Variable mit Namen *einfach* werden 4 Bytes als Gleitkommazahl reserviert.

4.2.1. Deklaration von Variablen

In C gibt es keine großen Einschränkungen für die Namensvergabe bei Variablen:

1. Es dürfen nur Buchstaben (keine Umlaute ä,ü,ö oder ß), Ziffern und der Unterstrich (_) verwendet werden.
2. Das erste Zeichen eines Variablennamens muss immer ein Buchstabe oder ein Unterstrich sein.
3. Ein Variablenname darf beliebig lang sein. Manche Compiler beachten jedoch nur die ersten 6 Zeichen.
4. Die festen Schlüsselwörter des Befehls

Wie erwähnt dürfen die festen Schlüsselwörter nicht für die Variablennamen verwendet werden. Da C nur einen sehr geringen Sprachumfang aufweist, lassen sich die Schlüsselwörter sehr schnell aufzählen:

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

Noch einige Bemerkungen zu den Variablennamen seien hier gestattet:

- **C ist case-sensitive**
  Dies bedeutet, dass der Compiler beim Übersetzen zwischen Groß- und Kleinschreibung in den Namen unterscheidet. Häufig führt dies am Anfang zu vielen Fehlermeldungen, da darauf beim Programmieren nicht geachtet wird.

- **Selbsterklärende Variablennamen**
  Bei vielen Programmierern (besonders bei Anfängern) ist zu beobachten, dass sie extrem schreibfaul sind und deshalb für die Variablen nur Namen wie „a, b1,b2,x,y,z“ verwenden. Dies führt bei einer späteren Betrachtung des Programms zu viel Verwirrung, Unübersichtlichkeit und Unverständlichkeit. Oft kann man durch selbsterklärende und aussagekräftige Variablennamen viel an Kommentaren sparen!

- **Verbesserung der Lesbarkeit**
  Durch Groß- und Kleinschreibung oder Unterstriche im Variablennamen bei zusammengesetzten Begriffen kann die Lesbarkeit des Programms weiter erhöht werden. So ist z.B. der Name *hopfenundmalzverloren* schlechter lesbar als *Hopfen_und_Malz_verloren*.

4.2.2. Wertzuweisung

Variablen nur anzulegen und zu deklarieren macht keinen Sinn. Wir wollen natürlich mit ihnen in den Programmen arbeiten. Dazu müssen den Variablen verschiedene Werte zugewiesen werden. Dies geschieht durch den so genannten Zuweisungsoperator „=“. Ein kleines Beispiel:

Durch die Anweisung

zahl = 5;

wird der Variablen *zahl* der Wert *5* zugewiesen. Sie hat nach dieser Anweisung den

Inhalt „5“. Diese Zuweisung von Werten lässt sich beliebig oft im Programm mit derselben Variablen wiederholen.

Durch die Anweisung

zahl1 = zahl2;

wird der Variablen *zahl1* der Wert der Variablen *zahl2* zugewiesen. Sie haben damit nach der Anweisung den gleichen Wert.

Zu beachten ist, dass die Werte, welche einer Variablen zugewiesen werden, vom gleichen Typ sein sollten. Ist dies nicht der Fall gibt der Compiler beim Übersetzen zumeist eine Warnung oder einen Fehler aus (siehe auch Kapitel 4.2.3).

Eine Wertzuweisung zu einer Variablen empfiehlt sich bereits bei der Deklaration. Dies nennt man dann **Initialisierung**. Damit ist ein definierter Anfangswert in der Variablen gespeichert und nicht irgendein Zufallswert. Eine „gute“ Deklaration der Variablen lautet demnach z.B.

int zahl = 0;

Damit wird der Speicher für die Variable belegt und sofort auf Null gesetzt. Somit hat die Variable *zahl* den Anfangswert 0.

4.2.3. Datentypumwandlung

implizite Datentypumwandlung

Wenn in Ausdrücken Operanden (Variablen, Konstanten) unterschiedlicher Datentypen verknüpft werden, dann werden so genannte *implizite Datentypumwandlungen* notwendig. Hierbei hält sich C an die folgenden Regeln:

Regel 1 (char, short, int und unsigned)
Es wird mit nichts kleinerem als **int** gerechnet, d.h. char und short werden implizit und ohne Einwirkung des Programmierers nach int umgewandelt. Sollte der int-Datentyp nicht ausreichen, um eine entsprechende Konstante aufzunehmen, wird mit **unsigned int** gerechnet.

Regel 2 (signed und unsigned)
Hier werden mehrere Fälle unterschieden:

1. **unsigned** nach längerem signed oder unsigned Datentyp:
   Der ursprüngliche Wert bleibt unverändert

2. **signed** nach gleichlangem oder längerem unsigned-Datentyp:
   Es wird das entsprechende Bitmuster im unsigned-Datentyp abgelegt, wobei das Vorzeichenbit erhalten bleibt, es also zu falschen Werten kommen kann.

3. **signed** oder **unsigned** nach einem kürzeren signed oder unsigned-Datentyp:
   Für den Fall dass der Wert zu groß ist gibt ANSI-C keine festen Vorschriften. Es ist aber meist der Fall, dass die vorne überhängenden Bits abgeschnitten werden.

4. **unsigned** nach gleichlangem signed-Datentyp:
   In diesem Fall wird meist einfach das Bitmuster im signed-Datentyp abgespeichert. Es kann also auch hier zu Wertverfälschungen kommen, wenn das Vorzeichenbit durch einen zu großen Wert belegt wird.

Regel 3 (Gleitpunktzahlen und Ganzzahlen)
Hier werden zwei Fälle unterschieden:

1. Gleitpunktzahl in Ganzzahl
   Der gebrochene Anteil der Gleitpunktzahl wird abgeschnitten. Diese Regel kann z.B. zum Runden von Gleitpunktzahlen verwendet werden. Wenn der ganzzahlige Anteil außerhalb des Wertebereichs des Ganzzahltyps liegt, so ist das Verhalten undefiniert.

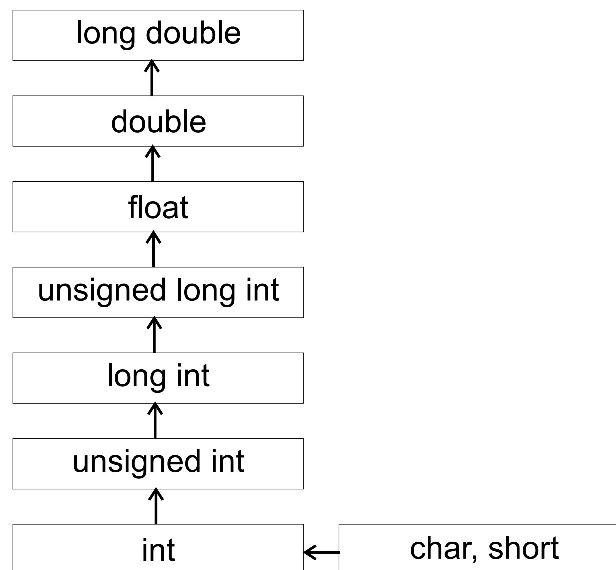
2. Ganzzahl in Gleitpunktzahl
   Wenn der Wert zwar im Wertebereich der darstellbaren Gleitpunktzahlen liegt, aber nicht genau darstellbar ist, dann ist das Ergebnis entweder der nächsthöhere oder der nächst niedrigere darstellbare Wert (ist in ANSI-C nicht vorgeschrieben)

Regel 4 (float, double und long double)
Hier werden zwei Fälle unterschieden:

1. **float** nach double
   float nach long double
   double nach long double
   Hier bleiben die ursprünglichen Werte unverändert.

2. **double** nach float
   long double nach float
   long double nach double
   Falls der Wert größer ist als der Zieldatentyp aufnehmen kann, liegt undefiniertes Verhalten vor. Wenn der Wert zwar im Wertebereich des Zieldatentyps liegt, aber nicht genau darstellbar ist, dann ist das Ergebnis entweder der nächsthöhere oder der nächst niedrigere darstellbare Wert (ist in ANSI-C nicht vorgeschrieben)

Regel 5 (übliche arithmetische Umwandlungen)
Die hier angegebenen impliziten Umwandlungen von Datentypen werden als übliche arithmetische Umwandlungen bezeichnet:



Horizontale Umwandlungen werden immer durchgeführt, vertikale nur bei Bedarf. Für die vertikalen Umwandlungen gilt, dass wenn bei einer Berechnung einer der Operanden einen Datentyp besitzt, der in der Grafik weiter oben steht, dann wird zuerst der andere Operand in diesen „höher stehenden“ Datentyp umgewandelt, bevor der Ausdruck berechnet wird.

explizite Datentypumwandlung

Ist eine Typumwandlung erwünscht (z.B. bei der Zuweisung einer ganzen Zahl zu einer Gleitpunktzahl) so muss über das so genannte **Typecasting** dem Compiler dies mitgeteilt werden. Die explizite Umwandlung ist immer dort notwendig, wo der Compiler die Umwandlung nicht selbständig durchführen kann oder darf.
Ein Typecast in ein in runden Klammern gesetzter Datentyp, der für den Ausdruck direkt rechts davon verwendet wird, d.h. das Ergebnis des Ausdruckes der rechts vom Typecast zugehörig steht, wird in den angegebenen Datentyp umgewandelt.
Beispiel:

```
int ganz1, ganz2;
double gleitpkt;

gleitpkt = (double)ganz1 + ganz2;
```

Ein andere Möglichkeit bietet sich bei Berechnungen. Hier kann durch Wahl der richtigen Konstanten schon die implizite Datentypumwandlung so beeinflusst werden, dass danach das richtige Ergebnis berechnet wird:

```
double gleitpkt;

gleitpkt = 1.0/2;
```

4.2.4. Bildschirmausgabe

Die Ausgabe eines Textes mit der Funktion *printf* haben wir in unserem Einführungsbeispiel schon kennen gelernt. Aber wie schon erwähnt ist diese Funktion um einiges mächtiger. Sie erlaubt z. B. auch die Ausgabe von Variablenwerten auf dem Bildschirm. Aber hier wird es schon etwas komplizierter.
Zuerst die Funktionsdeklaration:

int printf(const char *format [, argument]...);

Zur korrekten Ausgabe benötigt die Funktion den exakten Datentyp den sie ausgeben soll. Des weiteren müssen so genannte Platzhalter im Ausgabetext vorhanden sein, die dann durch die Variablenwerte ersetzt werden. Doch nun Schritt für Schritt.

Wir wissen schon wie man Text ausgibt: **printf(”Text”);**
Daran wird sich auch nichts ändern. Nur kann der Text jetzt außer den Buchstaben noch Kontrollzeichen enthalten, welche die Bildschirmausgabe beeinflussen. Alle Kontrollzeichen werden durch einen Backslash eingeleitet:

\a	Klingelton
\b	Backspace (ein Zeichen zurück)
\f	Seitenvorschub
\n	neue Zeile (!!)
\r	Wagenrücklauf (an den Anfang der aktuellen Zeile)
\t	Tabulator (!!)
\v	vertikaler Tabulator
\0oo	Zeichen das der Oktalzahl oo entspricht
\xhh	Zeichen das der Hexadezimalzahl hh entspricht
\'	Hochkomma
\"	Anführungszeichen (!!)
\\	Backslash (!!)

Die Anweisung
printf("Hallo \n\"Egon\" ");
würde zum Beispiel auf dem Bildschirm:

```
Hallo
"Egon"
```

ausgeben.

Das waren die Steuerzeichen, von denen jedoch meistens nur sehr wenige gebraucht werden (siehe Ausrufungszeichen). Kommen wir nun zur Ausgabe der Variablenwerte. Wie schon erwähnt benötigt printf dazu einige Angaben. Für jeden auszugebenden Wert muss im Ausgabetext ein **Platzhalter** mit folgendem Format eingefügt werden:

%FWGLU

% ist dabei das Kennzeichen für einen Platzhalter.

F [Formatierungszeichen] der Wertausgabe (siehe unten)

W [Weite]. Entspricht der Mindestanzahl der auszugebenden Zeichen.

G [Genauigkeit] der Nachkommastellen bei Gleitkommazahlen. Wird als . oder .*Ganzzahl* angegeben.

U Umwandlungszeichen für den Datentyp (siehe unten). Muss immer angegeben werden. Die anderen Parameter sind optional.

Für das **Formatierungszeichen** sind folgende Angaben möglich:

-	linksbündige Justierung
+	Ausgabe des Vorzeichens mit + oder -
␣	Falls das 1. Zeichen kein Vorzeichen ist, wird ein Leerzeichen ausgegeben
0	Es werden führende Nullen verwendet, wenn für die *Weite* ein Wert angegeben wurde und die Zahl zu kurz ist.
#	abhängig von dem verwendeten Umwandlungszeichen. Führt bei der Ausgabe von Hexadezimalen- oder Oktalzahlen zur Ausgabe einer führenden 0 oder 0x. Bei Gleitkommazahlen erzwingt es die Ausgabe des Dezimalpunktes auch wenn keine Nachkommastellen existieren.

Nun noch der komplizierteste Teil, die **Umwandlungszeichen**. Dieses Zeichen legt fest, um welchen Datentyp es sich handelt, der hier ausgegeben werden soll. Ist hier das falsche Umwandlungszeichen angegeben, so wundert sich der Anwender später über die seltsamen Ausgaben seines Programms. Folgende Umwandlungszeichen stehen zur Verfügung:

d,i	vorzeichenbehaftete Dezimalzahl (z.B. int)
o	vorzeichenlose ganze Oktalzahl (z.B. unsigned int)
u	vorzeichenlose ganze Dezimalzahl (z.B. unsigned int)
x,X	vorzeichenlose ganze Hexadezimalzahl (z.B. unsigned int). x = Kleinbuchstaben für ab,b,c,d,e,f. Bei X werden Großbuchstaben verwendet
f	Gleitkommazahl (float). Die Ausgabe ist ddd.dddddd mit 6 Nachkommastellen falls nichts anderes über die Genauigkeitsangabe erfolgt ist.
e,E	Gleitkommazahl in Exponentialschreibweise. Ausgabe erfolgt mit d.ddde±dd bzw. d.dddE±dd
c	Ausgabe des Wertes als Zeichen (z.B. char)
s	Ausgabe als Zeichenkette (z.B. char[])
p	Ausgabe der Adresse eines Zeigers
%	Ausgabe des Prozentzeichens

Zusätzlich gibt es noch eine **Längenangabe** für folgende Umwandlungszeichen:

> h Bei d,i,o,u,x,X wird das entsprechende Argument als *short* behandelt, also z.B. als short int.
>
> l Bei d,i,o,u,x,X,e,E,f wird das Argument als *long* behandelt, also z.B. als long int oder double.
>
> L Für e,E,f wird das Argument als *long double* behandelt.
>
> Bei allen anderen Umwandlungszeichen sollten die Längenangaben nicht verwendet werden. Es sollte jedoch darauf geachtet werden, speziell bei Variablen vom Typ short, long und long double diese Zeichen zu verwenden, da sonst eventuell falsche Ausgaben entstehen. Diese haben nichts mit einem falschen Inhalt der Variablen zu tun, sonder resultieren aus einer falschen Ausgabe durch printf().

Wie können nun die Variablen ausgegeben werden? Das ist jetzt nur noch eine Frage der Zusammenstellung obiger Zeichen.
Das folgende Programm demonstriert die Wirkungsweise der Umwandlungszeichen:

Listing 4.2: Verwendung der Umwandlungszeichen in printf()

```
#include <stdio.h>

int  main()
{
   int    ganz1 = 125,
          ganz2 = -19893;
   float  gleit1 = 1.23456789,
          gleit2 = 2.3e-5;

   printf("Demonstration zu den %s\n", "Umwandlungszeichen");
   printf("==========================================\n\n");

   printf("(1) dezimal:            ganz1=%d, ganz2=%i\n", ganz1, ganz2);
   printf("(2) oktal:              ganz1=%o, ganz2=%o\n", ganz1, ganz2);
   printf("(3) hexadezimal:        ganz1=%x, ganz2=%X\n", ganz1, ganz2);
   printf("(4) als unsigned-Wert:  ganz1=%u, ganz2=%u\n", ganz1, ganz2);
   printf("(5) als char-Zeichen:   ganz1=%c, ganz2=%c\n\n", ganz1, ganz2);

   printf("(6) f:    gleit1=%f, gleit2=%f\n", gleit1, gleit2);
   printf("(7) e,E:  gleit1=%e, gleit2=%E\n", gleit1, gleit2);
   printf("(8) g,G:  gleit1=%g, gleit2=%G\n\n", gleit1, gleit2);

   printf("(9) Adresse von ganz1=%p, Adresse von gleit2=%p\n\n", &ganz1, &gleit2);

   printf("(10) Das Prozentzeichen %%%n\n", &ganz2);
   printf("(11) ganz2 = %d\n", ganz2);
   return(0);
}
```

Das folgende Programm demonstriert die Wirkungsweise der Formatierungszeichen und der Weite:

Listing 4.3: Verwendung der Formatierungszeichen in printf()

```
#include <stdio.h>

int  main()
{
   int    ganz1 = 125,
          ganz2 = -19893,
          ganz3 = 20;
   float  gleit1 = 1.23456789,
          gleit2 = 2.3e-5;

   printf("Demonstration␣zu␣den␣%s\n", "Formatierungszeichen␣und␣Weite");
   printf("==================================================\n\n");

   printf("(1)␣␣|%20d|␣␣␣␣|%-+20d|\n", ganz1, ganz2);
   printf("(2)␣␣|%020o|␣␣␣␣|%-020o|\n", ganz1, ganz2);
   printf("(3)␣␣|%#20x|␣␣␣␣|%#20X|\n", ganz1, ganz2);
   printf("(4)␣␣|%+20i|␣␣␣␣|%20u|\n", ganz1, ganz2);
   printf("(5)␣␣|%#-*x|␣␣␣␣|%+*u|\n\n", ganz3, ganz1, 20, ganz2);

   printf("(6)␣␣|%-20f|␣␣␣␣|%20f|\n", gleit1, gleit2);
   printf("(7)␣␣|%+-20f|␣␣␣␣|%020f|\n", gleit1, gleit2);
   printf("(8)␣␣|%+#20g|␣␣␣␣|%-#20g|\n", gleit1, gleit2);
   printf("(9)␣␣|%+#20f|␣␣␣␣|%-#20f|\n", gleit1, gleit2);
   printf("(10)␣|%+#*e|␣␣␣␣|%-#*E|\n", ganz3, gleit1, 20, gleit2);
   return(0);
}
```

Das folgende Programm demonstriert die Wirkungsweise der Längenangabe und der Genauigkeit:

Listing 4.4: Verwendung der Längenangabe in printf()

```
#include <stdio.h>

int  main()
{
    float        einf_1, einf_2;
    double       dopp_1, dopp_2;
    long double  ddopp;

    einf_1 =10.0/3.0; /* Divsion */
    dopp_1 =10.0/3.0;
    dopp_2 =-10.0/3.0;

    printf("|%10.3f|\n", einf_1);
    printf("|%10.3e|\n", einf_1);
    printf("|%-10.3g|\n", einf_1);
    printf("|%30.9f|\n", dopp_1);
    printf("|%30.9E|\n", dopp_2);
    printf("|%-030.9f|\n\n", einf_1);

    einf_1=1e5/3.0;
    dopp_2=1e8/3.0;
    printf("|%30.2f|\n", einf_1);
    printf("|%30.2e|\n", einf_1);
    printf("|%30.2G|\n", einf_1);
    printf("|%30.8f|\n", dopp_2);
    printf("|%030.8f|\n\n", einf_1);

    ddopp = dopp_1 = 0.000000000730;
    printf("|%g|\n", dopp_1);
    printf("|%6.4f|\n", dopp_1);
```

```
    printf("|%6.89Lf|\n", ddopp);
    return(0);
}
```

Etwas einfacher aber lange nicht so komfortabel ist die einfache Ausgabe eines einzelnen Zeichens (z.B. char) auf dem Bildschirm. Dazu gibt es in der Bibliothek „stdio“ die Funktion **putchar**.

int putchar(int Zeichen);

putchar(antwort);
gibt z. B. den Inhalt der Variablen *antwort* aus. Diese muss aber den Code eines Zeichens aus dem ASCII-Zeichensatz haben, denn alle anderen Zeichen kann diese Funktion nicht ausgeben. Es ist auch nicht möglich einen Text damit komplett auszugeben. Dieser müsste dann Zeichen für Zeichen auf dem Bildschirm ausgegeben werden. Alle oben genannten *Kontrollzeichen* können jedoch einzeln auch mit dieser Funktion ausgegeben werden:
putchar('\n');
beginnt z.B. eine neue Zeile.

4.2.5. Tastatureingabe

Die Ausgabe auf dem Bildschirm über printf wurde im vorigen Abschnitt behandelt. Nun gibt es auch eine analoge Funktion zur Eingabe über Tastatur. Diese Funktion heißt **scanf**.

int scanf(const char *format [,argument]...);

Der Aufbau zum Einlesen des Wertes einer Variablen ist nahezu identisch zur Ausgabe mit printf(). Wieder muss in einem Text für jeden einzulesenden Wert ein Platzhalter platziert werden.
Auch bei *scanf()* können die Kontrollzeichen verwendet werden, jedoch ist zu empfehlen, diese wegzulassen. Damit bleiben nur noch die **Platzhalter**, die ein etwas anderes Format aufweisen:

%WLU

% ist auch hier ein Kennzeichen für einen Platzhalter

W [Weite] maximale Anzahl einzulesender Zeichen. Wird vor dieser Anzahl ein ungültiges Zeichen eingelesen, so wird die Eingabe für die Variable abgebrochen.

L [Längenangabe] für die Variable. h für *short* und l,L für *long*

U Umwandlungszeichen (siehe unten). Muss immer angegeben werden. Die anderen Parameter sind optional.

Die möglichen **Umwandlungszeichen** entsprechen denen von *printf()*. Hier noch einmal eine Aufzählung:

d,i	vorzeichenbehaftete Dezimalzahl (z.B. int)
o	vorzeichenlose ganze Oktalzahl (z.B. unsigned int)
u	vorzeichenlose ganze Dezimalzahl (z.B. unsigned int)
x,X	vorzeichenlose ganze Hexadezimalzahl (z.B. unsigned int)
f	Gleitkommazahl (float)
e,E	Gleitkommazahl in Exponentialschreibweise
c	Eingabe eines Zeichen (z.B. char)
s	Eingabe einer Zeichenkette (z.B. char[])
p	Eingabe der Adresse eines Zeigers
%	Eingabe des Prozentzeichens
[liste]	Eingabe einer Zeichenkette solange bis ein Zeichen auftaucht, das **nicht** in *liste* vorkommt.
[∧liste]	Eingabe einer Zeichenkette solange bis ein Zeichen auftaucht, das in *liste* vorkommt.

Bei *scanf()* muss immer die Adresse der Variablen angegeben werden, in die der eingegebene Wert geschrieben wird. Einer der häufigsten Fehler ist es, den Adressoperator **&** zu vergessen. Darauf folgt meist ein Programmabsturz nach der Eingabe!

Hier nun ein kleines Beispiel zur Eingabe von Werten in verschiedene Variablen:

Listing 4.5: Eingabe von Variablen mit scanf()

```
#include <stdio.h>

int  main()
{
    int    zahl1, zahl2;

    printf("Gib zwei ganze Zahlen mit Komma getrennt ein: ");
    scanf("%d, %d", &zahl1, &zahl2);

    printf("Die beiden Zahlen waren: %d und %d\n", zahl1, zahl2);
    return(0);
}
```

Ein etwas komplexeres Programm zur Demonstration, zu was scanf() alles fähig ist:

Listing 4.6: komplexe Eingabe von Variablen mit scanf()

```
#include <stdio.h>

int  main()
{
    int   ganz,
          hexa1, hexa2,
          start1, ende1, start2, ende2;
    float gleit;

    printf("Gib eine beliebige Zeile von Text ein, wobei in diesem\n");
    printf("irgendwo zunaechst eine ganze Zahl und dann spaeter eine\n");
    printf("Gleitpunktzahl steht:\n");
    scanf("%*[^0123456789]%d%*[^0123456789.]%f%*[^\n]", &ganz, &gleit);
    fflush(stdin);   /* Eingabepuffer leeren */
    printf("Die beiden versteckten Zahlen waren: %d, %g\n\n", ganz, gleit);

    printf("Gib einen beliebigen Text aus Buchstaben ein, in dem\n");
    printf("2 hexadezimale Zahlen versteckt sind.\n");
    scanf("%*[ghijklmnopqrstuvwxyzGHIJKLMNOPQRSTUVWXYZ]%n%x%n"
          "%*[ghijklmnopqrstuvwxyzGHIJKLMNOPQRSTUVWXYZ]%n%x%n",
          &start1, &hexa1, &ende1, &start2, &hexa2, &ende2);
    printf("%*s%.*s", start1, " ", ende1-start1, "------------------------");
    printf("%*s%.*s", start2-ende1, " ", ende2-start2, "------------------------");
    printf("\nDie beiden Hexazahlen: %x und %x ", hexa1, hexa2);
    printf("sind dezimal: %d und %d\n", hexa1, hexa2);
    return(0);
}
```

Eine Eingabe mit *scanf* wird immer mit ENTER abgeschlossen. Soll darauf sofort eine neue Eingabe erfolgen, so muss zuvor der Eingabepuffer gelöscht werden, da das ENTER noch immer dort herumschwirrt. Zu erreichen ist dies durch Aufruf der Funktion

```
int fflush( FILE *stream );
```

mit dem Parameter **stdin**:
fflush(stdin);

Soll nur ein Zeichen eingelesen werden, kann auch die Funktion **getchar** verwendet werden.

```
int getchar( void );
```

Analog zu putchar() können hier aber nur einzelne Zeichen und nie ganze Zahlen eingelesen werden. Wie bei scanf() werden die Zeichen erst nach Drücken der ENTER-Taste eingelesen. Es muss also nach jedem Zeichen erst ENTER gedrückt werden bevor die Funktion getchar() beendet wird. Ein erneuter Aufruf von getchar() liest nun analog zu scanf() das ENTER-Zeichen aus dem Puffer. Dieser muss also auch hier zuerst gelöscht werden.

Listing 4.7: Ein-/Ausgabe eines Zeichens mit getchar() und putchar()

```
#include <stdio.h>

int  main()
{
   char zeich;

   printf("Gib ein Zeichen ein: ");
   zeich = getchar();
   putchar(zeich);
   return(0);
}
```

Eine andere Möglichkeit, einzelne Zeichen ohne Pufferung direkt einzulesen bietet die Funktion getch():

```
int getch( void );
```

Diese Funktion ist nicht im ANSI-C Standard vorgesehen, wird aber von den meisten C-Compilern angeboten. Sie wartet nicht auf einen Abschluss der Eingabe über ENTER, sondern liest das Zeichen direkt nach dem Tastendruck ein über liefert den Wert zurück.

Listing 4.8: Eingabe eines Zeichens mit getch()

```
#include <stdio.h>

int  main()
{
   char zeich1, zeich2;

   printf("Gib 1. Zeichen ein: ");
   zeich1=getch();
   printf("\nGib 2. Zeichen ein: ");
   zeich2=getch();

   printf("\n2.Zeichen war %c und 1.Zeichen war %c\n", zeich2, zeich1);
   return(0);
}
```

4.3. Konstanten

Im Gegenteil zu Variablen können Konstanten während des Programmablaufs nicht verändert werden. Ihr Wert wird schon bei der Programmerstellung festgelegt. Der Datentyp ist durch den Aufbau der Konstanten festgelegt. Eine Konstante die wir schon kennen, ist die Textkonstante in Anführungszeichen.

4.3.1. Zeichenkonstanten

Es gibt in C zwei Zeichenkonstanten mit unterschiedlicher Schreibweise:

- **char-Konstanten**
  char-Konstanten sind 1Byte lang und werden in einfache Hochkommas gesetzt. Der Wert dieser Konstanten entspricht dem ASCII-Wert des Zeichen. Hier einige Beispiele:

'a' = 97, 'W' = 87, '*' = 42, '8' = 56
Char-Konstanten können nicht nur einer char-Variablen sondern auch einer ganzzahligen Variablen zugewiesen werden. Es wird dann der zugehörige ASCII-Wert gespeichert.

- **Zeichenketten**
  Zeichenketten sind Konstanten die aus mehreren Zeichen bestehen. Wir haben sie schon als Text bei der Bildschirmausgabe kennen gelernt. Sie werden in Anführungszeichen geklammert.
  Beispiele: „Text“, „Hallo“, „Das ist eine Zeichenkette“

4.3.2. Ganzzahlige Konstanten

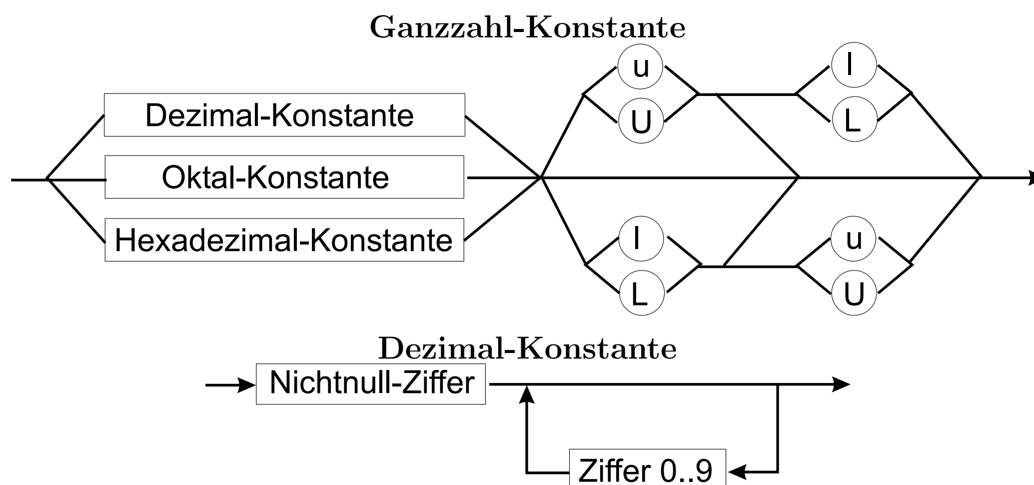
Bei den ganzzahligen Konstanten gibt es mehrere Schreibweisen:

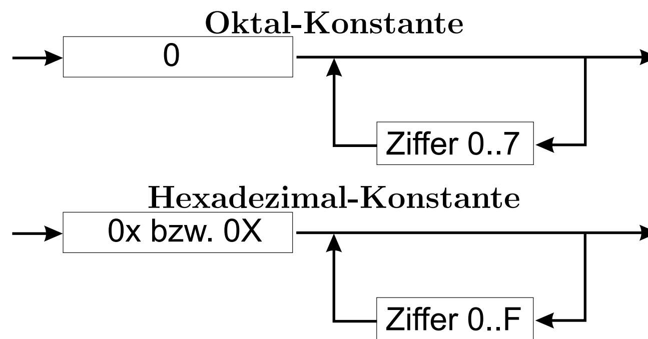
- **Dezimal-Konstanten**
  Das sind Zahlen, die mit einer von 0 verschiedenen Ziffer beginnen, also alle ganzen Zahlen des Dezimalsystems.

- **Oktal-Konstanten**
  Dies sind Zahlen des Oktalsystems, die mit 0 beginnen, also z.B. 045, 07, 01234 usw.

- **Hexadezimal-Konstanten** Diese Konstanten beginnen immer mit 0x oder 0X und beinhalten alle Zahlen des Hexadezimalsystems wie z.B. 0xF1, 0x124, 0xAB

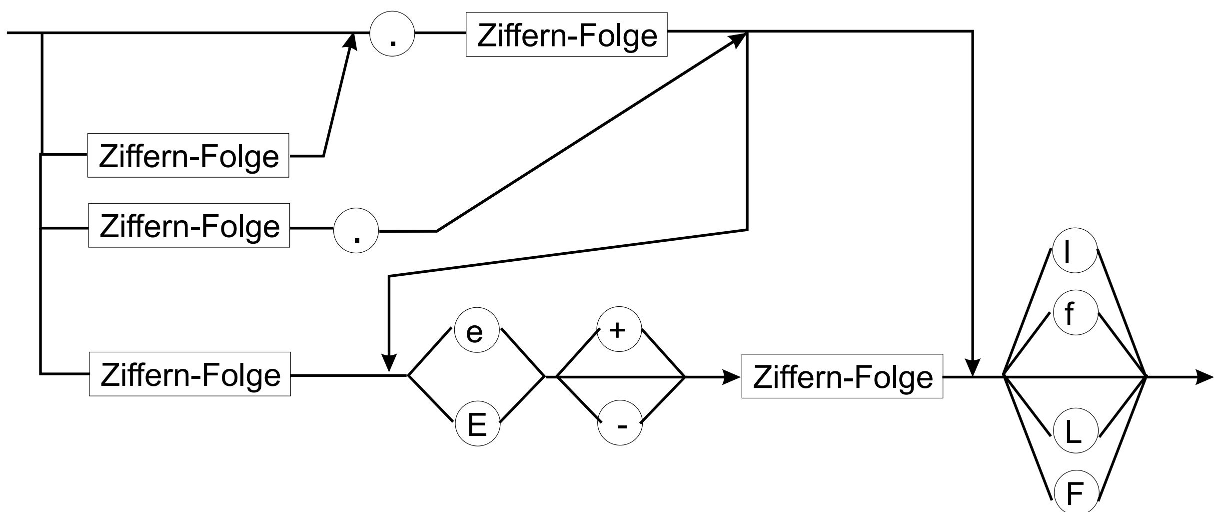
Zusätzlich kann am Ende der Konstanten noch angegeben werden, ob es sich um eine *long*(l,L), *unsigend* (u,U) oder *unsigned long* (ul,UL,uL,Ul) Konstante handelt.
Die folgenden Syntaxdiagramme zeigen die Zusammensetzung noch einmal auf:





4.3.3. Gleitpunkt Konstanten

Gleitpunkt-Konstanten können auf viele Arten angegeben werden, wie im folgenden Syntaxdiagramm zu erkennen ist. Die wohl am häufigste Form ist die mit Dezimalpunkt, also z.B. **12.5** oder **0.56**. Wichtig ist, dass ein **Punkt** als Dezimalzeichen verwendet wird. Dies gilt auch bei einer Eingabe über Tastatur.



Am Ende der Gleitpunktkonstante kann einer der Buchstaben l,L,f,F angegeben werden. Entsprechend des Buchstabens wird die Zahl interpretiert als:

- float bei f oder F
- long double bei l oder L
- als double ohne Angabe eines Buchstabens

4.3.4. symbolische Konstanten

C sieht die Möglichkeit vor, an Konstanten Namen zu vergeben. Das bringt Vorteile in Bezug auf Lesbarkeit der Programme und Arbeitsersparnis beim Programmieren sowie leichte Änderbarkeit der Programme. In ANSI-C gibt es zwei verschiedene Arten symbolische Namen an Konstanten zu vergeben:

- Definition mit **#define**
  Mit der #define Anweisung am Anfang eines Programmes können Texte und Zahlen einem Namen zugeordnet werden, z.B.
  #define PI 3.14156
  Überall dort wo jetzt der Zahlenwert 3.14156 im Programm notwendig wird, kann auch PI geschrieben werden. Die #define Anweisung wird nicht mit einem Semikolon abgeschlossen und die Raute muss immer in der ersten Spalte der Zeile stehen. Genau genommen handelt es sich hier um eine Preprozessor-Direktive. Für die Namen der Konstanten gelten die gleichen Regeln wie für Variablennamen.

- Definition mit **const**
  Die Definition der Konstanten erfolgt dabei analog zur Deklaration von Variablen mit gleichzeitiger Initialisierung:
  const double PI = 3.14156;
  Die Variable PI darf jetzt im Programm nicht mehr verändert werden.

Der Befehl *const* hat gegenüber dem *#define* den Vorteil, dass eine Berechnung nur einmal bei der Definition stattfindet. Bei #define wird der Ausdruck überall im Programm ersetzt und findet daher bei jeder Verwendung statt. Das erhöht zudem die Programmgröße. Der Nachteil von *const* ist der Verbrauch von Arbeitsspeicher, der z.B. bei Microcontrollern immer etwas knapp ist.

4.3.5. Aufzählungstypen

Häufig werden ganze Zahlen verwendet, um Begriff zu codieren, z.B. für Bewertungen (1: sehr gut, 2:gut, usw) oder Monatsnamen (1: Januar, 2: Februar, usw). Die Aufzählungstypen bieten eine einfache Möglichkeit, im Programm anstelle der Zahlencodes aussagekräftige Namen zu verwenden.

Allgemeine Syntax:
enum {enumerator-list}
enum Name {enumerator-list}

Die folgenden Beispiele verdeutlichen die verschiedenen Verwendungsmöglichkeiten des **enum**-Typs:

Beispiel 1:

```
enum {rot, gelb, blau};

hat dieselbe Wirkung wie:
const int rot  = 0;
const int gelb = 1;
const int blau = 2;
```

Beispiel 2:

```
enum {rot=1, gelb, blau};

hat dieselbe Wirkung wie:
const int rot  = 1;
const int gelb = 2;
const int blau = 3;
```

Beispiel 3:

```
enum {rot=12, gelb, blau=2};

hat dieselbe Wirkung wie:
const int rot  = 12;
const int gelb = 13;
const int blau = 2;
```

Mit dem enum-Datentyp können also fortlaufende **int**-Konstanten auf einfache Art und Weise erzeugt werden.

Folgende Regeln sollten beachtet werden:

- Wird kein Startwert vorgegeben, so beginnt die Nummerierung immer bei 0
- Es können den einzelnen Konstanten beliebige Werte vorgegeben werden. Allerdings beeinflusst dies auch die Zählung für alle nachfolgenden Konstanten.
- Es ist nicht vorgeschrieben, dass alle Werte in der enum-Liste verschieden sein müssen. Es können also auch gleiche Werte vorgegeben werden. Dies sollte allerdings vermieden werden.
- Die Namen für die einzelnen Konstanten dürfen im gesamten Gültigkeitsbereich des enum-Typs nur einmal vorkommen. Doppelte Namen sind also nicht erlaubt.
- Die enum-Wertenamen dürfen nicht mehr für Variablennamen verwendet werden.

Oft sieht man auch folgende Definition des Datentyps **bool**, der ja in C nicht extistiert:

typedef enum FALSE, TRUE bool;

Mit typedef wird ein neuer Typ erstellt, der vom Typ enum ist und bool heißt. Variablen vom Typ bool können dann in Verzweigungen, Zuweisungen und sonstigen Ausdrücken verwendet werden, in denen nur die Zustände FALSE (nicht erfüllt) oder TRUE (erfüllt) benötigt werden.

Kleines Beispiel zur Verwendung des enum-Datentyps:

Listing 4.9: Verwendung enum

```
#include <stdio.h>

enum Skatkartenwert {Ass=11, Zehn=10, Koenig=4, Dame=3, Bube=2};

int main()
{
  enum Skatkartenwert Karte;

  printf("Welchen Wert hat ihre Karte:");
  scanf("%d",&Karte);

  switch(Karte)
  {
  case Ass: printf("Sie haben ein Ass im Aermel!\n");
            break;
  case Koenig: printf("Sie haben einen Koenig!\n");
            break;
  case Dame: printf("Sie haben eine Dame!\n");
            break;
  case Bube: printf("Sie haben einen Buben!\n");
            break;
  default: printf("Diesen Kartenwert gibt es nicht!");
  }

  return(0);
}
```

5. Ausdrücke und Operatoren

5.1. Ausdruck und Anweisung

Die aus Operanden und Operatoren zusammengesetzte rechte Seite bezeichnet man als *Ausdruck*. Ein Ausdruck liegt auch bei

```
variable = ausdruck
```

vor. Wird nach einem solchen Ausdruck noch ein Semikolon angegeben:

```
variable = ausdruck;
```

spricht man von einer *Anweisung*.

5.2. Zuweisungsoperator

Den einfachen Zuweisungsoperator "=" haben wir schon im Kapitel Variablen kennen gelernt. Er weist den rechten Ausdruck, der wiederum aus z.B. arithmetischen Operatoren bestehen kann, der linken Hälfte (meist einer Variablen) zu:

```
a = 1 + 5;
text = "Hallo Welt";
zahl1 = zahl2;
```

5.3. Arithmetische Operatoren

Arithmetische Operatoren werden in die zwei Klassen der *unären* und der *binären* Operatoren eingeteilt. Unäre Operatoren wirken nur auf einen Operanden, ein Beispiel dafür ist das negative Vorzeichen. Binäre Operatoren verknüpfen zwei Operanden, ein typischer Vertreter dafür wäre der Operator zur Addition. Wo nicht speziell darauf verwiesen wird, können die arithmetischen Operatoren auf alle Ganzzahl-Datentypen und auch auf die Gleitkomma-Datentypen angewandt werden.

5.3.1. binäre arithmetische Operatoren

Die folgenden binären arithmetischen Operatoren sind in C definiert:

Operator	Bedeutung
+	addiert zwei Werte, z.B. a + b
-	subtrahiert zwei Werte, z.B. a - b
*	multipliziert zwei Werte, z.B. a * b
/	dividiert zwei Werte, z.B. a / b
%	berechnet den Divisionsrest (Modulo), z.B. a % b. Nur Ganzzahlen!!

Beispiele, die den % Operator erläutern:

```
13 :  3 = 4 Rest  1  ( 13 %  3 =  1)
36 :  7 = 5 Rest  1  ( 36 %  7 =  1)
43 : 22 = 1 Rest 21  ( 43 % 22 = 21)
```

Es werden immer zuerst die Operatoren auf der rechten Seite des Zuweisungsoperator ausgewertet und dann das Ergebnis der Variablen auf der linken Seite zugewiesen:
kreis_umfang = 2*kreis_radius*3.14;

Für die Auswertung der Operatoren auf der rechten Seite gelten bestimmte Regeln:

- Auch in C ist die mathematische Regel **Punkt vor Strich** gültig.
- Bei gleicher Priorität werden die Operatoren von **links nach rechts** abgearbeitet.
- Wird eine andere Reihenfolge benötigt, so müssen Klammern gesetzt werden.

Die Reihenfolge der Operatoren kann in bestimmten Fällen von Wichtigkeit sein.

Beispiel:
Der folgende Ausdruck würde bei einem short int Datentyp (-32768...32767) zu einem Überlauf und somit zu einem falschen Ergebnis führen:

int a = 6 * 10000 / 2;

Hier wird zuerst die Multiplikation durchgeführt, was zu einem Überlauf führt (6*10000 = 60000). Das Zwischenergebenis wird zu -5536 berechnet und wird somit negativ. Die anschließende Division ändert daran nichts mehr. Würde man hier zuerst die Division und dann die Multiplikation durchführen, hätte der Fehler vermieden werden können:

int a = 6 / 2 * 10000;

Eine andere Möglichkeit wäre gewesen, die Konstante 6 explizit als **long** Konstante anzugeben (siehe auch 4.2.3). Die ganze Berechnung wird dann im **long** Format durchgeführt und ergibt damit keinen Überlauf mehr:

int a = 6L * 10000 / 2;

Eine weitere Gefahr lauert in der Division. Dividiert man zwei Ganzzahlen, so erhält man das ganzzahlige Ergebnis, eventuelle Reste werden abgeschnitten. So ist z.B. 6/4 = 1. Das ist dann erwünscht, wenn man die Nachkommastellen abschneiden will, oder sie nicht interessant sind (z.B. beim Runden auf bestimmte Anzahl Nachkommastellen).
In andere Fällen kann dies jedoch zu falschen Ergebnissen führen. So würde z.B. die folgende Anweisung:

```
float a = 1 / 2;
```

dazu führen, dass der Variablen a der Wert 0 und nicht wie erwartet der Wert 0.5 zugewiesen wird. Der Prozessor berechnet zuerst die Division zweier Ganzzahlen und weist das Ergebnis erst dann der Variablen zu. Hat man es mit Konstanten zu tun, kann man sich behelfen, indem man diese explizit als **float**-Konstanten angibt:

```
float a = 1.0/2.0;
```

Etwas komplizierter wird es bei einer Division von zwei Variablen:

```
int a = 1;
int b = 2;
float c = a / b;
```

Hier hilft nur ein so genannter **Typecast** um eine der Variablen in **float** umzuwandeln:

```
int a = 1;
int b = 2;
float c = (float)a / b;
```

5.3.2. Unäre arithmetische Operatoren

In C gibt es drei unäre arithmetische Operatoren:

Operator	Bedeutung
−	negiert den Wert, z.B. −a
++	erhöht die Variable um 1, z.B. a++
−−	vermindert die Variable um 1, z.B. a−−

Eine Besonderheit sind die Inkrement/Dekrement-Operatoren. Sie können sowohl vor (*Präfix*) als auch hinter (*Suffix*) der Variablen stehen.

Präfix-Schreibweise

Hier werden die ++ und −− Operatoren vor allen anderen Operatoren ausgewertet. Beispiel:

```
vor = ++zahl;
```

Hatte *zahl* zuvor den Wert 5, so wird diese zuerst um 1 erhöht, erhält also den Wert 6, und wird dann der Variablen *vor* zugewiesen. Nach der Anweisung haben die Varia-

blen also die Werte:
vor = 6 und zahl = 6;

Suffix-Schreibweise

Hier werden die ++ und −− Operatoren nach allen anderen Operatoren ausgewertet. Beispiel:

vor = zahl++;

Hatte *zahl* zuvor den Wert 5, so wird diese zuerst um 1 erhöht, erhält also den Wert 6, zuvor wird sie aber der Variablen *vor* zugewiesen. Nach der Anweisung haben die Variablen also die Werte:
vor = 5 und zahl = 6;

5.4. Logische- und Vergleichsoperatoren

Zum Vergleich von Werten stehen in C die folgenden Operatoren zur Verfügung:

Operator	Bedeutung
<	Prüft, ob der linke Wert kleiner ist als der Rechte, z.B. a < b
>	Prüft, ob der linke Wert größer ist als der Rechte, z.B. a > b
<=	Prüft, ob der linke Wert kleiner oder gleich ist als der Rechte, z.B. a <= b
>=	Prüft, ob der linke Wert größer oder gleich ist als der Rechte, z.B. a >= b
! =	Prüft zwei Werte auf Ungleichheit, z.B. a ! = b
==	Prüft zwei Werte auf Gleichheit, z.B. a == b

Ein Vergleich kann zutreffen oder nicht, d.h. es gibt nur 2 Möglichkeiten:

- der Vergleich ist wahr, englisch **TRUE**
- der Vergleich ist falsch, englisch **FALSE**

Vergleiche werden deshalb vor allem für Abfragen verwendet. **TRUE** steht in C für einen Wert verschieden von 0.
Speziell der Gleichheits-Operator ist mit besonderer Vorsicht zu genießen: Es müssen immer zwei Gleichheitszeichen verwendet werden, da er sonst als Zuweisungsoperator ausgewertet wird!!

Bisher kennen wir die Vergleichsoperatoren, aber die Möglichkeit des einfachen Vergleichs von 2 Werten ist noch nicht gerade berauschend. In vielen Fällen will man mehrere Abfragen logisch kombinieren. Zu diesem Zweck stehen in C die **logischen Operatoren** zur Verfügung:

<table>
<tr><th>Operator</th><th>Bedeutung</th></tr>
<tr><td>!</td><td>logisches NOT eines boolschen Wertes (unärer Operator), z.B. !(a == b)</td></tr>
<tr><td>&&</td><td>logisches AND von zwei boolschen Werten, z.B. (a == b) && (c == d)</td></tr>
<tr><td>||</td><td>logisches OR von zwei boolschen Werten, z.B. (a == b) || (c == d)</td></tr>
</table>

Noch eine kurze Anmerkung zur Reihenfolge der Auswertung: In der Operator-Rangreihenfolge liegen die Vergleichsoperatoren höher als die logischen Verknüpfungen. Dementsprechend wären die Klammern in den obigen Beispielen nicht notwendig, denn durch die Rangreihenfolge würden die Ausdrücke ohnehin „richtig“ ausgewertet werden. Allerdings ist der Code bei weitem leichter lesbar und auch durchschaubarer, wenn man trotzdem die Klammerungen vornimmt. Vor allem können sich schnell Flüchtigkeitsfehler einschleichen, wenn man keine explizite Klammerung vornimmt. Es wird als guter Programmierstil angesehen, die beabsichtigte logische Struktur von Auswertungen durch Klammerung sichtbar zu machen, auch wenn sie eigentlich unnötig wären.

Beispiele zur Verwendung des Negationsoperators:

!(zaehler>3)	ist wahr, wenn der Wert von zaehler kleiner oder gleich 3 ist.
!(x==20)	ist wahr, wenn der Wert von s von 20 verschieden ist.
x=!324	weist x den Wert 0 zu, da 324 verschieden von 0, also TRUE ist, und die Negation von TRUE liefert FALSE, welches durch 0 ersetzt wird.

Beispiele zur Verwendung des UND-Operators:

(zaehler<4) && (zaehler>=1)	die Bedingung ist nur erfüllt, wenn der Wert von zaehler kleiner als 4 **und** gleichzeitig auch größer oder gleich 1 ist.
(alter>=18) && (alter<44)	die Bedingung ist nur erfüllt, wenn der Wert von alter kleiner als 44 **und** gleichzeitig auch größer oder gleich 18 ist.

Beispiel zur Verwendung des ODER-Operators:

<table>
<tr><td>(alter<6) || (alter>=65)</td><td>die Bedingung ist erfüllt, wenn der Wert von alter kleiner als 6 oder größer oder gleich 65 ist.</td></tr>
</table>

Keine unnötige Auswertung rechts von && und ||

Wenn der linke Operand bei einem mit && und || verknüpften Ausdruck bereits genug Informationen liefert, um den Wahrheitwsert des gesamten Ausdrucks zu bestimmen, dann wird der rechte Operand erst gar nicht ausgewertet.

&&

Wenn der linke Operand FALSE (in C == 0) liefert, dann ist der Wahrheitswert des gesamten Ausdruck in jedem Fall FALSE. Daher wertet C den rechten Operand erst gar nicht mehr aus. Es kann z.B. zur Vermeidung einer Division durch 0 im folgenden Fall hilfreich sein:

(n!=0) && (sum/n < durchschnitt)
Hier wird der rechte Operand nur ausgewertet, wenn n von 0 verschieden ist. Damit ist sichergestellt, dass sum nie durch 0 geteilt wird.

||

Wenn der linke Operand TRUE (in C != 0) liefert, dann ist der Wahrheitswert des gesamten Ausdruck in jedem Fall TRUE. Daher wertet C den rechten Operand erst gar nicht mehr aus.

So spart sich C unnütze Vergleiche, was natürlich zu schnelleren Programmabläufen führt. Die Regelung kann aber auch zu ärgerlichen Fehlern führen, wenn dies nicht beachtet wird.

5.5. Bitoperatoren

Zur Manipulation von einzelnen Bits in den Variablen bietet C sechs Bitoperatoren an:

<table>
<tr><th>Operator</th><th>Bedeutung</th></tr>
<tr><td>&</td><td>Bitweises-AND (auch Bit-AND), z.B. a & 0x02</td></tr>
<tr><td>|</td><td>Bitweises-OR (auch Bit-OR), z.B. a | 0x02</td></tr>
<tr><td>^</td><td>Bitweises-XOR (auch Bit-XOR), z.B. a ^ 0x02</td></tr>
<tr><td>~</td><td>Bitweises-NOT (auch Bit-NOT), entspricht dem sog. 1-er Komplement, z.B. ~a</td></tr>
<tr><td><<</td><td>Bitweises-left-shift (auch Bit-left-shift), z.B. a << 3</td></tr>
<tr><td>>></td><td>Bitweises-right-shift (auch Bit-right-shift), z.B. a >> 3</td></tr>
</table>

Alle Bit-Operatoren können nur mit *ganzzahligen* Werten verwendet werden!
Achtung: Die Bit-Operatoren unterscheiden sich von den logischen Operatoren. Häufig werden sie verwechselt, da die Schreibweise nahezu identisch ist (& bzw. &&, | bzw. ||).

Die Schiebeoperatoren werden oft auch als Ersatz für eine Multiplikation/Division durch Vielfache der Zahl 2 verwendet. Das Schieben um 1 Bit nach links bewirkt eine Multiplikation um 2, das Schieben nach rechts eine Division durch 2.

Beispiel zur Bildung des Zweier-Komplements unter Verwendung des Einer-Komplements:

Listing 5.1: Berechnung des Zweier-Komplements

```
#include <stdio.h>

int  main()
{
    int  zahl, n_zahl1, n_zahl2;

    zahl=10;

    n_zahl1 = -zahl;          /* minus zahl */
    printf("%d", n_zahl1);
    printf("\n");

    n_zahl2 = ~zahl+1;     /* 2-er Komplement */
    printf("%d", n_zahl2);
    printf("\n");
    return(0);
}
```

Das folgende Beispiel demonstriert die Wirkungsweise des &-Operators:

Listing 5.2: Beispiel zu AND

```
#include <stdio.h>

int main()
{
    int zahl_1, zahl_2;

    zahl_1 = -7;
    zahl_2 = 12;
    printf("%d", zahl_1&zahl_2);
    printf("\n");

    zahl_1 = 32;
    zahl_2 = 743;
    printf("%d",zahl_1&zahl_2);
    printf("\n");
    return(0);
}
```

Das folgende Beispiel demonstriert die Wirkungsweise des |-Operators:

Listing 5.3: Beispiel zu OR

```
#include <stdio.h>

int main()
{
    int zahl_1, zahl_2;

    zahl_1 = -7;
    zahl_2 = 12;
    printf("%d", zahl_1|zahl_2);
    printf("\n");

    zahl_1 = 32;
    zahl_2 = 743;
    printf("%d",zahl_1|zahl_2);
    printf("\n");
    return(0);
}
```

Das folgende Beispiel demonstriert die Wirkungsweise des ∧-Operators:

Listing 5.4: Beispiel zu XOR

```
#include <stdio.h>

int main()
{
    int zahl_1, zahl_2;

    zahl_1 = -7;
    zahl_2 = 12;
    printf("%d", zahl_1^zahl_2);
    printf("\n");

    zahl_1 = 32;
    zahl_2 = 743;
    printf("%d",zahl_1^zahl_2);
    printf("\n");
    return(0);
}
```

5.5.1. Setzen von Bits

Der Bitweise-OR Operator wird häufig zum Setzen einzelner Bits in einer Variablen verwendet.

Beispiel:
In einer short-Variablen *vorn_eins* sei der Wert 1 und in einer anderen short-Variablen *muster* sei der Wert 0xFF00 gespeichert. Um die ersten 8 Bits der Variablen *vorn_eins* auf 1 zu setzen, könnte folgende Anweisung verwendet werden:
vorn_eins = vorn_eins | muster;

	VORZ.	16384 2^{14}	8192 2^{13}	4096 2^{12}	2048 2^{11}	1024 2^{10}	512 2^9	256 2^8	128 2^7	64 2^6	32 2^5	16 2^4	8 2^3	4 2^2	2 2^1	1 2^0		
vor_eins	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	=	1
muster	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	=	0xFF00
ergebnis	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	1	=	-255

Es werden hier die Bits auf 1 gesetzt, die in der Variablen *muster* oder in *vorn_eins* schon gesetzt waren.

5.5.2. Löschen/Prüfen von Bits

Der Bitweise-AND Operator wird häufig zum Löschen einzelner Bits oder zum Testen ob die Bits gesetzt sind verwendet.

Beispiel:
In einer Variablen *vorn_null* ist der Wert 0x7FFF gespeichert. Wir wollen nun die unteren 8 Bits dieser short-Variablen löschen. Die dazu erforderliche Anweisung könnte so aussehen:
vorn_null = vorn_null & 0xFF00;

	VORZ.	16384 2^{14}	8192 2^{13}	4096 2^{12}	2048 2^{11}	1024 2^{10}	512 2^9	256 2^8	128 2^7	64 2^6	32 2^5	16 2^4	8 2^3	4 2^2	2 2^1	1 2^0		
vor_null	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	=	0x7FFF
	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	=	0xFF00
ergebnis	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	1	=	0x7F00

Es werden hier die Bits auf 0 gesetzt, die in der Konstanten auf 0 sind oder in *vorn_null* schon gelöscht waren. Alle anderen Bits bleiben gesetzt.

Auf die gleiche Art und Weise kann man prüfen, ob ein bestimmtes Bits in einer Variablen gesetzt ist. Nehmen wir obiges Beispiel und wollen prüfen, ob in der Variablen *vorn_null* z.B. das Bit 10 (=2^9) gesetzt ist, so könnte diese Anweisung so aussehen:
(vorn_null & 0x0400) == 0x0400

	VORZ.	16384 2^{14}	8192 2^{13}	4096 2^{12}	2048 2^{11}	1024 2^{10}	512 2^9	256 2^8	128 2^7	64 2^6	32 2^5	16 2^4	8 2^3	4 2^2	2 2^1	1 2^0		
vor_null	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	=	0x7FFF
	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	=	0x0400
ergebnis	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	=	0x0400

Es bleibt hier, wie oben, nur das eine Bit auf 1, welches in beiden Werten gesetzt ist. Der obige Ausdruck ist also WAHR, wenn das Bit gesetzt ist bzw. der Ausdruck den Wert 0x0400 liefert.

5.5.3. Invertieren von Bits

Der Bitweise-XOR Operator wird häufig zum Invertieren einzelner Bits in einer Variablen verwendet.

Beispiel:
In einer short-Variablen *umkehr* sei der Wert 0xA7C3 gespeichert. Wir wollen nun die ersten und die letzten 3 Bits der Variablen invertieren, die restlichen Bits sollen unverändert bleiben. Die dazu erforderliche Anweisung könnte so aussehen:
umkehr = umkehr ∧ 0xE007;

	VORZ.	16384 2^{14}	8192 2^{13}	4096 2^{12}	2048 2^{11}	1024 2^{10}	512 2^9	256 2^8	128 2^7	64 2^6	32 2^5	16 2^4	8 2^3	4 2^2	2 2^1	1 2^0		
umkehr	1	0	1	0	0	1	1	1	1	1	0	0	0	0	1	1	=	0xA7C3
	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	=	0xE007
ergebnis	0	1	0	0	0	1	1	1	1	1	0	0	0	1	0	0	=	0x47C4

Hier werden alle Bits, die in beiden Werten identisch sind, auf Null gesetzt. Alle anderen Werte bleiben unverändert. Alle Bits die in der Konstanten auf 1 gesetzt sind, werden daher invertiert.

5.6. Zusammengesetzte Zuweisungsoperatoren

In der Programmierpraxis werden häufig Anweisungen der folgenden Form benötigt:

a = a + b;

Um Schreibarbeit einzusparen, wurden in C die zusammengesetzten Zuweisungsoperatoren eingeführt:

+ =, − =, ∗ =, / =, % =, >>=, <<=, & =, | =, ^=

Der rechts von einem zusammengesetzten Zuweisungsoperator stehende Ausdruck sollte immer geklammert angesehen werden:

var <*op*>= ausdruck;

entspricht immer

var = var <*op*> (ausdruck);

5.7. Priorität und Anwendbarkeit von Operatoren

Wie schon erwähnt haben die Operatoren unterschiedliche Prioritäten. Da man die Priorität oft nicht vollständig wissen kann, ist es zu empfehlen, oft mit runden Klammern zu arbeiten und so die Priorität vorzugeben. Zur Vollständigkeit hier noch die Tabelle, sortiert nach Prioritäten von hoch nach tief:

<table>
<tr><th>Operatoren</th><th>Priorität</th></tr>
<tr><td>()</td><td>hoch</td></tr>
<tr><td>! ∼ ++ −− −(Vorzeichen)</td><td></td></tr>
<tr><td>∗ % /</td><td></td></tr>
<tr><td>+ −</td><td></td></tr>
<tr><td><< >></td><td></td></tr>
<tr><td>< <= > >=</td><td></td></tr>
<tr><td>== !=</td><td></td></tr>
<tr><td>&</td><td></td></tr>
<tr><td>^</td><td></td></tr>
<tr><td>|</td><td></td></tr>
<tr><td>&&</td><td></td></tr>
<tr><td>||</td><td></td></tr>
<tr><td>? :</td><td></td></tr>
<tr><td>= += −= ∗= /= %= >>= <<= &= |= ^=</td><td>niedrig</td></tr>
</table>

Die Zuweisungsoperatoren haben demnach die niederste Priorität, d.h. es wird zuerst alles andere Berechnet und dann erst wird die Zuweisung des Ergebnisses durchgeführt. Innerhalb einer Zeile haben die Operatoren dieselbe Priorität. Hier gilt zumeist die Abarbeitungsrichtung von **links nach rechts**. Ausnahmen bilden hier die Zuweisungsoperatoren sowie die Operatoren „!, ∼, ++, −−,−“.

Für Ganzzahlen sind alle Operatoren erlaubt. Bei Gleitkommazahlen sind folgende Operatoren **nicht**erlaubt:

~, %, <<, >>, &, |, ^

sowie alle damit zusammengesetzten Zuweisungsoperatoren.

5.8. lvalue

Ein lvalue ist ein Ausdruck, der sich auf ein Objekt im Arbeitsspeicher, z.B. einer Variablen, bezieht. Bei jeder Zuweisung mit dem Zuweisungsoperator muss der linke Ausdruck ein lvalue, also z.B. ein Variablennamem sein.
Keine lvalues sind: Konstanten, Feldnamen ohne folgende Indexangabe, Funktionsnamen, Namen aus einer Aufzählung (enum), Objekte, die explizit mit const als konstant vereinbart wurden.

Beispiele:

```
int a,b;
a   = 27;  /* richtig */
a  += 27;  /* richtig */
b++ = 27;  /* falsch, b++ ist kein lvalue */
a+b = 27;  /* falsch, a+b ist kein lvalue */
27  = a;   /* falsch, 27 ist kein lvalue */
a   = b;   /* richtig */
```

Die falschen Zuweisungen führen schon beim Compilieren zu Fehlermeldungen!

6. Kontrollstrukturen

Jetzt soll ein wenig Leben in unsere Programme kommen, denn wir beschäftigen uns in diesem Kapitel mit Konstrukten, die uns helfen, unseren Programmablauf zu beeinflussen. Wir wollen ja schließlich auf Zustände reagieren können und nicht einfach nur Anweisungen hintereinander schreiben, die dann immer genau in dieser Reihenfolge ausgeführt werden können. Salopp gesprochen geht es bei den Kontrollstrukturen um Möglichkeiten, wie wir die Reihenfolge der Ausführung nach gewissen Kriterien steuern können.

6.1. Blöcke

Die einfachste Kontrollstruktur ist der Block. Er wird im Struktogramm als ein Rechteck dargestellt. Ein Block dient dazu mehrere Anweisungen zusammenzufassen. Mit einem solchen Block haben wir schon in der main-Funktion gearbeitet. Er ist immer von geschweiften Klammern umschlossen. Blöcke können auch ineinander verschachtelt werden. Wichtig ist nur, dass zu jeder öffnenden geschweiften Klammer, eine schließende geschweifte Klammer folgt. Durch folgende Regeln lässt sich die Übersicht leicht erhalten:

- Jeder Block beginnt mit der geschweiften Klammer in einer neuen Zeile
- Nach einem Blockanfang werden die Anweisungen eingerückt (z.B. mit Tabluator oder Leerzeichen)
- Die schließende geschweifte Klammer wird wieder ausgerückt, so dass die letzte schließende Klammer wieder in der ersten Spalte steht.

```
Anweisung1;
{ // Beginn Block 1
  Anweisung2;
  Anweisung3;
  {  // Beginn Block 2
     Anweisung4;
     Anweisung5;
     Anweisung6;
  }  // Ende Block 2
  Anweisung7;
} // Ende Block 1
```

Am Ende eines Blockes steht kein Semikolon!!

6.2. Verzweigungen

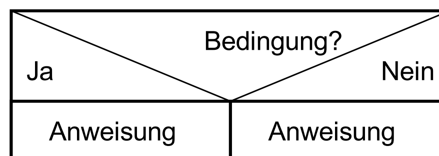
Mittels Verzweigungen oder auch Entscheidungen kann man im Programm an eine andere Stelle verzweigen. In C gibt es zwei verschiedene Varianten der Verzweigung.

6.2.1. Einfache Verzweigung

Mittels einer **if-else** Struktur steuert man den Programmfluss entsprechend dem Ergebnis der Auswertung einer Bedingung. Formal sieht eine if-else Struktur folgendermaßen aus:

```
if (ausdruck)
{
  Anweisungen;
}
else
{
  Anweisungen;
}
```

Im Struktogramm wird die if-else folgendermaßen dargestellt:



Der nach *if* in Klammern angegebene Block wird dann ausgeführt, wenn der *Ausdruck* erfüllt ist. Der Ausdruck ist erfüllt wenn er von 0 verschieden ist (TRUE). Anderenfalls wird der Block im *else* Zweig ausgeführt. Folgende Regeln gilt es bei der if-else Struktur zu beachten:

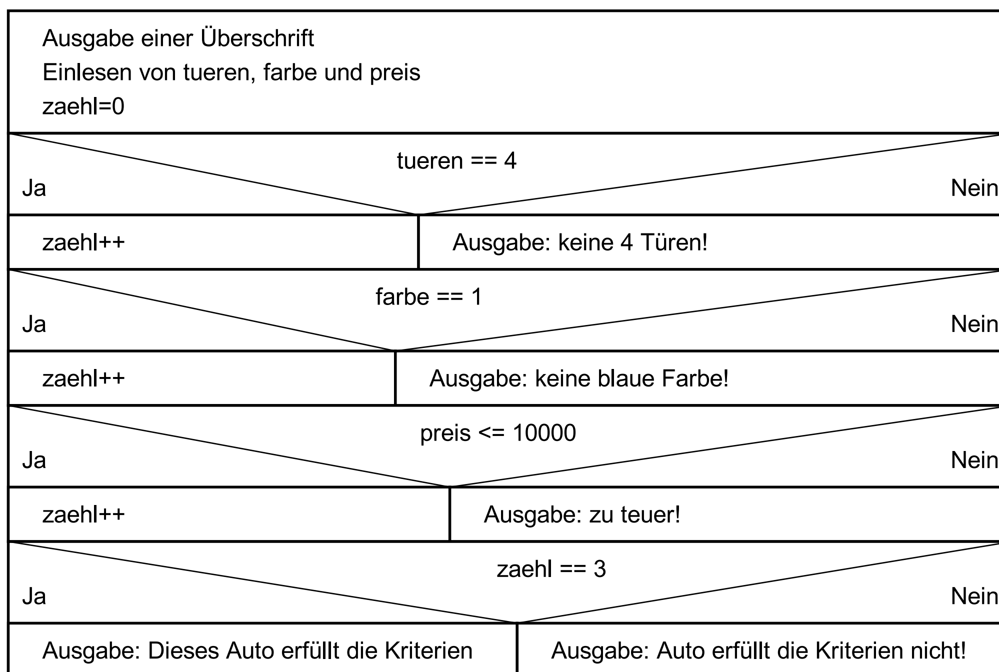
- Runde Klammern schließen die Bedingung (Ausdruck) ein und dürfen nicht ausgelassen werden.
- Nach der Bedingung darf kein Strichpunkt stehen, denn dieser würde bereits als Anweisung interpretiert werden (if ohne Bedingung, was jedoch keinen Sinn macht!).
- Anstatt einer einzelnen Anweisung kann natürlich auch ein Block stehen, falls mehrere Anweisungen hintereinander ausgeführt werden müssen. Zur Vereinfachung sollten wie oben immer Blöcke gesetzt werden, so dass es nicht zu unvorhergesehenen Reaktionen bei der Abarbeitung des Programms kommt.
- Der else-Zweig ist *optional*. Er kann also weggelassen werden, wenn er nicht gebraucht wird.

- Die Bedingung ist immer erfüllt, sobald der Ausdruck von 0 verschieden ist. Er kann daher auch nur aus einer Variablen bestehen z.B. wird statt
  if (zahl != 0) auch oft **if (zahl)** und statt
  if (zahl == 0) auch oft **if (!zahl)** angegeben.

Beispiel:
Sie wollen ein Auto kaufen, das folgende Kriterien erfüllt:

- Türanzahl: 4
- Farbe: blau (Eingabe 1 für blau, 0 für andere Farbe)
- Preis: höchstens 10000Euro

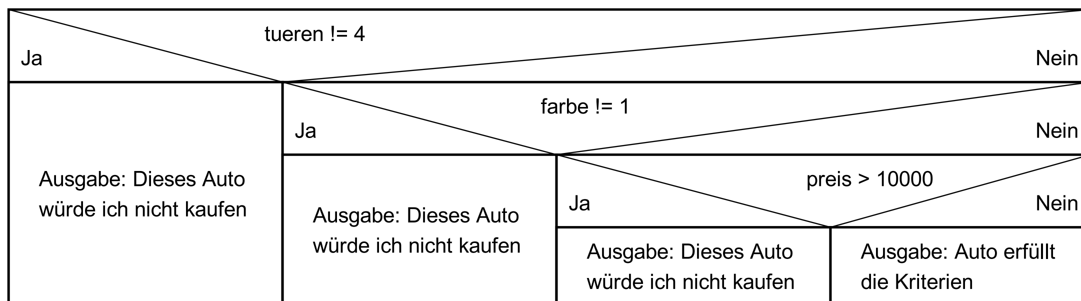
Stellen Sie sich vor, dass Sie die Daten von einem interessanten Auto am Bildschirm eingegeben und Ihr Programm gibt dann aus, ob dieses Auto Ihre Forderungen erfüllt oder nicht. Zuerst soll ein Struktogramm entworfen werden, das danach in ein C-Programm umgesetzt wird.



Listing 6.1: Auswahlkriterien beim Autokauf1

```
#include <stdio.h>

int  main()
{
    int     tueren,  farbe,  zaehl;
    long    preis;

    printf("%45s\n","Autokauf");
    printf("%45s\n","========");
    printf("\n\n");

    printf("Geben␣Sie␣die␣Tueranzahl␣ein:␣");
    scanf("%d",&tueren);

    printf("Ist␣das␣Auto␣blau,␣dann␣geben␣Sie␣1,␣andernfalls␣0␣ein:␣");
    scanf("%d",&farbe);

    printf("Geben␣Sie␣den␣Preis␣des␣Autos␣ein:␣");
    scanf("%ld",&preis);
    printf("\n");

    zaehl=0;        /*  Die Variable  zaehl  wird mit 0 vorbesetzt   */

    if (tueren==4)
    {
       ++zaehl;
    } else
    {
       printf("␣-->␣keine␣4␣Tueren␣!\n");
    }

    if (farbe)
    {
       ++zaehl;
    } else
    {
       printf("␣-->␣keine␣blaue␣Farbe␣!\n");
    }

    if (preis<=10000)
    {
       ++zaehl;
    } else
    {
       printf("␣-->␣zu␣teuer␣!\n");
    }
    printf("\n");

    if (zaehl==3)
    {
        printf("Dieses␣Auto␣erfuellt␣Ihre␣Vorstellungen␣!\n\n");
        printf("Sie␣koennen␣dieses␣Auto␣ohne␣Bedenken␣kaufen␣!\n");
    } else
    {
       printf("Sie␣sehen␣ja␣selbst,␣dass␣dieses␣Auto␣nicht␣die␣geforderten\n");
       printf("Kriterien␣erfuellt␣!\n\n");
       printf("Lassen␣Sie␣lieber␣die␣Finger␣davon␣!\n");
    }
    return(0);
}
```

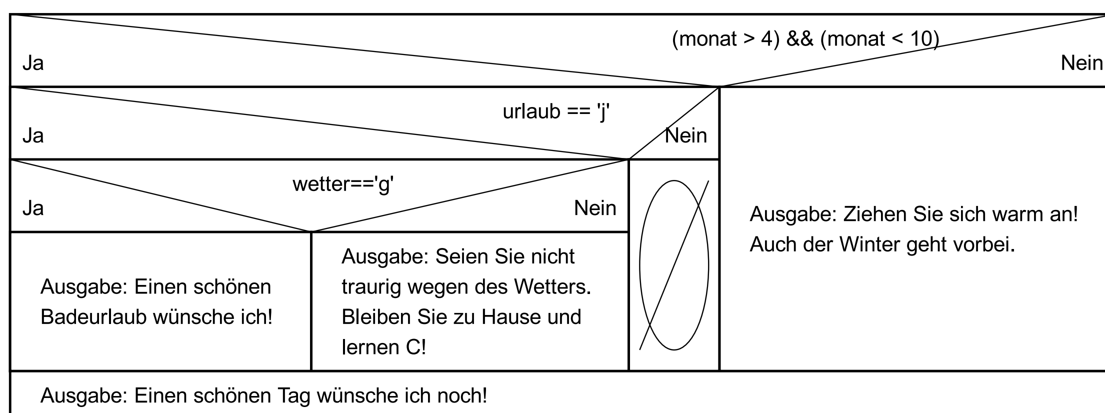
Wenn wir beim letzten Programm auf die Ausgabe der Gründe für einen Nichtkauf verzichtet hätten, könnte der Programmteil, der die Kriterien überprüft, wie folgt aussehen:



Listing 6.2: Auswahlkriterien beim Autokauf2

```
....
    if (tueren!=4)
    {
       printf("Dieses Auto wuerde ich nicht kaufen!");
    } else
       if (!farbe)
       {
          printf("Dieses Auto wuerde ich nicht kaufen!");
       } else
          if (preis>10000)
          {
             printf("Dieses Auto wuerde ich nicht kaufen!");
          } else
          {
             printf("Dieses Auto erfuellt die Kriterien!");
          }
....
```

if-Anweisungen lassen sich, wie alle Anweisungen in C, ebenfalls verschachteln. Innerhalb eines Blockes können wiederum if-else Strukturen untergebracht werden. Ein kleines Beispiel hierzu:



Listing 6.3: Urlaubsgedanken

```
if ((monat>4) && (monat<10))
{
  if (urlaub=='j')
    if (wetter=='g')
    {
      printf("Einen schoenen Badeurlaub wuensche ich noch!");
    } else
    {
      printf("Seinen Sie nicht traurig wegen des Wetters!\n");
      printf("Bleiben Sie zu Hause und lernen C.\n");
    }
} else
{
  printf("Ziehen Sie sich warm an.\n Auch der Winter geht vorbei!");
}
printf("Einen schoenen Tag wuensche ich noch!");
```

Zu beachten ist hier der Bezug des else-Zweiges einmal zur ersten und einmal zur dritte if-Abfrage. Der else-Zweig gehört immer zur letzten Ebene der if-Abfragen. Soll dies geändert werden, müssen entsprechende neue Blöcke mit geschweiften Klammern aufgezogen werden (daher am besten immer mit diesen Blöcken arbeiten, auch wenn sie nicht notwendig wären!)

In C gibt es noch eine Kurzform für obige if-else Struktur. Sie wird selten benutzt, hat aber in manchen Fällen durchaus ihre Dahseinsberechtigung (z.B. bei Parameterübergaben an Funktionen oder Zuweisungen). Zur Erklärung ein kleines Beispiel:

```
if (zahl1 > zahl2)
  max = zahl1;
else
  max = zahl2;
```

Diese doch recht häufige Folge von Anweisungen lässt sich auch einfacher Schreiben:

```
max = (zahl1 > zahl2) ? zahl1 : zahl2;
```

Formal hat diese **bedingte Bewertung** folgenden Aufbau:

***Ausdruck1* ? *Ausdruck2* : *Ausdruck3*;**

Zunächst wird *Ausdruck1* bewertet. Ist sein Wert verschieden von 0 (TRUE), so wird *Ausdruck2* bewertet andernfalls *Ausdruck3*. Es wird immer nur einer der Ausdrücke 2 oder 3 bewertet. Dieser ist dann das Ergebnis der gesamten bedingten Bewertung.

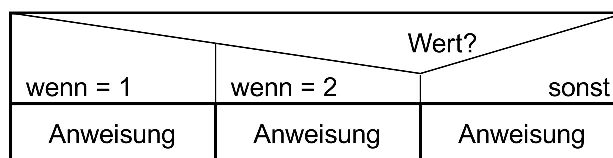
Zu beachten ist, dass nach *Ausdruck2* kein Semikolon steht, wohl aber am Ende der gesamten Anweisung!

6.2.2. mehrfache Verzweigung

Mit der **switch**-Anweisung kann unter mehreren Alternativen ausgewählt werden, nicht nur unter zwei wie bei der if-Anweisung. Der formale Aufbau der switch-Anweisung lautet:

```
switch (Ausdruck)
{
  case Ausdruck1: Anweisungen;
                  break;
  case Ausdruck2: Anweisungen;
                  break;
  case Ausdruck3: Anweisungen;
                  break;
  default: Anweisungen;
           break;
}
```

Es wird der bei *switch* angegebene Ausdruck ausgewertet und das Ergebnis mit den einzelnen *case*-Ausdrücken, die **int**- oder **char**-Werte (Konstanten) liefern müssen, verglichen. Wird dort keine Übereinstimmung gefunden, wird in den *default*-Zweig gesprungen. Dieser *default*-Zweig ist optional, sollte jedoch immer vorhanden sein.
Im Struktogramm sieht die Anweisung wie folgt aus:



Folgende Regeln gilt es bei der switch-Anweisung zu beachten:

- Die Reihenfolge der case und default Marken ist beliebig, es dürfen jedoch keine doppelten Konstanten vorhanden sein.
- Um die switch-Anweisung zu verlassen ist der Befehl **break** notwendig. Dieser Befehl wird später noch behandelt. Hier nur so viel dazu: Wird er nicht verwendet, so arbeitet die switch-Anweisung ab einem gefundenen case-Zweig alle Anweisungen bis nach unten ab!
- In den case-Marken sind nur ganzzahlige Konstanten (char, int) erlaubt. Mit Zeichenketten oder Gleitkommazahlen kann in der switch-Anweisung nicht gearbeitet werden.
- Alle case-Marken, auch die Letzte, mit break abschließen. Der Grund liegt in einer evtl. späteren Erweiterung bei welcher der vorhandene Code nicht mehr geprüft wird.
- default-Marke sollte immer mit angegeben werden. In der Programmierung kann man praktisch nie sicher sein, dass alle möglichen Fälle zu 100% abgedeckt sind.

Als Beispiel hier ein Währungsumrechner von Schweizer Franken in EURO. Der Benutzer soll wählen, in welche Richtung er die Geldbeträge umrechnen möchte. Als Kurs wurde hier der Faktor 1.4 angenommen. Das Programm könnte im Struktogramm folgendermaßen aussehen:

Auswahl der Währungsumrechnung		
wenn = 1	wenn = 2	sonst
Eingabe des Franken-Betrags Berechnung EURO-Betrag Ausgabe des EURO-Betrags	Eingabe des EURO-Betrags Berechnung des Franken-Betrags Ausgabe des Franken-Betrags	Fehlermeldung

Listing 6.4: Währungsumrechner mit switch

```
#include <stdio.h>

int main()
{
  int wahl;
  double sf,euro;

  clrscr();
  printf("\n\nWaehrungsumrechnung Schweizer Franken-Euro\n");
  printf("------------------------------------------\n\n");
  printf("Geben Sie ein:\n");
  printf("1 fuer Umrechnung SF in Euro\n");
  printf("2 fuer Umrechnung Euro in SF");
  scanf("%d",&wahl);

  switch (wahl)
  {
    case 1:
      printf("\nGeben Sie den Franken-Betrag ein:");
      scanf("%lf",&sf);
      euro = sf / 1.4;
      printf("Sie erhalten %10.2lf Euro\n",euro);
      break;
    case 2:
      printf("\nGeben Sie den Euro-Betrag ein:");
      scanf("%lf",&euro);
      sf = euro * 1.4;
      printf("Sie erhalten %10.2lf Franken\n",sf);
      break;
    default:
      printf("\nFalsche Eingabe !");
  }
  return(0);
}
```

6.3. Wiederholungen

Schleifen oder Wiederholungen dienen dazu, eine bestimmte Programmstelle mehrmals auszuführen. In C werden drei unterschiedliche Schleifenarten unterschieden:

- Zählergesteuert
- Fußgesteuert
- Kopfgesteuert

Welche der drei Schleifen soll man nun benutzen? Das ist immer davon abhängig, welche Aufgaben damit zu erledigen sind. Eine kleine Auswahlhilfe soll hier gegeben werden:

Anzahl der Durchgänge bekannt?
Ja
Nein
Zählergesteuerte Schleife
Ja
Nein
Fußgesteuerte Schleife
Kopfgesteuerte Schleife

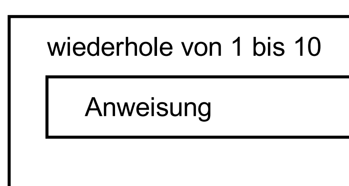
6.3.1. Zählergesteuerte Schleife

Syntaktisch sieht die **for**-Schleife wie folgt aus:

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

Ausdruck1 initialisiert die Schleifen/Zählervariable.
Ausdruck2 legt das Abbruchkriterium für die Schleife fest. Die Anweisungen werden wiederholt, solange Ausdruck2 nicht erfüllt ist.
Ausdruck3 verändert die Schleifenvariable bei jedem Durchgang, z.B. Inkrementieren um 1.

Im Struktogramm sieht das Ganze folgendermaßen aus:



Hier einige Regeln zur Verwendung der for-Schleife:

- Für die for-Schleife wird immer eine Zählvariable benötigt, die am besten eine ganzzahlige Variable ist (int,char).
- Vorsicht: Kein Semikolon am Ende des Schleifenkopfs. Dies würde bewirken, dass nur eine leere Anweisung zur Schleife gehört und diese dadurch nicht das gewünschte Ergebnis liefert!
- Die Ausdrücke in der Klammer können ganz oder teilweise fehlen, jedoch müssen die Semikolon immer vorhanden sein!
- Die Abbruchbedingung sollte immer so formuliert sein, dass die Schleife bearbeitet wird **solange** die Bedingung erfüllt ist, d.h. meist steht als Abbruchbedingung so etwas wie *zählvariable <= ende* oder *zählvariable > ende*, nie aber *zählvariable == ende*.
- Nach dem letzten Ausdruck in den runden Klammern folgt kein Semikolon!
- Nach dem Verlassen der Schleife ist die Zählvariable über den Endwert hinausgelaufen. Dies gilt es zu berücksichtigen, wenn mit dem Wert im Programm weitergearbeitet werden soll.
- Die Zählvariable sollte in der Schleife nicht verändert werden.

Beispiel:
Es soll die geometrische Reihe

$$1 + \frac{1}{2} + \frac{1}{4} + ... + \frac{1}{2^n}$$

berechnet werden, wobei n einzugeben ist. Diese Reihe besteht aus n+1 Summanden.

Initialisierung der Variablen:
reihe_teiler = 1
summe = 1

Eingabe von n (ganzzahlig)

wiederhole von 1 bis n

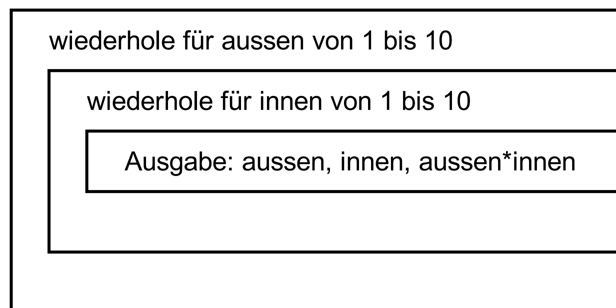
reihe_teiler = reihe_teiler / 2
summe = summe + reihe_teiler

Ausgabe der Summe

Listing 6.5: Berechnung der geometrischen Reihe

```
#include <stdio.h>

int main()
{
   int      zaehl, n;
   double   reih_teil=1.0, summe=1.0;

   printf("\n\nBerechnung der geometrischen Reihe\n");
   printf("==================================\n\n\n\n");

   printf("Geben Sie  n  ganzzahlig positiv ein !\n");
   scanf("%d",&n);

   for (zaehl=1 ; zaehl<=n ; ++zaehl)
   {
       reih_teil=reih_teil/2.0;
      summe+=reih_teil;
   }

   printf("\n\n\nDie Summe der geometrischen Reihe bis %d ist:\n",n);
   printf("%lf\n", summe);
   return(0);
}
```

Auch die for-Schleifen lassen sich wiederum ineinander verschachteln. Dies wird besonders bei der Bearbeitung von mehrdimensionalen Felder und bei der Bildschirmausgabe verwendet.
Wir wollen mit Hilfe eines kleinen C-Programms das Einmaleins auf dem Bildschirm ausgeben:



Listing 6.6: Ausgabe des kleinen Einmaleins

```
#include <stdio.h>

int main()
{
    int     aussen,   innen;

    printf("\n\nDas Einmaleins\n");
    printf("==============\n\n\n\n");

    for (aussen=1 ; aussen<=10 ; aussen++)
    {
       for (innen=1 ; innen<=10 ; innen++)
       {
          printf("%30d * %2d = %3d\n",aussen,innen,aussen*innen);
       }
    }
    return(0);
}
```

Es soll ein Programm erstellt werden, das ein Quadrat in der Mitte des Bildschirms zeichnet. Die Größe soll vom Benutzer eingegeben werden können. Der Bildschirm in den Konsolen hat eine Breite von 80 Zeichen.

Listing 6.7: Zeichnen eines Quadrats auf dem Bildschirm

```
#include <stdio.h>

int  main()
{
    int i, j, groesse;

    printf("Ausgabe eines Rechtecks in der Mitte eines Bildschirms\n");
    printf("==============================================================\n\n");
    printf("Groesse des Rechtecks: ");
    scanf("%d", &groesse);

    printf("\n\n");

    for (i=1 ; i<=groesse ; i++)
    {
       for (j=1 ; j<=40-groesse/2 ; j++) /* Einruecken in einer Zeile */
          printf(" ");
       for (j=1 ; j<=groesse ; j++)  /* In einer inneren Schleife */
          printf("*");                /* Sternchen fuer eine Zeile ausgeben */
       printf("\n");   /* Zeilenvorschub in aeusseren Schleife */
    }
    return(0);
}
```

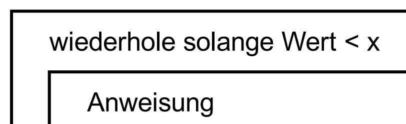
6.3.2. kopfgesteuerte Schleife

Syntaktisch sieht die **while**-Schleife wie folgt aus:

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

Vor jedem Schleifendurchgang wird der *Ausdruck* berechnet. Solange das Ergebnis verschieden von 0 ist (TRUE), wird der Schleifenblock ausgeführt.

Da die Abfrage der Schleifen-Bedingung wie bei der for-Schleife am Anfang durchgeführt wird, wird die while-Schleife genauso wie die for-Schleife dargestellt.



Jede **for**-Schleife

```
for (Audruck1; Ausdruck2; Ausdruck3)
  Anweisung;
```

lässt sich auch durch eine **while**-Schleife formulieren:

```
Audruck1;
while (Ausdruck2)
{
  Anweisung;
  Ausdruck3;
}
```

Wann soll eine **for** und wann eine **while**-Schleife in C verwendet werden?

- **for**-Schleifen werden üblicherweise dann verwendet, wenn ein ganzer Bereich beginnend bei einem bestimmten Startwert bis zu einem vorgegebenen Endwert mit einer gewissen Schrittweite zu durchlaufen ist.
- **while**-Schleife dagegen werden immer dann verwendet, wenn lediglich ein Endekriterium gegeben ist und es nicht vorhersagbar ist, wie oft die Anweisungen ausgeführt werden müssen.

Im folgenden Beispiel sollen beliebige Zeichen von der Tastatur eingelesen und in Großbuchstaben umgewandelt werden. Da hier nicht sicher ist, wann der Benutzer alle Zeichen eingegeben hat, wird die while-Schleife verwendet. Dabei spielt die Konstante EOF[1] ein wichtige Rolle, kennzeichnet sie doch das Ende der Eingabe oder einer Datei. Bei der Eingabe durch die Tastatur kann sie durch Drücken von *Strg+Z* eingegeben werden.

Listing 6.8: Umwandeln der Eingabe in Großbuchstaben

```c
#include <stdio.h>
#include <ctype.h>

int  main()
{
   int zeich;

   zeich=getchar();
   while (zeich != EOF)
   {
       putchar(toupper(zeich));
       zeich=getchar();
   }
   return(0);
}
```

Das Makro **toupper()** wandelt die Klein- in Großbuchstaben um.

[1]EOF = End of File

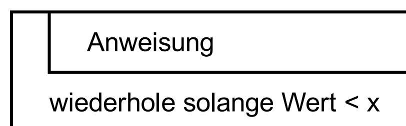
6.3.3. fußgesteuerte Schleife

Im Unterschied zu beiden obigen Schleifen wird bei der **do..while**-Schleife die Bedingung am Ende abgefragt, d.h. die Anweisungen in der Schleife werden zumindest einmal ausgeführt.
Syntaktisch sieht die **do..while**-Schleife wie folgt aus:

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

Nach jedem Schleifendurchgang wird der *Ausdruck* berechnet. Solange das Ergebnis verschieden von 0 ist (TRUE), wird der Schleifenblock ausgeführt.

Im Struktogramm wird die do..while-Schleife wie folgt dargestellt:



In der Praxis werden while-Schleifen öfter verwendet als do..while-Schleifen. Die fußgesteuerten Schleifen werden immer dann verwendet, wenn sichergestellt werden muss, dass der Schleifenkörper mindestens einmal ausgeführt wird.
Zur besseren Übersicht sollte das abschließende **} while(..);** immer in derselben Zeile stehen. Somit ist eindeutig erkennbar, dass hier nicht eine neue while-Schleife beginnt, sondern dass es sich um den Abschluss einer do..while-Schleife handelt. Wichtig ist auch, dass das Semikolon am Ende nicht vergessen wird!

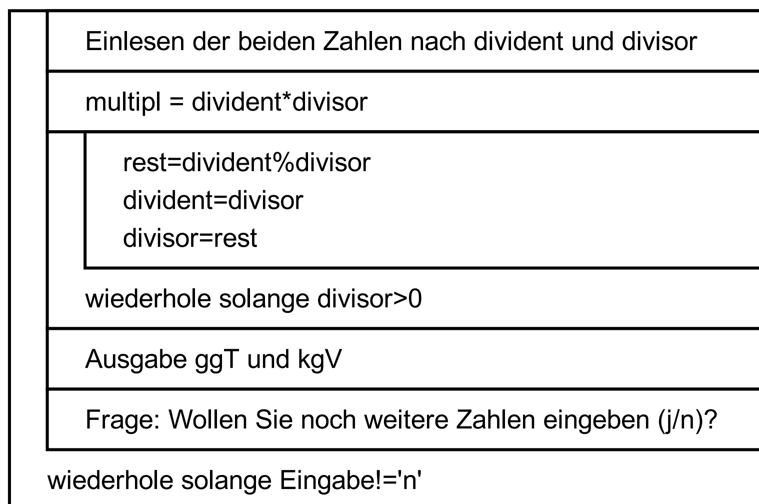
Beispiel zur Berechnung des größten gemeinsamen Teilers (ggT) und das kleinste gemeinsam Vielfache (kgV) zweier int-Zahlen. Den ggT zweier Zahlen kann man nach dem Euklid´schen Algorithmus bestimmen:
Die erste Zahl ist der Divident und die zweite Zahl der Divisor. Man geht nun nach folgenden Verfahren vor:

1. Ganzzahlige Division mit aktuellen Dividenten und Divisor durchführen

2. Ist der Rest ungleich 0, so wird der Divisor zu neuem Dividenten und der Rest zum neuen Divisor und wieder mit Punkt 1 fortfahren.

3. Ist der Divisionrest gleich 0, so ist der Divisor der ggT und das Verfahren ist beendet.

Das kgV ist dann das Produkt dieser beiden Zahlen, geteilt durch ggT.



Listing 6.9: Berechnung von ggT und kgV zweier Zahlen

```c
#include <stdio.h>
#include <ctype.h>

int  main()
{
    long int   divident, divisor, multipl, rest;

    printf("ggT und kgV\n"
           "===========\n");
    do
    {
        printf("\n\nGeben Sie zwei Zahlen (mit Komma getrennt) ein: ");
        scanf("%ld, %ld", &divident, &divisor); fflush(stdin);

        multipl=divident*divisor;

        do  /* EUKLIDsche Algorithmus:  */
        {
            rest = divident % divisor;
            divident = divisor;         /* Nach Verlassen dieser Schleife   */
            divisor = rest;             /* befindet sich der ggT in der     */
        } while (divisor>0);            /* Variablen   divident             */

        printf("ggT = %ld\n", divident);                /* Ausgabe des ggT */
        printf("kgV = %ld\n\n\n", multipl/divident);    /* Ausgabe des kgV */

        printf("Wollen Sie noch zwei Zahlen eingeben (j/n)? ");
    } while (tolower(getchar())!='n');
    return(0);
}
```

6.3.4. break und continue

Nicht immer ist es sinnvoll, eine Schleife nur über das Abbruchkriterium zu verlassen. Die **break**-Anweisung beendet eine Schleife oder *switch*-Anweisung sofort. Bei mehrfach verschachtelten Schleifen bewirkt das **break** nur den Abbruch einer Schleifenebene.

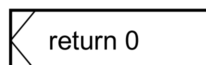
Wann ist die break-Anweisung zu verwenden:

1. **Verlassen einer switch-Anweisung**
   Diese Verwendung wurde schon besprochen und dient der korrekten Bearbeitung der switch-Anweisung.

2. **Verlassen von Schleifen**
   Manchmal treten Situationen auf, bei denen eine Schleife sofort verlassen werden kann. Dies widerspricht zwar den strengen Gesetzen der strukturierten Programmierung, dient aber oft zur Geschwindigkeitserhöhung der Programme.

3. **Bei Endlosschleifen**
   Werden Endlosschleifen benutzt, müssen diese trotzdem bei Programmende verlassen werden. Dies wird durch eine break-Anweisung erreicht.

Im Struktogramm hat die break-Anweisung folgendes Symbol:



Beispiel:
Eine Marktfrau wird gefragt, wie viele Eier sie bei sich habe. Sie antwortet: "Wenn ich 11er Packungen mache, dann bleiben 5 übrig. Verwende ich dagegen 23er Packungen, so verbleiben 3". Wie viele Eier hat sie mindestens?

Listing 6.10: Problem der Eierfrau1 - Endlosschleife

```
#include <stdio.h>

int  main()
{
    int i=1;
    while (1)
    {
        if (i%11==5 && i%23==3)
            break;
        i++;
    }
    printf("%d Eier\n", i);
    return(0);
}
```

Vertreter der einen strukturierten Programmierung hätten die Aufgabe wahrscheinlich wie folgt gelöst:

Listing 6.11: Problem der Eierfrau2

```
#include <stdio.h>

int  main()
{
    int i=0;
    do
    {
        i++;
    } while ( ((i%11)!=5) || ((i%23)!=3) );
    printf("%d Eier\n", i);
    return(0);
}
```

Soll während eines Schleifendurchlaufs sofort der nächste Durchlauf gestartet werden, so kann dazu die **continue**-Anweisung verwendet werden. Die continue-Anweisung ist also der break-Anweisung sehr ähnlich, nur bewirkt sie das Gegenteil. Sie bewirkt nicht den Abbruch der ganzen Schleife sondern nur den Abbruch des aktuellen Schleifendurchgangs. Das Programm verzweigt sofort zum Schleifenende. Bei *while*-Schleifen und *do..while*-Schleifen bewirkt die Anweisung, dass sofort die Abbruchbedingung wiederum kontrolliert wird. Bei *for*-Schleifen wird die Zählvariable verändert und anschließend wird die Abbruchbedingung geprüft.

6.3.5. Sprünge mit goto

Nur zur Vollständigkeit soll hier noch die **goto**-Anweisung erwähnt werden. Die Springerei mit goto ist in der strukturierten Programm nicht mehr notwendig und sollte daher nicht mehr verwendet werden.

Die Syntax lautet:

```
goto Marke;
...
Marke: Anweisungen;
```

Die *goto*-Anweisung bewirkt einen Sprung zur Programmstelle *Marke*.

Die Markennamen werden nach den gleichen Regeln wie Variablennamen vergeben, gefolgt von einem Doppelpunkt. Nach einer Marke muss mindestens eine Anweisung stehen, eventuell auch nur die leere Anweisung.

6.3.6. Endlosschleifen

Endlosschleifen werden häufig in der Steuerungstechnik oder der Systemprogrammierung eingesetzt. Sie wiederholen, wie der Name schon sagt, den Schleifenkörper endlos. So ist z.B. bei einem Controller die Hauptschleife meist eine Endlosschleife, die immer wieder das Programm zyklisch von vorne abarbeitet. Endlosschleifen werden nur dann abgebrochen, wenn eine **break**-Anweisung im Schleifenkörper aufgerufen oder das Programm von außen beendet wird.
Es lassen sich mit jeder der drei Schleifentypen Endlosschleifen erzeugen:

```
for (;;;)
{...}

while (1)
{...}

do
{...} while (1);
```

Entweder wird hier die Abbruchbedingung immer auf 1 (TRUE) gesetzt oder einfach weggelassen.
Ein Beispiel für die Anwendung ist z.B. die ständige Wiederholung eines Programmteils, bis der Benutzer eine bestimmte Eingabe (z.B. in einem Auswahlmenu) macht.

```
...
do
{
  printf("Bitte auswählen:\n");
  printf("1: xxxx\n");
  printf("2: xxxx\n");
  printf("3: Ende\n");
  scanf("%d",&eingabe);
  ...
  if (eingabe==3)
    break;  /* Hier wird die Schleife verlassen */
} while(1);
...
```

7. Felder

In der Programmierpraxis sind häufig Probleme anzutreffen, bei denen man nicht mit einfachen Variablen auskommt, sondern einen ganzen Block von Variablen des gleichen Typs benötigt. Ein Feld oder *Array* ist die Zusammenfassung einer Anzahl von gleichartigen Objekten, wie z.B. int-Variablen.

7.1. eindimensionale Felder

Die einfachste Art von Felder sind die eindimensionalen Felder. Ein Beispiel aus der Mathematik wäre z.B. ein Vektor mit seinen Koordinaten. Ein eindimensionales Feld wird folgendermaßen deklariert:

***Typ* Name[*Anzahl*];**

Mit dieser Angabe wird ein Feld *Name* mit *Anzahl* Elementen von Datentyp *Typ* deklariert. Die Anzahl muss immer in eckigen Klammern angegeben werden und muss schon beim Compilieren feststehen, d.h. es sich immer um eine ganzzahlige Konstante handeln. Ein kleines Beispiel für die Reservierung eines Speicherblocks von 20 char-Variablen:

```
char Zeichen[20];
```

Die einzelnen Elemente belegen im Speicher immer einen zusammenhängenden Speicherbereich:

Zeichen1	
Zeichen2	
Zeichen3	
Zeichen4	
...	

Die Größe des Speicherplatzes, welches ein Feld belegt, lässt sich leicht berechnen:

Anzahl Bytes pro Element * Anzahl Elemente

So wird z.B. für obiges Beispiel ein Speicher von *1Byte * 20 = 20 Byte* belegt.

In C sind nur statische Felder erlaubt, d.h. die Größe des Feldes muss schon bei der Compilierung festgelegt werden. Es kann also für die Feldgröße nur eine Konstante verwendet werden!
Um die Größe zur Laufzeit festzulegen, ist eine dynamische Speicherreservierung erforderlich. Dieses Thema wird noch später im Skript behandelt.

7.2. mehrdimensionale Felder

Häufig reicht in der Praxis ein eindimensionales Feld nicht aus. C bietet die Möglichkeit, Felder mit n-Dimensionen zu erstellen. Dies ist aber oft sehr undurchschaubar und sollte wenn möglich vermieden werden. Felder bis zu 3 Dimensionen lassen sich jedoch noch einigermaßen verständlich behandeln (z.B. für Tabellen, Matrizen, räumliche Gebilde...)
Soll ein mehrdimensionales Feld angelegt werden, so ist folgende Deklaration zu verwenden:

***Typ* Name [*Dimension1*][*Dimension2*]...[Dimension n];**

Für jede Dimension ist die Anzahl der Elemente in einer weiteren eckigen Klammer anzugeben. Ansonsten gelten dieselben Anmerkungen wie bei eindimensionalen Feldern.
Der Speicherverbrauch wird hier sehr schnell sehr groß. Es muss hier zur Berechnung des Speicherverbrauchs jede Dimension multipliziert werden:

Anzahl Bytes pro Element * Dimension1 * Dimension2 *...

Beispiel:
Wie viel Speicher benötigt folgendes Feld:
double x[10][200][50][30][100];

7.3. Initialisierung von Feldern

Bei Variablen kennen wir schon die Möglichkeit einer Initialisierung. Wie sieht es aber bei Feldern aus? Bei Variablen geschieht die Initialisierung durch Zuweisung eines Wertes bei der Deklaration. Genauso funktioniert es auch bei Feldern. Da wir es hier aber sozusagen mit mehreren Variablen zu tun haben, ist es verständlich, dass die Syntax ein wenig anders aussieht.

7.3.1. Initialisierung eindimensionaler Felder

Die Initialisierung eindimensionaler Felder erfolgt, indem die Werte in geschweiften Klammern eingebettet werden, wie z.B.

```
int Nummern[5] = { 1, 5, 3, 7, 9};
```

Wenn in der eckigen Klammer eine Anzahl von Elementen angegeben ist, dürfen nicht mehr Zahlen zum Initialisieren in den geschweiften Klammern stehen. Werden jedoch weniger Zahlen angegeben, so werden die restlichen Elemente automatisch mit 0 gefüllt. So kann z.B. mit der Anweisung

```
int Nummern[5] = {0};
```

erreicht werden, dass alle Elemente des Feldes *Nummern* auf 0 gesetzt werden.
Werden die Felder vollständig initialisiert, d.h. für jedes gewünschte Element wird ein Wert in den geschweiften Klammern angegeben, so kann die Angabe für die Anzahl der Elemente entfallen. So belegt die Deklaration

```
int Nummern[ ] = { 1, 5, 3, 7, 9};
```

ebenfalls ein Feld mit 5 Elementen, da bei der Initialisierung 5 Werte angegeben wurden.

7.3.2. Initialisierung mehrdimensionaler Felder

Die Initialisierung mehrdimensionaler Felder erfolgt analog zu eindimensionalen Feldern. Aus Gründen der Übersichtlichkeit empfiehlt es sich, hier für jede Dimension eine neue „Klammern-Ebene“ auf zumachen. Beispiel für ein 2-dimensionales Feld:

```
int Matrix[3][5] = {
            { 1   2,  3,  4,  5},
            { 6   7,  8,  9, 10},
            {11, 12, 13, 14, 15}
            };
```

So wird jede *Zeile* in eine separate geschweifte Klammen-Ebene gestellt. Die einzelnen *Zeilen* werden durch Kommata getrennt.
Auch hier gilt, dass wenn nicht für alle Elemente ein Wert angegeben wird, werden die restlichen Werte mit 0 initialisiert. So würden z.B. bei folgender Deklaration

```
int Matrix[3][5] = {
            { 1   2,  3,  4,  5},
            { 6   7,  8,  9, 10}
            };
```

alle Elemente der letzten Zeile mit 0 initialisiert.

```
int Matrix[3][5] = {
            { 1 },
            { 6 },
            {11 }
            };
```

Hier werden nur die jeweils ersten Elemente einer Zeile initialisiert. Alle anderen Elemente erhalten den Wert 0.
Bei mehrdimensionalen Feldern kann auf die Angabe der **ersten** Dimension verzichtet werden, wenn alle Elemente initialisiert werden:

```
int Matrix[ ][5] = {
            { 1   2,  3,  4,  5},
            { 6   7,  8,  9, 10},
            {11, 12, 13, 14, 15}
            };
```

Der Compiler errechnet die erste Dimension aus der Anzahl der angegebenen Elemente. Auf die Angabe aller weiteren Dimensionen kann aber nicht verzichtet werden!

7.4. Zugriff auf Elemente

Wir haben gelernt, wie man Felder und damit auf einen Schlag viele einzelne Variablen vom gleichen Typ anlegt.

int Zahlen[20];

Doch wie kann man nun auf die einzelnen Elemente des Feldes zugreifen? Einzelne Elemente des Arrays werden durch Angabe des gewünschten Index in eckigen Klammern angesprochen. Das erste Element eines Arrays hat immer den Index 0! Wollen wir also nun das dritte Element aus dem zuvor angelegten Array ansprechen, so wäre dies:

Zahlen[2]

Der Index reicht dabei immer von 0 bis **n-1**, in unserem Fall also von 0 bis 19.

> Beachten Sie, dass jedes Feld mit der Elementnummer 0 und nicht mit 1 beginnt. Die größte Zahl für den Index beträgt n-1 und nicht n. Dies führt, besonders bei Anfängern, häufig zu Fehlern und evtl. Abstürzen von Programmen, da nicht beachtet wird, dass es das Element mit Index n nicht mehr gibt.

Beispiel:
Wir wollen ein Programm erstellen, das zunächst in einem Feld alle Kleinbuchstaben speichert. Danach kann der Benutzer eine Nummer eingeben, zu der ihm der entsprechende Kleinbuchstabe am Bildschirm ausgegeben wird.

Listing 7.1: Beispiel zum Zugriff auf Feldelemente

```
#include  <stdio.h>

int  main()
{
    char    buchst[26];
    int     i,  zahl;

    for (i=0 ; i<26 ; i++)  /* Array mit Kleinbuchstaben belegen */
        buchst[i] = 'a'+i;

    while (1)
    {
       printf("Geben Sie eine Zahl zw. 1 und 26 ein (Ende=Zahl 100): ");
       scanf("%d", &zahl);

       if (zahl==100)
          break;
       else if (zahl<1 || zahl>26)
          printf("     ......Falsche Eingabe (Eingabe wiederholen!)....\n");
       else
          printf(" ---> %d. Kleinbuchstabe = %c\n\n", zahl, buchst[zahl-1]);
    }

    printf("-------Programmende------\n");
   return(0);
}
```

Folgendes Programm zeigt den Fehler auf, der bei der Nichtbeachtung des maximalen Index entsteht. Häufig ist dieser Fehler mit einer falschen Formulierung der for-Schleife anzutreffen:

Listing 7.2: Unerlaubter Zugriff auf Feldelemente

```
#include  <stdio.h>

int  main()
{
   int  i;
   int  vor_array=0;
   int  quad[5];
   int  nach_array=0;

   for (i=1; i<=5; i++)
      quad[i] = i*i;

   for (i=1; i<=5; i++)
      printf("%2d * %2d = %3d\n", i, i, quad[i]);

   printf(" ----> vor_array=%d\n", vor_array);
   printf(" ----> nach_array=%d\n", nach_array);
   return(0);
}
```

Mögliche Ausgabe des Programms:

```
1 * 1 =  1
2 * 2 =  4
3 * 3 =  9
4 * 4 = 16
5 * 5 = 25
----> vor_array=0
----> nach_array=25
```

Hier ist zu erkennen, dass durch das Schreiben über die Grenzen hinaus, die Variable *nach_array* überschrieben wurde. Dieses Überschreiben fremden Speicherplatzes kann vor allen Dingen in größeren Programmen zu schwer auffindbaren Fehlern führen.

Analog zu den eindimensionalen Feldern erfolgt der Zugriff auf Elemente bei den mehrdimensionalen Feldern. Hier müssen allerdings alle Dimensionen jeweils in eckige Klammern gesetzt werden:

```
...
int Tabelle[5][2];
...
Tabelle[0][0] = 5; /* Zugriff auf die Elemente */
Tabelle[0][1] = 1;
Tabelle[4][0] = 9;
Tabelle[4][1] = Tabelle[3][2];
```

7.5. Zeichenketten

Bisher haben wir nur die Möglichkeit, einzelne Zeichen oder Zahlen in Variablen abzuspeichern. Was ist aber nun zu tun, wenn nicht nur ein Zeichen, sondern eine ganze Zeichenkette (engl. *string*) zusammenhängend abzuspeichern ist.

7.5.1. Besonderheiten von C-Zeichenketten

Wir kennen bereits eine Zeichenketten-Konstante, wie z.B. *ïch lerne C"*. Sie ist eine Folge von beliebig vielen Zeichen, die in doppelte Hochkommas eingeschlossen sind.
Eine solche String-Konstante kann als **char**-Feld aufgefasst werden:

[0]	i
[1]	c
[2]	h
[3]	
[4]	l
[5]	e
[6]	r
[7]	n
[8]	e
[9]	
[10]	C
[11]	\0

Der C-Compiler markiert das Ende eines solchen char-Feldes mit dem Null-Zeichen \0 (ein Byte, das nur 0-Bits enthält = Wert 0), damit beim Programmablauf das Ende der Zeichenkette erkannt werden kann. Deshalb ist die Länge eines solchen Arrays um 1 größer als die Anzahl der eigentlichen Zeichen.

Beispiele:

```
"ich lerne C"  12 Bytes
"Hallo"         6 Bytes
""              1 Byte (Leerstring belegt 1 Byte: \0)
```

Die Anführungszeichen gehören nicht zur Zeichenkette, sondern sind nur die Begrenzer.

7.5.2. Deklaration von Zeichenketten (Strings)

Wie oben erwähnt, werden die Zeichenketten in C durch char-Felder dargestellt. Entsprechend verläuft auf die Deklaration von String-Variablen analog zu der von eindimensionalen Feldern:

Listing 7.3: Verwendung von Zeichenketten

```
#include <stdio.h>

int main() {
    char vorname[255], nachname[255];

    printf("Geben Sie Ihren Vornamen ein: ");
    scanf("%s", vorname);  // Bei Feldern kein &!
    printf("Geben Sie Ihren Nachnamen ein: ");
    scanf("%s", nachname); // Bei Feldern kein &!
    printf("\nHallo %s %s!\n", vorname, nachname);
}
```

Eine Besonderheit gilt hier bei der Eingabe über scanf():
Verwenden wir für alle anderen Variablen immer den Adressoperator &, so ist das bei Strings oder allgemein bei Feldern nicht erlaubt. Es handelt sich bei dem Feldnamen bereits um einen Zeiger, der die eigentliche Adresse des Zielspeichers (hier des Feldes) beinhaltet. Mehr dazu im Kapitel Zeiger.

7.5.3. Initialisierung von Zeichenketten

Es ist zwischen den reinen char-Zeichen, char-Feldern und Strings zu unterscheiden. Wie oben dargestellt, haben die Strings noch zusätzlich eine Endekennung in Form des Null-Terminierungszeichens. Bei reinen Zeichen oder char-Feldern fehlt dieses Terminierungszeichen. So ist z.B. 'A' nicht das gleiche wie "A". Bei 'A' handelt es sich um ein einzelnes Zeichen, bei "A" um eine Zeichenkette, die im Rechner als Feld mit 2 Elementen dargestellt wird.
Bei der Initialisierung von Feldern, wird festgelegt, ob der Inhalt eine Zeichenkette (String) ist oder ob es sich um ein char-Feld mit einzelnen Zeichen handelt:

```
char gruss[]      = "Hallo";           // String, 6 Bytes gross
char Nachname[11] = "Mustermann";      // String, Mustermann hat 10 Zeichen+0
char artikel[]    = {'d','e',',','\0'};// String, da letztes Zeichen = 0
char vorname[30]  = "Hans";            // String, vorname[4] ist Endezeichen
char test1[]      = {'d','a','s'};     // char-Feld, da Endezeichen fehlt
```

```
char test2[4]     = {'d','a','s'};      // String, da letztes Element nicht
                                        // initialisiert und damit = 0
```

Folgende Regel gelten in Zusammenhang mit Zeichenketten:

- Die zur Verfügung stehende Stringlänge ist um 1 kleiner als die Größe des char-Feldes, da das Endezeichen immer mit abgespeichert werden muss!
- Ein Byte mit dem Wert 0 kann nicht innerhalb einer Zeichenkette vorkommen. Es würde sonst das Ende der Zeichenkette eine Stelle vor diesem Wert bedeuten. Dieses Verhalten kann aber manchmal, z.B. zur Kürzung von Zeichenketten, erwünscht sein.
- Eine direkte Zuweisung von Zeichenketten untereinander über den Zuweisungsoperator ist nicht möglich. Dafür stehen umfangreiche Hilfsfunktionen zur Verfügung (siehe Kapitel 7.5.5)
- Werden Zeichenketten direkt initialisiert und die Größe des Feldes nicht angegeben (nur []), so wird für das Endezeichen automatisch Speicherplatz reserviert.
- Ist die Größe des Feldes bei der Initialisierung angegeben, so muss das Endezeichen zur Länge der Zeichenkette hinzugerechnet werden.

7.5.4. Eingabe von Zeichenketten über die Tastatur

Wir kennen aus dem vorigen Kapitel bereits die Methoden zur Eingabe über scanf(). Beim Einlesen mit scanf() ist aber Vorsicht geboten, da scanf() bei der Formatangabe %s eine Zeichenkette nur bis zum ersten Zwischenraumzeichen (Leer- oder Tabulatorzeichen) einliest.
Am folgenden Programm kann dies getestet werden, wenn als Name sowohl Vor- als auch Nachname mit Leerzeichen getrennt eingegeben werden. Der Nachnamen wird dann nicht mehr eingelesen.

Listing 7.4: Eingabe einer Zeichenkette mit scanf()

```
#include  <stdio.h>

int  main()
{
   char   name[50];

   printf("Gib deinen Namen ein: ");
   scanf("%s", name);

   printf("\n   Dein Name ist also %s\n", name);
   return(0);
}
```

Es gibt verschiedene Möglichkeiten, auch scanf() dazu zu bewegen, das Leerzeichen mit einzulesen. Eine einfachere Möglichkeit zur Eingabe von Zeichenketten mit beliebigen Zwischenraumzeichen bietet da die Funktion gets():

char *gets(char *buffer);

Wie bei scanf() erwartet gets() die Adresse des char-Feldes (also den Namen des Feldes). gets() liest dabei immer eine ganze Zeile (bis Zeilenvorschub \n bzw. Eingabe von ENTER) ein und speichert die gelesenen Zeichen in das char-Feld, wobei das \n durch das Endezeichen \0 ersetzt wird. Die Verwendung von gets() empfiehlt sich immer dann, wenn ganze Zeilen einzulesen sind.

Listing 7.5: Eingabe einer Zeichenkette mit gets()

```
#include  <stdio.h>

int  main()
{
   char   name[50];

   printf("Gib deinen Namen ein: ");
   gets(name);

   printf("\n   Dein Name ist also ");
   printf("%s", name);
   return(0);
}
```

7.5.5. Die Headerdatei <string.h>

Die Headerdatei **string.h** enthält eine Vielzahl von Funktionen, die für den Umgang mit Strings konzipiert wurden. Nachfolgend werden einige dieser in ANSI-C standardisierten Funktionen vorgestellt. Es gibt noch einige mehr, die bei Bedarf in der Hilfe des C-Compilers nachgeschlagen werden können. Die Aufzählung aller Funktionen würde den Umfang dieses Skripts sprengen.

strcpy - Kopieren einer Zeichenkette

char *strcpy(char *ziel, const char *quelle);

Kopiert alle Zeichen der Zeichenkette *quelle* (inkl. Nullterminierungszeichen) in den Speicherbereich auf den *ziel* zeigt. Die Zeichenketten dürfen sich nicht überlappen und *ziel* muss groß genug sein um *quelle* aufzunehmen.
Rückgabe: Zeiger auf *ziel*

Listing 7.6: Beispiel zu strcpy()

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

int  main()
{
   char  gruss[] = "Hallo, wie gehts";
```

```
    char   gruss1[100];

    strcpy(gruss1, gruss);
    printf("%s\n", gruss1);

    return(0);
}
```

strncpy - Kopieren von n Zeichen einer Zeichenkette

char *strncpy(char *ziel, const char *quelle, size_t n);

Kopiert die n-ersten Zeichen der Zeichenkette *quelle* in den Speicherbereich auf den *ziel* zeigt. Die Zeichenketten dürfen sich nicht überlappen und *ziel* muss groß genug sein um *quelle* aufzunehmen.
Rückgabe: Zeiger auf *ziel*

Listing 7.7: Beispiel zu strncpy()

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

int main()
{
    char   string1[] = "Das ist der String1",
           string2[] = "String2";    /* Laenge = 7 */

    strncpy(string1, string2, 7);    /* n gleich der Laenge von string2 */
    printf("%s\n", string1);

    return(0);
}
```

strcat - Anhängen einer Zeichenkette an eine Andere

char *strcat(char *ziel, const char *quelle);

Kopiert alle Zeichen der Zeichenkette *quelle* (inkl. Nullterminierungszeichen) an das Ende des Speicherbereichs auf den *ziel* zeigt. Das Terminierungszeichen von *ziel* wird dabei überschrieben. Die Zeichenketten dürfen sich nicht überlappen und *ziel* muss groß genug sein um *quelle* zusätzlich aufzunehmen.
Rückgabe: Zeiger auf *ziel*

Listing 7.8: Beispiel zu strcat()

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

int main()
{
    char   string1[100] = "Zeichen",
           string2[] = "ketten";

    strcat (string1, string2);
    printf("%s\n", string1);
    strcat(string1, " sind char-Array");
    printf("%s\n", string1);
```

```
    return(0);
}
```

strncat - Anhängen von n Zeichen einer Zeichenkette an eine Andere

char *strncat(char *ziel, const char *quelle, size_t n);

Kopiert die n-ersten Zeichen der Zeichenkette *quelle* an das Ende des Speicherbereichs auf den *ziel* zeigt. Das Terminierungszeichen von *ziel* wird dabei überschrieben. Die Zeichenketten dürfen sich nicht überlappen und *ziel* muss groß genug sein um *quelle* zusätzlich aufzunehmen.
Rückgabe: Zeiger auf *ziel*

strcmp - Vergleich zweier Zeichenketten

int strcmp(const char *zkette1, const char *zkette2);

Vergleicht jedes Zeichen von *zkette1* mit dem Zeichen an der gleichen Position von *zkette2*. Der Vergleich wird solange fortgeführt, bis entweder zwei unterschiedliche Zeichen gefunden wurden oder das Ende der Zeichenkette erreicht wurde. Achtung: Groß/Kleinschreibung wird unterschieden!!
Rückgabe: Die Funktion hat drei unterschiedliche Rückgabewerte:

- <0: *zkette1* < *zkette2*
- 0: *zkette1* == *zkette2*
- >0: *zkette1* > *zkette2*

strncmp - Vergleich von n Zeichen zweier Zeichenketten

int strncmp(const char *zkette1, sonst char *zkette2, size_t n);

Vergleicht die n-ersten Zeichen von *zkette1* mit dem Zeichen an der gleichen Position von *zkette2*. Der Vergleich wird solange fortgeführt, bis entweder zwei unterschiedliche Zeichen gefunden wurden oder das Ende der Zeichenkette erreicht wurde. Achtung: Groß/Kleinschreibung wird unterschieden!!
Rückgabe: Die Funktion hat drei unterschiedliche Rückgabewerte:

- <0: *zkette1* < *zkette2*
- 0: *zkette1* == *zkette2*
- >0: *zkette1* > *zkette2*

strlen - Ermitteln der Länge Zeichenketten

size_t strlen(const char *zkette);

Ermittelt die Länge der übergebenen Zeichenkette. Das Nullterminierungszeichen wird nicht mitgezählt.

Listing 7.9: Beispiel zu strlen()

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

int main()
{
    char string[] = "Winter";
    char string2[] = ""; /* Leerstring */

    printf(" Laenge von \"%s\": %d\n", string, strlen1(string));
    printf(" Laenge von \"%s\": %d\n", string2, strlen1(string2));

    return(0);
}
```

strchr - Suchen eines Zeichens in einer Zeichenkette

char *strchr(const char *zkette, int zeichen);

Sucht nach dem ersten Vorkommen von *zeichen* in der Zeichenkette *zkette*. Das Nullterminierungszeichen am Ende der Zeichenkette ist in die Suche eingeschlossen.
Rückgabe: Zeiger auf *zeichen* wenn es gefunden wurde, ansonsten einen NULL-Zeiger.

strrchr - Rückwärtiges Suchen eines Zeichens in einer Zeichenkette

char *strrchr(const char *zkette, int zeichen);

Sucht nach dem letzten Vorkommen von *zeichen* in der Zeichenkette *zkette*. Das Nullterminierungszeichen am Ende der Zeichenkette ist in die Suche eingeschlossen.
Rückgabe: Zeiger auf *zeichen* wenn es gefunden wurde, ansonsten einen NULL-Zeiger.

strstr - Suchen einer Zeichenkette in einer Zeichenkette

char *strstr(const char *kette1, const char *kette2);

Sucht nach dem ersten Vorkommen von *kette1* in der Zeichenkette *kette2*.
Rückgabe: Zeiger auf *zeichen* wenn es gefunden wurde, ansonsten einen NULL-Zeiger.

Umwandeln einer Zeichenkette in einen numerischen Wert

Für die Umwandlung existieren drei Funktionen:

double atof(const char *zkette);
int atoi(const char *zkette);
long atol(const char *zkette);

Die Funktionen wandeln eine Zeichenkette in den entsprechenden Datentyp um. Führende Leerzeichen und Tabulatoren werden übersprungen und die Umwandlung bei dem ersten Zeichen, das nicht mehr als Ziffer erkannt wird, abgebrochen.

Eine weitere Funktion sscanf() arbeitet analog zur schon bekannten Funktion scanf(), nur dass die Eingabe jetzt nicht über Tastatur sondern durch einen Zeichenkette erfolgt.

int sscanf(const char *puffer, const char *format, ...);

Liest formatierte Daten aus einem String ein. Sie verwendet die Formatzeichenkette, um die eingelesenen Daten in die richtigen Typen umzuwandeln.
Rückgabe: Gibt die Anzahl der eingelesenen und umgewandelten Felder zurück. Im Fehlerfall EOF.

Umwandeln von numerischen Werten in Zeichenketten

int sprintf(char *zkette, const char *format, ...);

Die Funktion nimmt eine formatierte Ausgabe in eine Zeichenkette vor. Die Funktionsweise entspricht der Funktion printf(), jedoch muss zusätzlich eine Zeichenkette übergeben werden, in welche der Text ausgegeben wird.
Rückgabe: Anzahl der ausgegebenen Zeichen ohne Nullterminierungszeichen. Im Fehlerfall EOF.

8. Funktionen

Unterprogramme in C werden Funktionen genannt. Die Verwendung von Funktionen verbessert die Struktur eines Programmes. Zusätzlich ermöglichen Funktionen das Erstellen von C-Programmen in so genannten Modulen, d.h. der C-Code kann über mehrere Quellcodedateien verteilt werden. Der C-Standard definiert bestimmte Funktionen, die zum Sprachumfang der Programmiersprache gehören (z.B. *printf*, *strcmp*, usw). Die Benutzung von Funktionen in einem C Programm wird durch Inkludieren der zugehörigen Header-Datei (siehe 8.2.1) ermöglicht. Der Zugriff auf C-Standard Funktionen und auf eigene Funktionen unterscheidet sich dabei nicht.

8.1. Aufbau

Eine Funktion in C wird wie folgt definiert:

```
Rückgabetyp Funktionsname(typ1 name1, typ2 name2, ..., typN nameN)
{
  Funktionscode;

  return(Wert vom Typ Rückgabetyp);   // Kann bei void entfallen
}
```

Für den Funktionsnamen gelten die selben Einschränkungen wie auch bei Variablennamen. Ein Funktion hat entweder genau einen oder keinen Rückgabewert. Soll eine Funktion keinen Rückgabewert besitzen, so ist als Rückgabetyp **void** anzugeben. Wenn ein Rückgabewert existiert ist dieser beim Verlassen der Funktion mit dem Befehl **return(Rückgabewert)** zu setzen (**return** kann an jeder beliebigen Stelle der Funktion stehen. Die Funktion wird dann an dieser Stelle verlassen).

Ein C-Programm kann aus einer oder auch mehreren solcher Funktionen bestehen. Eine Funktion mit Namen *main()* muss das Konsolenprogramm immer enthalten, da diese Funktion den Programmanfang kennzeichnet.

8.1.1. Parameter

Eine Funktion kann beliebig viele Parameter bearbeiten. Wie viele Parameter ein Funktion bearbeitet, wird bei ihrer Definition festgelegt. Die Liste der Parameter steht in run-

den Klammern hinter dem Funktionsname und ist eine Liste von Typ-Variablennamen-Paaren, getrennt durch Kommata.
Funktionen können genau einen oder keinen Rückgabewert besitzen.
Wir haben bereits ein Programm zur Berechnung des GGT (Größter gemeinsamer Teiler) von 2 Zahlen kennen gelernt. Das folgende Beispiel zeigt das gleiche Programm wobei hier die Berechnung des GGT als eigene Funktion gefasst ist:

Listing 8.1: Berechnung des GGT innerhalb einer Funktion

```
#include <stdio.h>

// Prototyp fuer GGT Funktion
int ggt(int a, int b);

int main()
{
  int a, b;

  // Zahlen vom Benutzer einlesen:
  printf("Bitte 2 Zahlen zur Berechnung des GGT eingeben\n");
  printf("Zahl 1: ");
  scanf("%d", &a);
  printf("Zahl 2: ");
  scanf("%d", &b);

  // Ausgabe Ergebnis:
  printf("GGT von %d und %d ist %d\n", a, b, ggt(a,b));

  return 0;
}


// Funktion zur Berechnung des GGT von zwei Zahlen
int ggt(int a, int b)
{

  while(a != b)          // Algorithmus zur GGT Berechnung
  {
    if(a > b) a -= b;
    else      b -= a;
  }

  return a;              // Rueckgabe des Ergebnis
}
```

8.2. Funktionsprototypen

Funktionen müssen im Quellcode vor ihrer Benutzung entweder definiert oder zumindest deklariert werden. D.h., befindet sich eine Funktion und ihr aufrufender Programmcode in der selben Quellcodedatei, so muss der Funktionscode entweder **vor** dem aufrufenden Code stehen, oder ein Prototyp der Funktion muss erstellt werden. Der Prototyp einer Funktion besteht aus dem Funktionskopf (Rückgabewert, Funktionname und Parameterliste) mit einem abschließenden Semikolon. Durch Angabe eines Prototypen wird dem Compiler die Funktion bekannt gemacht und somit ihre Benutzung erlaubt (Der Compiler arbeitet die Quelldatei sequentiell vom Anfang bis zum Ende ab).

8.2.1. Header-Dateien

Headerdateien in C enthalten typischerweise Prototypen von Funktionen und Definitionen von Datentypen (siehe auch Kapitel 9). Auch eine kurze Beschreibung der Funktionen kann, bzw. sollte enthalten sein. Beispielsweise enthält die Headerdatei „stdio.h“ Prototypen für Funktionen wie *printf()*, *scanf()* und viele andere. Bei der Erstellung größerer Projekte in C wird der Programmcode meist in mehrere Quelldateien unterteilt (siehe Modulare Programmierung). Dabei wird normalerweise zu jeder Quelldatei eine zugehörige Header-Datei angelegt. Header-Dateien sind immer nötig, wenn

- Funktionen aus den C-Standard Bibliotheken benutzt werden (Also eigentlich immer)
- Funktionen aus zugekauften Bibliotheken verwendet werden (Man hat dabei im Normalfall den Quellcode der Funktionen nicht zur Verfügung)
  Beispiel: GUI Bibliotheken zur Windowsprogrammierung(MFC, DirectX,...)
- Eigene Projekte mehr als eine Quellcodedatei umfassen

8.3. Gültigkeit von Variablen

Variablen haben je nach Ort ihrer Definition einen bestimmten Gültigkeitsbereich („Lebensdauer“).

8.3.1. Globale Variablen

Globale Variablen werden im Programmcode außerhalb jeglicher Funktion deklariert. Auf globale Variablen kann von jeder Funktion aus zugegriffen werden. Sie sind, wie es der Name schon sagt, global gültig. Wird in einer Funktion jedoch eine Variable gleichen Namens definiert, so wird bei einem Zugriff innerhalb der Funktion immer die funktionslokale Variable benutzt. Wenn eine globale Variable einen initialen Wert enthalten soll, so kann sie bei der Deklaration auch gleich initialisiert werden.

Gültigkeit	Zur gesamten Programmlaufzeit
Speicherort	Datensegment
Vorteile	- Einfache Handhabung bei kleinen Programmen - spart Parameterübergaben an Funktionen
Nachteile	- Einsatz wird bei größeren Programmen schnell unübersichtlich - Überdeckung mit lokalen Variablen in Funktionen - Fehlersuche wird schwieriger (Nicht mehr so einfach innerhalb einer Funktion lokalisierbar)

8.3.2. Lokale Variablen

Lokale Variablen werden innerhalb einer Funktion deklariert. Dazu gehören die Übergabeparameter an die Funktion selbst, sowie eventuell weitere deklarierte Variablen innerhalb der Funktion. Lokale Variablen sind nur innerhalb der Funktion gültig und werden nach Verlassen der Funktion gelöscht. Wird innerhalb einer Funktion eine Variable mit gleichem Namen deklariert wie auch eine globale Variable benannt wurde, so ist kein Zugriff auf die globale Variable mehr möglich.

Gültigkeit	Zur Laufzeit der Funktion
Speicherort	Stack
Vorteile	- Variablen die nur innerhalb der Funktion benötigt werden sind nicht außerhalb sichtbar - Fehlerhaftes Beschreiben der Variablen von anderen Funktionen aus ist nicht möglich - Saubere Kapselung der Daten und zugehörigem Funktionscode
Nachteile	- Funktion muss eventuell mehr Parameter übergeben bekommen, als wenn sie auf globale Variable zugreifen würde

Listing 8.2: Berechnung der Potenz in einer Funktion

```
#include <stdio.h>

float hoch(float a, int b)   /* Funktionsname(Parameter-Deklarationen)  */
{
    int     zaehler;            /*   Lokale Deklarationen      */
    float   rueckgabe_wert=1;   /*                             */

    if (b<0)
    {                                         /*                       */
       a = 1/a;                               /*                       */
       b = -b;                                /*                       */
    }                                         /*    Funktions-         */
    for (zaehler=1 ; zaehler<=b ; ++zaehler)  /*      Anweisungen      */
       rueckgabe_wert *= a;                   /*                       */
                                              /*                       */
    return(rueckgabe_wert);                   /*                       */
}


int  main()    /*   Funktionsname(leere Liste von formalen Parametern)   */
{
    int     exponent;        /*   Lokale Deklarationen             */
    float   zahl, ergeb;     /*                                    */

    printf("Dieses␣Programm␣berechnet␣x␣hoch␣y\n\n\n");

    printf("x:␣");
    scanf("%f", &zahl);
    printf("y:␣");
    scanf("%d", &exponent);
    printf("\n");

    ergeb = hoch(zahl, exponent);  /*  Aufruf der Funktion   hoch        */
    printf("....%g␣hoch␣%d␣=␣%g\n", zahl, exponent, ergeb);

    printf("....%g␣hoch␣%d␣=␣%g\n", zahl, exponent, hoch(zahl, exponent));
    return(0);
}
```

8.3.3. Static Variablen

Die so genannten static Variablen sind lokale Variablen innerhalb einer Funktion, die jedoch nur einmal existieren und auch nach Ablauf einer Funktion nicht gelöscht werden. Wenn static Variablen bei ihrer Deklaration auch definiert werden, so geschieht diese Initialisierung nur einmal beim Programmstart und nicht bei jedem erneuten Funktionsaufruf. Listing 8.3 verdeutlicht die Benutzung einer static Variablen.

Gültigkeit	Zur gesamten Programmlaufzeit
Speicherort	Datensegment
Vorteile	- Benutzung, wenn Inhalt über mehrere Funktionsaufrufe benötigt wird - Ähnliches Verhalten wie globale Variable, jedoch nur innerhalb der Funktion sichtbar

Listing 8.3: Beispiel einer static Variable

```
#include <stdio.h>

void f()
{
  static int anzahl = 0;
  anzahl++;
  printf("Funktion f wurde %d mal aufgerufen\n", anzahl);
}

int main()
{
  f();f();f();f();
  return 0;
}
```

8.4. Aufruf von Funktionen

8.4.1. Call-by-value

Bei dieser Form der Parameterübergabe wird eine Kopie der übergebenen Parameter erzeugt. Die lokalen Variablen innerhalb des Funktionslaufes arbeiten somit nur mit der Kopie der Eingabedaten und können die Daten in der aufrufenden Funktion nicht beeinflussen. Diese Verhalten haben wir bei fast alle unseren bisher verwendeten Funktionen verwendet. Ausnahmen davon waren z.B. Funktionen wie scanf() oder die String-Funktionen. Listing 8.4 verdeutlicht die Parameterübergabe mit call-by-value.

Listing 8.4: Parameterübergabe call-by-value

```
#include <stdio.h>

float neurad(int i, float rad)
{
    i = 0;
    rad = rad * 2;
    printf("In der Funktion: I=%2d RADIUS=%6.2f\n",i,radius);
    return (rad);
}

int main()      /* Beispiel fuer Parameteruebergabe und die
               Aenderbarkeit von Parametern in der Funktion */
{
        float radius, r;
        int i;

        i = 10;
        radius = 5.23;

        printf("Vor Aufruf: I=%2d RADIUS=%6.2f\n",i,radius);

        r = neurad(i,radius);

        printf("Nach dem Aufruf: I=%2d RADIUS=%6.2f\n",i,radius);
        printf("Aber Funktionswert: R=%6.2f\n",r);

        return(0);
}
```

8.4.2. Übergabe mit C++ Referenzen

Die Übergabe von Parametern als Referenzen wird in der Deklaration der Funktion festgelegt. Es kann für jeden Parameter einzeln bestimmt werden ob er als Referenz oder mittels „call-by-value“ übergeben werden soll. Um einen Parameter für die Übergabe als Referenz zu bestimmen wird der &-Operator benutzt. Die folgende Funktion tauscht den Inhalt von 2 Integervariablen des aufrufenden Programms:

```
void tausche(int &a, int &b)
{
  int temp;
  temp = a;
  a = b;
  b = temp;
}
```

Im Gegensatz zu Call-by-value wird dabei keine Kopie der Übergabeparameter angelegt. Es existiert auch keine zugehörige lokale Variable innerhalb der Funktion. Stattdessen arbeitet die Funktion direkt mit den Variablen der aufrufenden Funktion und kann diese daher verändern. Solch ein Aufruf wird immer dann verwendet, wenn ein einzelner Rückgabewert, den eine Funktion immer liefern kann, nicht mehr ausreicht. Listing 8.5 verdeutlicht nochmals den Unterschied zu „call-by-value“.

Listing 8.5: Parameterübergabe an Funktionen

```
#include <stdio.h>

void tausche_by_value(int a, int b)
{
  int temp;
  temp = a; a = b; b = temp;    // Dreieckstausch
}

void tausche_by_cppref(int &a, int &b)
{
  int temp;
  temp = a; a = b; b = temp;    // Dreieckstausch
}

int main()
{
  int x, y;

  x = 1; y = 2;
  // Tausch mit call by value
  tausche_by_value(x,y);
  printf("Call by value    : x=%d y=%d\n",x,y);

  x = 1; y = 2;
  // Tausch mit call by cpp ref
  tausche_by_cppref(x,y);
  printf("Call by C++ Ref  : x=%d y=%d\n",x,y);

  return 0;
}
```

8.4.3. Auswertung der Parameter

Aus Sicht des aufrufenden Programms können an Funktionen nicht nur direkt Variablen übergeben werden, sondern generell Ausdrücke. Dabei kann es sich um das Ergebnis einer anderen Funktion, einer Berechnung, usw. handeln. Es ist zu beachten, dass es keine definierte Reihenfolge bei der Auswertung der zu übergebenden Ausdrücke an ein Funktion gibt. Dies bedeutet, dass ein Programm die Parameter entweder von rechts nach links oder in umgekehrter Reihenfolge an die Funktion übergibt. Werden die Parameter daher beim Funktionsaufruf verändert, so kann das Programm je nach Compiler verschiedene Ergebnisse liefern. Es ist daher immer Vorsicht angebracht, wenn z.B. Variablen beim Aufruf einer Funktion noch verändert werden. Die Übergabe von Ausdrücken die untereinander Abhängigkeiten aufweisen ist zu vermeiden! Listing 8.6 veranschaulicht die Problematik:

Listing 8.6: Problem bei Auswertung der Parameter

```
#include <stdio.h>

int addiere(int a, int b)
{
  return (a+b);
}


int main()
{
  int ergebnis, i, j;

  i = 7;  j = 5;
  ergebnis = addiere(++i, j+i);
  printf("1. Ergebnis: %d\n", ergebnis);


  i = 7;  j = 5;
  ergebnis = addiere(j+i, ++i);
  printf("2. Ergebnis: %d\n", ergebnis);

  return 0;
}
```

Listing 8.6 liefert als Ausgabe (Compiliert mit MS Visual C++):

```
1. Ergebnis: 20
2. Ergebnis: 21
```

Wird das Programm mit einem anderen Compiler gebaut so könnte jedoch genau die umgekehrte Ausgabe erfolgen!

8.5. Modulare Programmierung in C

Unter modularer Programmierung versteht man meist die Unterteilung eines Programms in mehrere Quellcodedateien. Sinnvollerweise enthalten Module logisch zusammengehörigen Funktionen. Ein Modul in C besteht im Normalfall aus einer Quellcodedatei (*dateiname*.c) und der dazugehörigen Headerdatei (*dateiname*.h). Wenn nun ein Modul Funktionen aus einem anderen benutzen möchte so geschieht dies durch Inkludieren der jeweiligen Headerdatei. Der Einsatz von Modulen bringt viele Vorteile:

- Mehrere Quellcodedateien erleichtern die Arbeit mit mehreren Entwicklern
- Bessere Trennung von Funktionsschichten im Programm (z.B. Modul für I/O, Modul für Kernfunktionalität, ...)
- Logische Funktionsblöcke können in Module gekapselt werden
- Bessere Wiederverwendbarkeit (reuse) von Programmteilen
- Austausch von Modulen wird möglich
- Bessere Schnittstellendefinition im Programm

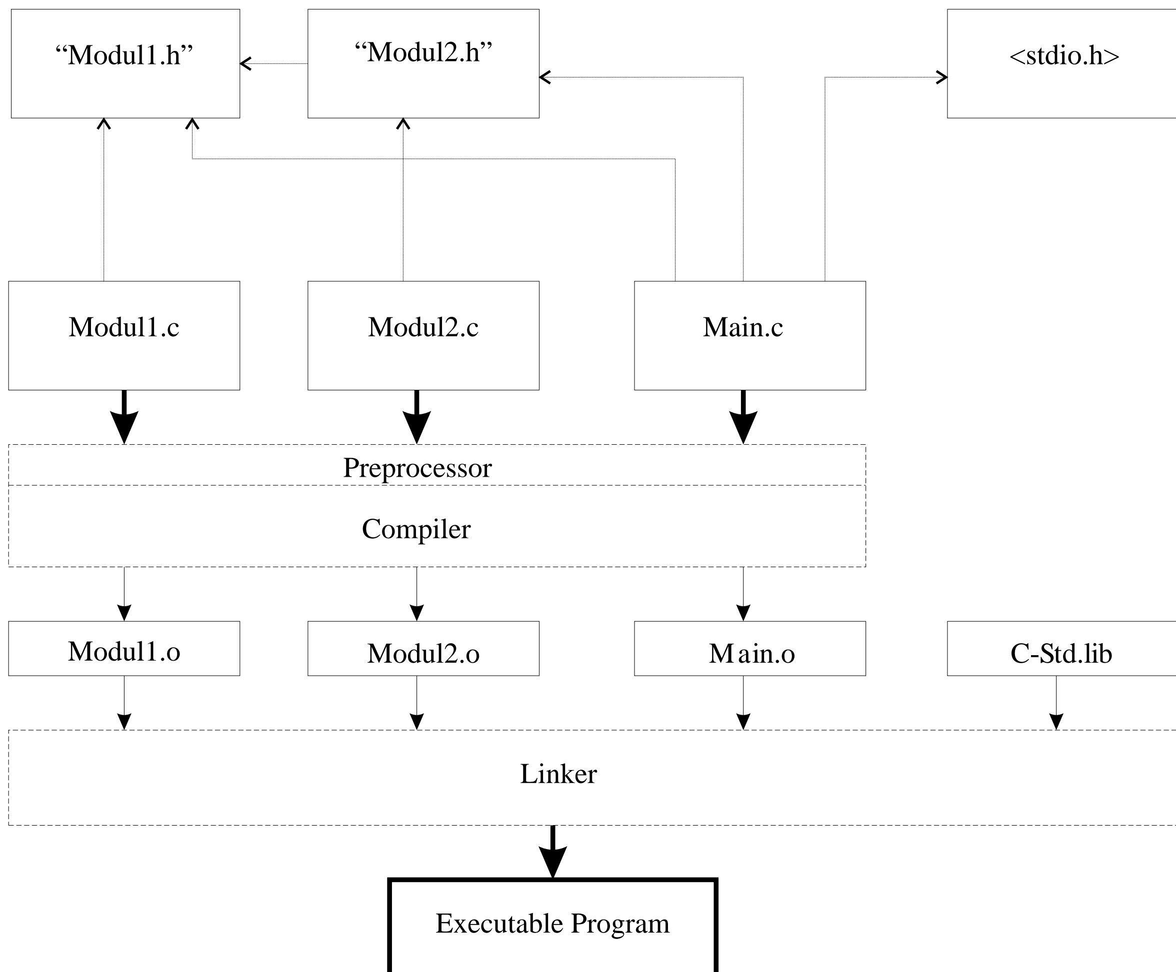


Abbildung 8.1.: Modulare Programmierung

Abbildung 8.1 zeigt die Entstehung eines Programms aus mehreren Modulen.
Die Module werden vom Compiler einzeln in Objektcodedateien (*dateiname*.o) umgewandelt. Der Linker sorgt dann dafür, dass die einzelnen Objektcodedateien zu einem ausführbaren Programm verbunden (gelinkt) werden.

9. Strukturen

Strukturen in C werden verwendet, um mehrere Basisdatentypen oder auch andere Strukturen zu einem neuen Datentyp zu verbinden. Der neu entstehende Datentyp wird meist als Schablone für einen „Datensatz“ verwendet. Beispiel: Ein Punkt im dreidimensionalen Koordinatensystem wird durch seine 3 Koordinaten beschrieben. In einem Programm, das mit Punkten arbeitet wäre es also sinnvoll den Strukturdatentyp “Punkt“ anzulegen, der aus den drei Koordinaten x, y und z besteht.

9.1. Definition

Eine Struktur in C wird wie folgt definiert:

```
struct Strukturname
{
  typ1   name1;
  typ2   name2;
  ...
  typN   nameN;
};
```

Strukturen können beliebige Datentypen enthalten - außer sich selbst. Es ist jedoch möglich den Typ „Zeiger auf die Struktur“ innerhalb der Struktur selbst zu verwenden (Dies wird oft bei der dynamischen Speicherverwaltung benötigt). Strukturen können in C verwendet werden wie jeder andere Datentyp auch. Bei der Deklaration einer Variable eines Strukturtyps muss normalerweise immer das Schlüsselwort **struct** verwendet werden. Dies lässt sich durch geschickte Verwendung von **typedef** umgehen (siehe Beispiel Listing 9.1).gehen (siehe Beispiel Listing 9.1). Hier noch einmal die Regel für die Definition einer Struktur:

- Eine Strukturdefinition beginnt immer mit dem Schlüsselwort **struct**
- Nach dem Schlüsselwort steht der Name der Struktur
- Eine Struktur besteht aus einer Liste von Komponenten bzw. Einzelvariablen. Jede Komponente wird wie üblich mit ihrem Datentyp und Namen deklariert.
- Die Namen der Komponenten müssen innerhalb einer Struktur eindeutig sein

- Die ganze Liste ist in geschweifte Klammern zu setzen
- Jede Deklaration ist mit einem Semikolon abzuschließen
- Eine Struktur darf die eigenen Struktur nicht enthalten, es sein denn als einen Zeiger.

9.2. Darstellung im Datenhierarchie Diagramm

Die folgende Abbildung zeigt die Darstellung von Strukturen als Datenhierarchie. Der linke Teil zeigt die allgemeine Form der Darstellung für beliebige Strukturen. Der rechte Teil entspricht dem Beispiel für den Datentyp „Punkt“. Das Datenhierarchiediagramm eignet sich auch für die Darstellung verschachtelter Strukturen.

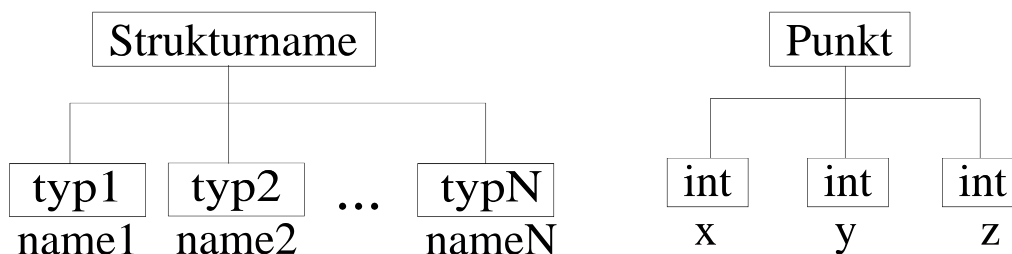


Abbildung 9.1.: Datenhierarchie Diagramm

9.3. Verwendung

9.3.1. Deklarieren und Initialisieren von Strukturvariablen

Eine Strukturvariable wird wie jede andere Variable eines Basisdatentyps deklariert. Bei der Deklaration ist zu beachten, dass die Angabe des Datentyps aus dem Schlüsselwort **struct** und dem Strukturname besteht (Vereinfachung durch **typedef** möglich). Um eine Strukturvariable gleich bei ihrer Deklaration zu definieren wird ähnlich wie auch bei der Definition von Feldern vorgegangen. Eine Liste der Wertinhalte kann angegeben werden:

```
struct student
{
  int   matrikelnr;
  float notenschnitt;
};

struct student student1 = {4711, 1.73};
```

Bei der Angabe der Werte sind die einzelnen Datentypen innerhalb der Struktur zu beachten. Werden weniger Initialwerte angegeben, als die Strukturvariable Komponenten enthält, so wird der Rest mit 0 initialisiert (vergleiche Felder). Es dürfen nicht mehr Werte angegeben werden, als Komponenten in der Strukturvariablen vorhanden sind.
Außerdem ist es möglich, gleich bei der Definition des Strukturdatentyps Variablen davon anzulegen:

```
struct student
{
  int   matrikelnr;
  float notenschnitt;
} student1, student2;

student1.matrikelnr = 4711;
student1.notenschnitt = 1.73;
student2.matrikelnr = 666;
student2.notenschnitt = 2.07;
```

Um den Speicherbedarf einer Variable eines Strukturtypen zu ermitteln kann, wie auch bei den Basisdatentypen, der **sizeof** Operator verwendet werden. Ein Strukturdatentyp benötigt soviel Speicherplatz wie die Summe seiner Elemente.

9.3.2. Zugriff auf Strukturelemente

Um auf eine einzelne Komponente einer Strukturvariablen zuzugreifen, muss der Punktoperator verwendet werden:

```
struct punkt einPunkt;      // Deklaration einer Variable
int a;

einPunkt.x = 5;             // Zuweisung von einzelnen Elementen
einPunkt.y = 7;
einPunkt.z = 3;

a = einPunkt.y;             // Auslesen eines Strukturelements
```

Mit den einzelnen Komponenten sind alle Operationen zugelassen, die für den jeweiligen Komponententyp gültig sind. Listing 9.1 zeigt ein Programmbeispiel zur Verwendung von Strukturen.

Listing 9.1: Benutzung von Strukturvariablen

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

struct punkt            // Definition eines Strukturdatentyps
{
   int x;
   int y;
   int z;
};

typedef struct punkt Punkt;   // Definition eines eigenen Datentyps

int main()
{
  struct punkt a = {1, 1, 1};   // Typangabe ohne typedef
  Punkt        b = {2, 2, 2};   // Neuer Typ mittels typedef

  int dx, dy, dz;
  double       abstand;

  printf("X-Koordinate von Punkt a: %d\n", a.x);
  printf("Y-Koordinate von Punkt a: %d\n", a.y);
  printf("Z-Koordinate von Punkt a: %d\n", a.z);

  printf("X-Koordinate von Punkt b: %d\n", b.x);
  printf("Y-Koordinate von Punkt b: %d\n", b.y);
  printf("Z-Koordinate von Punkt b: %d\n", b.z);

  dx = b.x - a.x;  // Berechne Koordinatenabstaende
  dy = b.y - a.y;
  dz = b.z - a.z;

  // Berechnung des Punktabstands
  abstand = sqrt(dx*dx + dy*dy);
  abstand = sqrt(abstand*abstand + dz*dz);

  printf("Abstand der Punkte a und b: %lf\n", abstand);

  return 0;
}
```

Strukturvariablen gleichen Typs lassen sich auch wie gewohnt einander zuweisen:

```
struct punkt Punkt1, Punkt2;

Punkt1.x = 5;
Punkt1.y = 10;

Punkt2 = Punkt1;
```

Ein Test auf Gleichheit ist aber nicht möglich. Dies kann nur manuell durch einen Vergleich der einzelnen Komponenten erreicht werden.

9.3.3. Felder von Strukturvariablen

Da Strukturvariablen meist als „Datensätze“ verwendet werden ist es oft nötig ganze Felder solcher Variablen anzulegen (z.B. Studentenverwaltungsprogramm). Felder von

Strukturvariablen in C werden analog zu den Feldern der Basisdatentypen deklariert. Der Zugriff auf die Elemente des Feldes erfolgt wie gewohnt mit dem Indexoperator ([ ]):

```
struct student Studenten[100];  // Feld mit 100 Studentendatensätzen

Studenten[0].matrikelnr = 4711;   // Initialisierung 1. Feldelement
Studenten[0].notenschnitt = 1.73;

Studenten[5] = Studenten[0];      // Zuweisung 6. Feldelement

                                  // Zuweisung an Strukturkomponenten
                                  // des 10. Datensatzes:
Studenten[9].matrikelnr = 12345;
Studenten[9].notenschnitt = Studenten[5].notenschnitt;
```

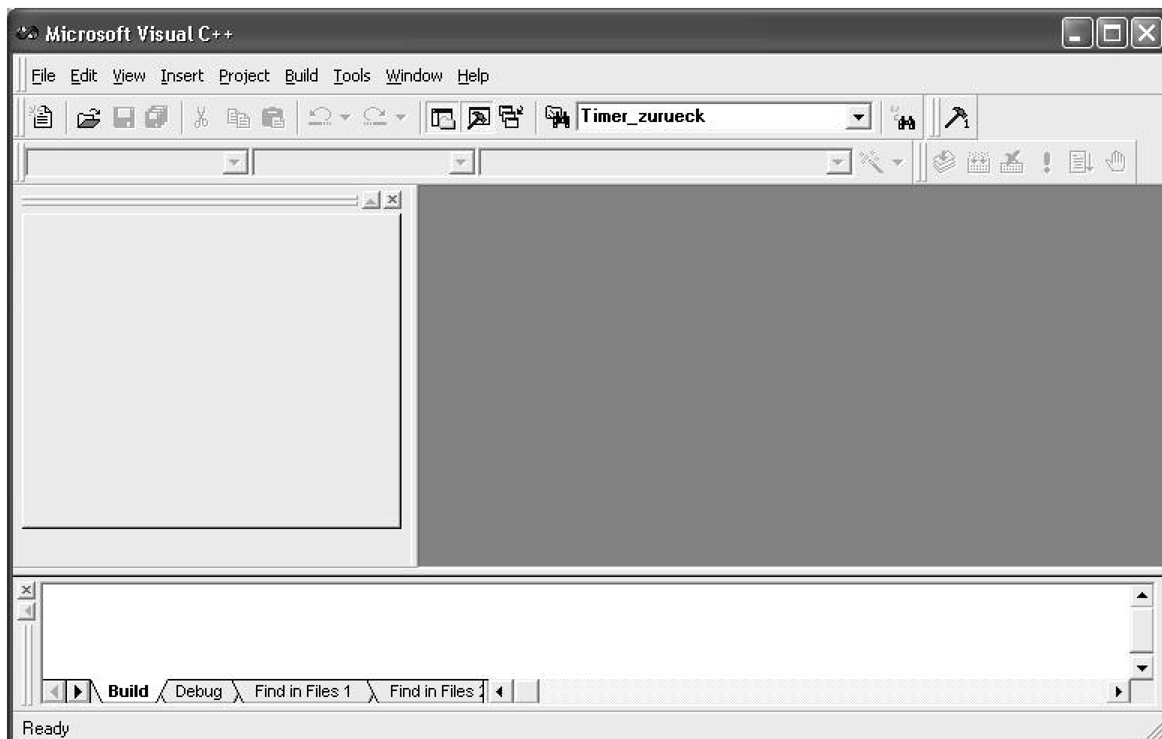
A. ASCII-Tabelle

<table>
<tr><td></td><td></td><td></td><td>b7 b6 b5</td><td>0 0 0</td><td>0 0 1</td><td>0 1 0</td><td>0 1 1</td><td>1 0 0</td><td>1 0 1</td><td>1 1 0</td><td>1 1 1</td></tr>
<tr><td colspan="4">Bits</td><td colspan="2">Steuer-
zeichen</td><td colspan="2">Symbole
Ziffern</td><td colspan="2">Groß-
buchstaben</td><td colspan="2">Klein-
buchstaben</td></tr>
<tr><td>b4</td><td>b3</td><td>b2</td><td>b1</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0
NUL
0 0</td><td>16
DLE
20 10</td><td>32
SP
40 20</td><td>48
0
60 30</td><td>64
@
100 40</td><td>80
P
120 50</td><td>96
‘
140 60</td><td>112
p
160 70</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>1</td><td>1
SOH
1 1</td><td>17
DC1
21 11</td><td>33
!
41 21</td><td>49
1
61 31</td><td>65
A
101 41</td><td>81
Q
121 51</td><td>97
a
141 61</td><td>113
q
161 71</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>0</td><td>2
STX
2 2</td><td>18
DC2
22 12</td><td>34
”
42 22</td><td>50
2
62 32</td><td>66
B
102 42</td><td>82
R
122 52</td><td>98
b
142 62</td><td>114
r
162 72</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>1</td><td>3
ETX
3 3</td><td>19
DC3
23 13</td><td>35
#
43 23</td><td>51
3
63 33</td><td>67
C
103 43</td><td>83
S
123 53</td><td>99
c
143 63</td><td>115
s
163 73</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>4
EOT
4 4</td><td>20
DC4
24 14</td><td>36
$
44 24</td><td>52
4
64 34</td><td>68
D
104 44</td><td>84
T
124 54</td><td>100
d
144 64</td><td>116
t
164 74</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td><td>5
ENQ
5 5</td><td>21
NAK
25 15</td><td>37
%
45 25</td><td>53
5
65 35</td><td>69
E
105 45</td><td>85
U
125 55</td><td>101
e
145 65</td><td>117
u
165 75</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td><td>6
ACK
6 6</td><td>22
SYN
26 16</td><td>38
&
46 26</td><td>54
6
66 36</td><td>70
F
106 46</td><td>86
V
126 56</td><td>102
f
146 66</td><td>118
v
166 76</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td><td>7
BEL
7 7</td><td>23
ETB
27 17</td><td>39
’
47 27</td><td>55
7
67 37</td><td>71
G
107 47</td><td>87
W
127 57</td><td>103
g
147 67</td><td>119
w
167 77</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td>8
BS
10 8</td><td>24
CAN
30 18</td><td>40
(
50 28</td><td>56
8
70 38</td><td>72
H
110 48</td><td>88
X
130 58</td><td>104
h
150 68</td><td>120
x
170 78</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>9
HT
11 9</td><td>25
EM
31 19</td><td>41
)
51 29</td><td>57
9
71 39</td><td>73
I
111 49</td><td>89
Y
131 59</td><td>105
i
151 69</td><td>121
y
171 79</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td><td>10
LF
12 A</td><td>26
SUB
32 1A</td><td>42
*
52 2A</td><td>58
:
72 3A</td><td>74
J
112 4A</td><td>90
Z
132 5A</td><td>106
j
152 6A</td><td>122
z
172 7A</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>1</td><td>11
VT
13 B</td><td>27
ESC
33 1B</td><td>43
+
53 2B</td><td>59
;
73 3B</td><td>75
K
113 4B</td><td>91
[
133 5B</td><td>107
k
153 6B</td><td>123
{
173 7B</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>0</td><td>12
FF
14 C</td><td>28
FS
34 1C</td><td>44
,
54 2C</td><td>60
<
74 3C</td><td>76
L
114 4C</td><td>92
\
134 5C</td><td>108
l
154 6C</td><td>124
|
174 7C</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>13
CR
15 D</td><td>29
GS
35 1D</td><td>45
_
55 2D</td><td>61
=
75 3D</td><td>77
M
115 4D</td><td>93
]
135 5D</td><td>109
m
155 6D</td><td>125
}
175 7D</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td><td>14
SO
16 E</td><td>30
RS
36 1E</td><td>46
.
56 2E</td><td>62
>
76 3E</td><td>78
N
116 4E</td><td>94
∧
136 5E</td><td>110
n
156 6E</td><td>126
∼
176 7E</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>15
SI
17 F</td><td>31
US
37 1F</td><td>47
/
57 2F</td><td>63
?
77 3F</td><td>79
O
117 4F</td><td>95
-
137 5F</td><td>111
o
157 6F</td><td>127
DEL
177 7F</td></tr>
</table>

B. Arbeiten mit Microsoft Visual C++

Wie schon erwähnt, gibt es zur Programmierung in C viele Entwicklungsumgebungen. Die Beispiele in diesem Skript sollten zum größten Teile mit allen ANSI-C Compiler übersetzt und bearbeitet werden. Trotz aller Kompatibilität gibt es bei manchen Herstellern kleine Abweichungen, die einem manchmal das Leben schwer machen.
Alle Beispiele in diesem Skript wurden unter Microsoft Visual C++ entwickelt und getestet. Die Verwendung solch einer mächtigen Entwicklungsumgebung zur Programmierung von einfachsten Programmen mag dem Einen oder Anderen zwar etwas übertrieben vorkommen, aber es hat auch einige Vorteile die hier kurz aufgezählt werden sollen:

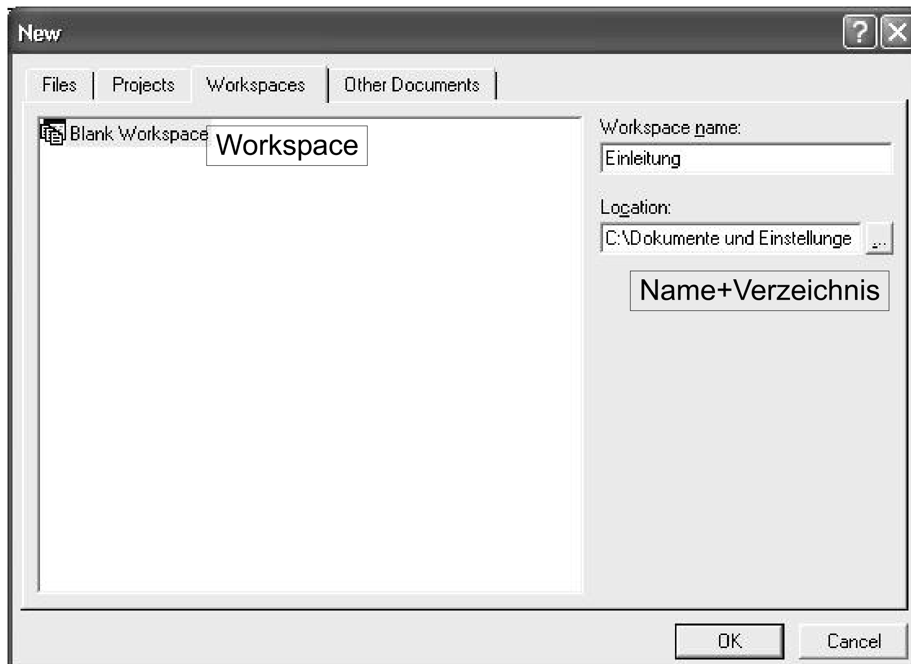
- gute Unterstützung bei der Programmierung durch Syntaxhervorhebungen und automatischen Formatierungen
- einfache Projektverwaltung und Codeübersetzung
- gute Unterstützung bei der Fehlersuche
- ein mächtiger Debugger
- fast schon Industrie-Standard



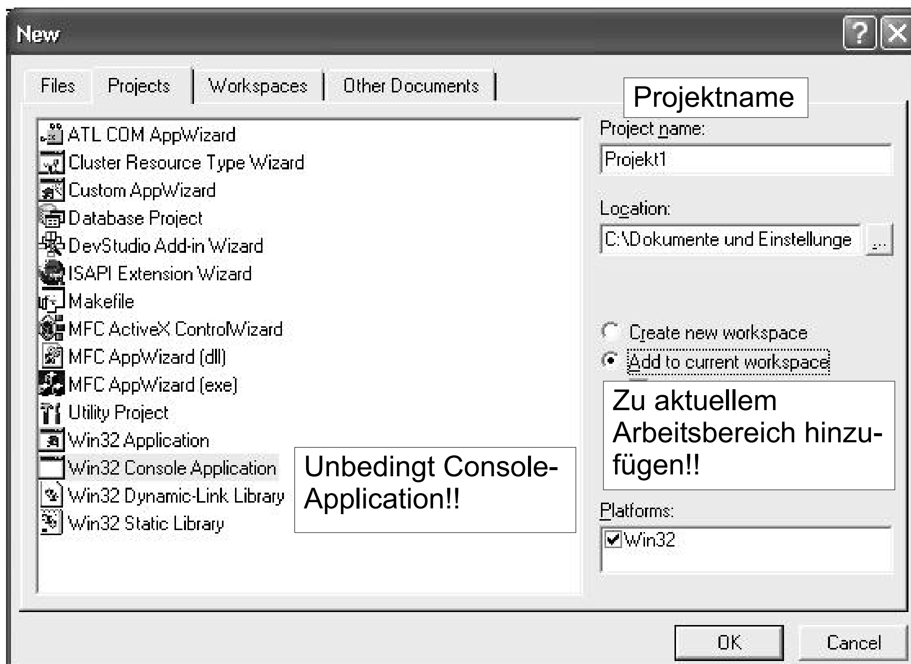
Die Oberfläche gliedert sich in drei Hauptbereiche. Links ist die Projektverwaltung zu sehen, in der alle verwendeten Dateien ersichtlich sind. Das große Fenster rechts, dient zum Editieren des Quellcodes. Unten sind noch verschiedene Fenster zur Anzeige von Fehlermeldungen oder zu Anzeige von Zuständen beim Debuggen.
Eine umfangreiche Hilfe (leider allerdings in Englisch) beschleunigt oft die Programmierung von noch unbekannten Befehlen. Die Hilfe wird aufgerufen, indem die Fehlermeldung oder der unbekannte Befehl markiert wird und dann die **F1-Taste** gedrückt wird.

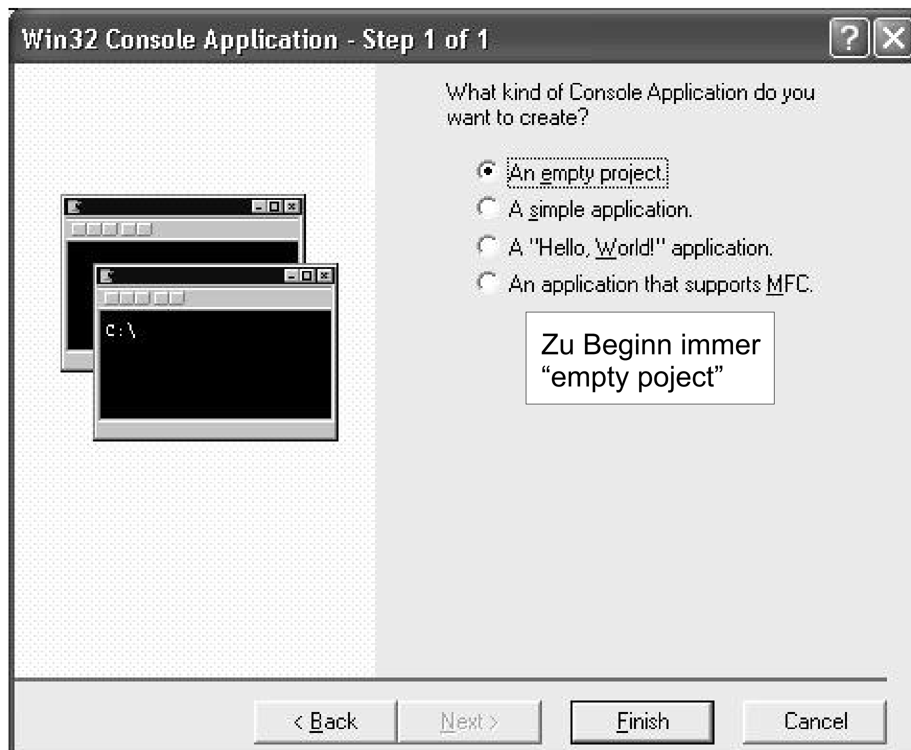
B.1. Programmerstellung

In Visual C++ werden alle Entwicklungen in so genannten Arbeitsbereichen oder auch Workspaces verwaltet. Jeder dieser Arbeitsbereiche kann wiederum viele Projekte oder Programme enthalten. Dies hat den Vorteil, dass z.B. für alle Übungen eines Kapitels nur ein Arbeitsbereich angelegt werden muss. Im folgenden soll kurz aufgezeigt werden, wie ein leerer Arbeitsbereich angelegt wird:

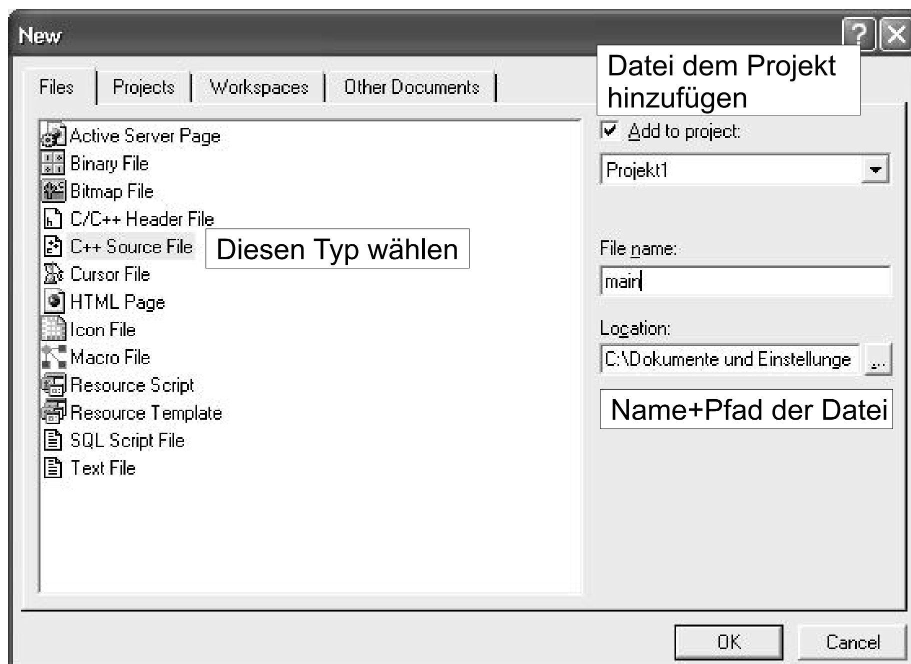


Um ein Programm zu erstellen, muss im Arbeitsbereich ein Projekt angelegt werden. Wir wollen in diesem Skript nur so genannten Konsolenprogramme erzeugen. Das sind die Programme, die ohne Probleme auch unter DOS ablauffähig sind und keine Grafikausgabe besitzen. Die folgenden Abbildungen zeigen, wie solch ein neues Projekt angelegt wird:





Nun haben wir ein Projekt erzeugt. Aber irgendwie fehlt uns immer noch etwas wichtiges. Richtig! Der Teil, in dem wir unser Programm eingeben können. Dazu legen wir in unserem Projekt ein neues C-Sourcefile an. Auch hier werden wir wieder von Visual C++ unterstützt:



Hier noch einige Tricks beim Editieren des Source-Codes:

- Der Editor beherrscht so genanntes „Syntax-Highlighting“. Dies bedeutet dass alle C-Schlüsselwörter in einer anderen Farbe als der „unbekannte“ Code dargestellt werden. Zusätzlich lassen sich noch z.B. alle Arten von Konstanten, Kommentare usw. in einer anderen Farbe darstellen. Die Einstellung erfolgt über das Menu „Tools –> Options –> Format“.

- Die automatischen Einrückungen beim Schreiben des Codes sollten beibehalten werden. So ist sofort die aktuelle Ebene der Schachtelung von einzelnen Blöcken ersichtlich. Am Ende einer Funktion sollte der Cursor wieder in der ersten Spalte stehen, ansonsten wurde entweder ein Semikolon oder eine geschweifte Klammer vergessen.

- Das automatische Layout kann auch durch Markieren eines Teils des Quellcodes und Drücken der Tasten **Alt+F8** erneut durchgeführt werden.

Die verschiedenen Schritte des Compilers und Linkers wurden in Visual C++ automatisiert. Es gibt drei Möglichkeiten ein Programm zu übersetzen. Alle drei Möglichkeiten werden durch die entsprechenden Buttons ausgeführt.

1. Datei nur Compilieren: **Ctrl-F7**. Hier wird nur eine Object-Datei der aktuellen Sourcecode-Datei erzeugt und eine Fehlerüberprüfung durchgeführt. Das Programm kann danach aber noch nicht ausgeführt werden.

2. Programm Compilieren und Linken: **F7**. Es werden alle Dateien übersetzt und zu einem ausführbaren Programm zusammengesetzt. Danach existiert die exe-Datei und kann gestartet werden.

3. Programm Compilieren, Linken und Ausführen: **Ctrl-F5**. Damit werden alle Schritte durchgeführt und das Programm, sofern kein Fehler aufgetreten ist, sofort gestartet.

Visual C++ legt mehrere Dateien in das Verzeichnis des Arbeitsbereichs an. Jedes Projekt bekommt ein eigenes Verzeichnis. Innerhalb des Projektes gibt es noch die beiden Verzeichnisse „Debug „ und „Release“ in welchen die übersetzten Programm und Object-Dateien abgelegt werden. Dort können alle Dateien bis auf die exe-Datei gelöscht werden, da diese beim erneuten Übersetzten wieder erstellt werden. Allerdings dauert ein neuer Übersetzungsvorgang dann etwas länger.
Im Projektverzeichnis liegen die Quellcode-Dateien (*.cpp, *.c, *.h) sowie eine Datei (*.dsp) die die Projekteinstellung beinhaltet. Die Datei (*.plg) enthält eine Log-Datei über den Erstellungsvorgang mit allen Aktionen die durchgeführt wurden.
Im Arbeitsbereich-Verzeichnis liegt die Datei (*.dsw) die die Einstellungen für den Arbeitsbereich enthält.

B.2. Fehlersuche

Treten bei der Übersetzung oder beim Linken Fehler auf, so werden diese im unteren Fenster angezeigt. Durch einen Doppelklick auf eine Fehlermeldung, springt der Editor automatisch an die fehlerhafte Stelle. Oft sind die Meldungen nicht ganz eindeutig, so dass es einer gewissen Übung bedarf, sie richtig zu interpretieren. Es ist meistens schon damit getan, den ersten Fehler zu beheben und dann erneut zu übersetzen. Viele Folgefehler sind dann schon behoben. Eine Hilfe zu jeder Fehlermeldung kann mit **F1** aufgerufen werden.

B.3. Debugging

Zum Testen der Programme gibt es die so genannten Debugger. Auch in Visual C++ ist ein sehr mächtiger Debugger eingebaut. Mit Hilfe dieses Tools können z.B. die Programme Schritt für Schritt ausgeführt werden, Werte von Variablen beobachtet werden usw. Die Arbeit mit dem Debugger erfordert eine gewisse Übung um die Programme richtig auszuführen. Es stehen verschiedene Arten der Programmausführung zur Verfügung:

- Ausführung jeder einzelnen Anweisung. **F11**
- Ausführung eines Funktionsaufrufes in einem Schritt. **F10**
- Ablauf bis zu einem Breakpoint. **F5**
- Ablauf des Programms bis zur Cursorposition. **Ctrl+F10**
- Setzen/Löschen eines Breakpoints. **F9**

C. Übungen

Versuchen Sie die etwas komplexeren Übungen ab Kapitel C.5 zuerst auf dem Papier anhand eines Struktogrammes oder Programmablaufplanes zu lösen und zu verfeinern.

C.1. Übungen zu Zahlensystemen

1. **Umwandlung von Dualzahlen in Dezimalzahlen**
   Welchen Zahlen würden die folgenden Dualzahlen im Zehnersystem entsprechen?

   - $10010001_2 =$
   - $1010_2 =$
   - $11101011_2 =$
   - $111111_2 =$
   - $11010010010010101110110_2 =$
   - $110_2 =$

2. **Umwandlung von Dezimalzahlen in Dualzahlen**
   Welchen Zahlen würden die folgenden Dezimalzahlen im Binärsystem entsprechen?

   - $445_{10} =$
   - $32000_{10} =$
   - $657_{10} =$
   - $255_{10} =$
   - $128_{10} =$
   - $1024_{10} =$

3. **Umwandlung von Hexadezimalzahlen in Dual- und Dezimalzahlen**
   Welchen Zahlen würden die folgenden Hexadezimalzahlen im Binär- und Zehnersystem entsprechen?

   - $0xAFFE_{16} =$
   - $0xFEE_{16} =$
   - $0x1234_{16} =$
   - $0xCAFE_{16} =$
   - $0x8000_{16} =$
   - $0x5555_{16} =$

C.2. Übungen zum Einstieg

4. Erstellen Sie ein Programm, das nacheinander die folgenden Zeichenketten auf dem Bildschirm ausgibt:

```
Eine Meinungsumfrage unter Löwen ergab:
die Mehrheit lehnt den Käfig ab,
wünscht jedoch eine geregelte Verpflegung!
```

   Beachten Sie den Zeilenumbruch nach jeder Zeile! Machen Sie sich mit dem Debugger vertraut, und schauen Sie sich die sequentielle schrittweise Abarbeitung Ihres Programms im Debugger an.

5. Das folgende Programm enthält mehrere syntaktische Fehler:

```
*/ Schauen Sie sich das Programm wirklich genau an /*
#include < stdio.h >
int Main()
{
  printf("Das zukünftige ist viel zu unbestimmt"\n)
  printf("um als präzise Handlungsmaxime zu dienen).
  return(0);
}
```

   Korrigieren Sie die Fehler und testen Sie die Lauffähigkeit des Programms.

6. Was würde das folgende Programm am Bildschirm ausgeben?

Listing C.1:

```
#include <stdio.h>

int main()
{
  /* Frohe Weihnachten */
  printf("␣␣␣*\n␣␣***\n␣*****\n*******\n");
  printf("␣␣␣*\n␣␣␣*\n\n");
  printf("Merry␣X-max␣und\n␣␣Frohe␣Y-nachten\n");

  return(0);
}
```

7. Was würde das folgende Programm am Bildschirm ausgeben?

Listing C.2:

```
#include <stdio.h>

int main()
{
  printf("Kommentare␣werden␣in␣C␣mit␣/*..*/␣geklammert.\n");
  printf("Beachte␣jedoch:");
  printf("\n␣␣-␣Kommentare␣innerhalb␣von␣Anfuehrungszeichen\n");
  printf("␣␣␣␣werden␣als␣Text␣und␣nicht␣als␣Kommentar␣interpretiert\n");

  return(0);
}
```

8. Erstellen Sie ein Programm, das folgendes am Bildschirm ausgibt:

```
  Hauptmenue
  ==========
(A)endern
(B)eenden
(D)rucken
(E)ingeben
(L)oeschen

Was wünschen Sie zu tun?
```

C.3. Übungen zu Datentypen

9. Welche 4 Grunddatentypen kennen Sie?

10. Zählen Sie alle in ANSI C möglichen Datentyp-Angaben auf! Dabei sollten Sie folgende Gruppenzuteilung für die einzelnen Datentypen vornehmen:

    - vorzeichenbehaftete Ganzzahlen
    - nicht vorzeichenbehaftete Ganzzahlen
    - Gleitpunkttypen

11. Bereichsüberläufe beim Datentyp int. Hier wird angenommen, dass der Datentyp int 2 Bytes belegt. Geben Sie nun die resultierende Dualdarstellung mit entsprechendem Dezimalwert für folgende Dezimalzahlen im int-Datentyp an:

    - $-65000_{10} =$
    - $100000_{10} =$
    - $33000_{10} =$
    - $65535_{10} =$

12. Schreiben Sie ein Programm, das mit Hilfe von Tabulatoren und 3 Ausgabeanweisungen folgende Tabelle mit Integerkonstanten auf dem Bildschirm ausgibt:

```
1   2   3   4
2   4   6   8
3   6   9   12
```

13. Erlaubte und unerlaubte Variablennamen in C. Streichen Sie aus der folgenden Liste alle in C nicht erlaubten Variablennamen heraus:

    - hans_im_glueck
    - 7_und_40_elf
    - ____mittel_streifen
    - karl_iV
    - null_8_15
    - drei_*_hotel
    - abc_schuetze
    - kündigung
    - KINDERGARTEN

14. Definieren Sie den Tag, den Monat und das Jahr Ihres Geburtstages als Integerkonstanten und geben Sie den Satz:
„Ich wurde am tt.mm.jjjj geboren.“
auf dem Bildschirm aus, wobei tt, mm und jjjj durch die konkreten Integerkonstanten Ihres Geburtstages zu ersetzen sind.

15. Erstellen Sie nun ein Programm, das 5 Zeichen (pro Zeile eines) mit *getchar()* einliest und dann nach jeder Eingabe den zugehörigen ASCII-Wert ausgibt.
Ein Beispiel für einen möglichen Ablauf dieses Programms könnte sein:

```
d↩
d: 100
3↩
3: 51
?↩
?: 63
```

16. Erstellen Sie ein Programm, das eine Oktalzahl einliest und dann die dieser Oktalzahl entsprechende Dezimal- und Hexadezimalzahl ausgibt.

C.4. Übungen zu Operatoren

Was gibt folgendes Programm am Bildschirm aus:

Listing C.3:

```c
#include <stdio.h>

int  main()
{
   float  wert;

   wert = 49 * 49 * 49;
   printf("%f\n", wert);

   wert = 49.0 * 49 * 49;
   printf("%f\n", wert);

   wert = 49L * 49 * 49;
   printf("%f\n", wert);
   return(0);
}
```

17. Erweitern Sie das Programm aus Aufgabe 12 indem Sie sich zusätzlich die Gleitkommakonstante PI mit dem 3.1415926 definieren, jeden Wert in der Tabelle vor seiner Ausgabe mit PI multiplizieren und das Ergebnis als Gleitkommazahl ebenfalls mit Tabulatoren und 3 Ausgabeanweisungen tabellarisch ausgeben.

18. Schreiben sie ein Programm, das Sie nacheinander zur Eingabe von 2 Integerzahlen auffordert, die von Ihnen eingegebenen Zahlen auf zwei Variablen vom Typ int abspeichert und danach Summe, die Differenz, das Produkt und den Quotienten ausgibt.

19. Ändern Sie das Programm aus Aufgabe 18, indem Sie Gleitkommazahlen für die Eingabe und Ausgabe verwenden.

20. Erstellen Sie ein Programm, das eine Integerzahl einliest und dann sowohl von dieser als auch von ihren 4 direkten Nachfolgern die Quadratwurzel berechnet und zusammen mit der Zahl in einer kleinen Tabelle ausgibt. Verwenden Sie zur Berechnung die Funktion sqrt() der Standardbibliothek math.

21. Schreiben Sie ein Programm, das den n-ten Buchstaben des Alphabetes auf den Bildschirm ausgibt, nachdem es Sie nach der Zahl n gefragt hat.

22. Lesen Sie 3 Zahlen, eine vom Typ int, eine vom Typ long und eine vom Typ float, von der Tastatur ein. Speichern Sie die Summe der 3 Zahlen jeweils auf eine Variable vom Typ int, eine vom Typ long und eine vom Typ float ab und geben Sie die Werte auf dem Bildschirm aus. Erklären Sie sich die unterschiedlichen Ergebnisse.

23. Dividieren Sie zwei Integerzahlen, die Sie von der Tastatur eingelesen haben, einmal mit und einmal ohne erzwungene Typumwandlung zur Gleitkommazahl und speichern Sie das Ergebnis jeweils auf eine Variable vom Typ float. Geben Sie da Ergebnis aus und erklären sich die Abweichung der beiden Werte, wenn die eine Integerzahl nicht durch die andere teilbar ist.

24. Schreiben Sie ein Programm, das von Ihnen einen float-Wert einliest, von diesem Wert 1 subtrahiert und das Ergebnis als float-Wert wieder ausgibt. Geben Sie nach Abfrage durch das Programm den Wert 1.23456e10 ein. Welches Ergebnis erwarten Sie, wenn Ihr Programm eine Million mal die 1 von Ihrem Wert subtrahieren würde?

25. Erstellen Sie ein Programm zur Berechnung von Eigenschaften eines Würfels, das bei einzugebender Kantenlänge a

    - den Flächenumfang ($= 6 \bullet a \bullet a$)
    - das Volumen ($= a \bullet a \bullet a$)
    - die Länge der Raumdiagonale ($= a \bullet sqrt(3)$)

    berechnet und ausgibt.

26. Welchen Wert haben die folgenden arithmetischen Ausdrücke?

    ```
    3/4     15%4     15/2.0     3+5%4     3*7%4
    ```

27. Wie werden die Operanden und Operatoren im folgenden Ausdruck zusammengefasst?
    x = -4 * ++i - 6 % 4
    Setzen Sie entsprechende Klammern und überprüfen Sie, ob das Programm noch den gleichen Wert für x ermittelt.

28. Bestimmen und erklären Sie sich die Bildschirmausgaben des nachstehenden Programms.

Listing C.4:

```
#include <stdio.h>

int main()
{
  int erg, a, b, c;
  int y=5;
  printf("Wert von 7||(y==0): %d\n",7||(y==0));
  printf("Wert von y: %d\n\n",y);
  a = b = c = 0;
  erg = ++a || ++b && ++c;
  printf("erg = %d, a = %d, b = %d, c = %d\n", erg, a, b, c);
  a = b = c = 0;
  erg = ++a && ++b || ++c;
  printf("erg = %d, a = %d, b = %d, c = %d\n", erg, a, b, c);
  return(0);
}
```

29. Logische Ausdrücke
Schreiben Sie ein Programm mit dessen Hilfe Sie für eine Integerkonstante x=7 den Wert der folgenden logischen Ausdrücke berechnen und anzeigen können. Erklären Sie sich die ermittelten Ausgaben!

```
x < 9 && x >= -5
!x && x >= 3
x++ == 8 || x == 7
```

30. Welche Ergebnisse würde die Auswertung der folgenden Ausdrücke liefern:

- 4 - 11 - -6 =
- (4- 11) - -6 =
- 4 - (11 - -6) =
- 7 + 410 % 4 * 100 - 3 =
- 5.6 / 2 * 1.4 =
- (5.6 / 2) * 1.4 =
- 5.6 / (2 * 1.4) =

31. Welchen Wert hätte die Variable x am Ende des folgenden Programms:

Listing C.5:

```
#include <stdio.h>

int main()
{
  int x;

  x = 1;
  x = x+x;
  x = x+2*x;
  return(0);
}
```

32. Geben Sie zu den folgenden mathematischen Ausdrücken die entsprechende C-Ausdrücke an.

$$\frac{18}{2} \cdot \frac{4+5}{9-6} \%(6 + \frac{8}{4})$$

$$\frac{4 - 10 + \frac{100+100-40+80}{5 \cdot 2 \cdot 4} + 36}{\frac{90-30}{10-5}}$$

33. Die Zahl PI lässt sich auf 6 Stellen genau mit der folgenden Kettenbruchentwicklung berechnen:

$$PI = 3 + \frac{1}{7 + \frac{1}{15 + \frac{1}{1 + \frac{1}{292 + \frac{1}{2}}}}}$$

Ergänzen Sie nun das folgende Programm, indem sie diesen Kettenbruch in einen C-Ausdruck umwandeln, den Sie der Variablen pi zuweisen.

```
#include <stdio.h>

int main()
{
  float pi;

  pi = ....

  printf("PI = %f\n", pi);
  return(0);
}
```

34. Welche Ausdrücke wären in C erforderlich, um zu überprüfen,
    - ob der Wert der Variablen a im Intervall [-20,100] liegt
    - ob der Wert der Variablen x negativ x, aber zugleich auch der Wert der Variablen y im Intervall [5,30] liegt
    - ob der Wert der ganzzahligen Variablen z ungerade ist und zugleich auch durch 3 und 5 teilbar ist
    - ob der Wert der ganzzahligen Variablen jahr
      - durch 400 oder
      - durch 4 aber nicht durch 100 teilbar ist
    - ob das Produkt der beiden int-Variablen a und b in den Datentyp unsigned char ohne Überlauf untergebracht werden kann
    - ob der Wert der char-Variablen antwort werden das Zeichen 'j' noch das Zeichen 'J' enthält
    - ob der Wert der int-Variablen zaehler nicht im Intervall [5,25] liegt
35. Geben sie Ausdrücke an, die folgende Prüfungen durchführen. Bei den Überprüfungen sollten Sie nur logischen und Bitoperatoren verwenden. Geben Sie nun Ausdrücke an, die überprüfen ob
    - der Wert der int-Variablen x ungerade ist
    - der Wert der unsigned Variablen x größer als 255 ist
    - das 7.Bit (von rechts her gezählt) in der unsigned Variablen x auf 1 gesetzt ist
    - der Wert der int-Variablen x im Intervall [0,127] liegt
    - der Wert der unsigned-Variablen x durch 4 teilbar ist

36. Was würde das folgende Programm ausgeben?

Listing C.6:

```
#include  <stdio.h>

int main()
{
   int  a, b, c, x;
     x = -4+5*6-7;     printf("%d\n", x);
     x =  4+5*6-7;     printf("%d\n", x);
     x =  4*5%6-7;     printf("%d\n", x);
     x =  (4+5)%6/4;   printf("%d\n", x);

     a = 3;
     b = 2;
     c = 1;
     x = a | b & c;    printf("%d\n", x);
     x = a | b & -c;   printf("%d\n", x);
     x = a ^ b & -c;   printf("%d\n", x);
     x = a & b && c;   printf("%d\n", x);

     a = 1;
     b = -1;
     x = a | !a;       printf("%d\n", x);
     x = a | -a;       printf("%d\n", x);
     x = a ^ a;        printf("%d\n", x);
     x = a << 4;       printf("%d\n", x);
     b = b << 4;       printf("%d\n", b);
     b = b >> 4;       printf("%d\n", b);
     return(0);
}
```

37. Was würde das folgende Programm ausgeben?

Listing C.7:

```
#include <stdio.h>

int main()
{
   int zahl, zahl_1, zahl_2, zahl_3;

   zahl_1=1023;
   zahl_2=-1;
   zahl_3=14;

   zahl=zahl_1|zahl_2;
   printf("%d",zahl);
   printf("\n");

   zahl_2=zahl_1&0x000f;
   printf("%d",zahl_2);
   printf("\n");

   zahl_3=zahl_3^zahl_2;
   printf("%d",zahl_3);
   printf("\n");

   zahl_2=zahl_2<<zahl_3;
   printf("%d",zahl_2);
   printf("\n");

   zahl_1=zahl&&zahl_1&&zahl_2&&zahl_3;;
   printf("%d",zahl_1);
   printf("\n");

   zahl_2=zahl_2<<(zahl_1+2);;
   printf("%d",zahl_2);
   printf("\n");

   printf("%d",~zahl_2);
   printf("\n");

   return(0);
}
```

38. In einer int-Variablen (2 Byte Länge) x sei ein beliebiger Wert gespeichert. Es sollen nun die beiden Bytes in der Variablen vertauscht werden. Wenn z.B. in der Variablen x der Wert 0xabcd stünde, dann sollte nach der Zuweisung dort der Wert 0xcdab stehen. Schreiben Sie in Programm, das nach Eingabe der Variablen diese Vertauschung vornimmt und den Wert vor und nach der Vertauschung ausgibt.

39. Schreiben Sie ein Programm, das Ihnen das Bitmuster des Wertes in einer char-Variablen ausgibt. Wenn z.B. das Muster des Zeichens 'e' ausgegeben werden soll, dann sollte Ihr Programm die Bitfolge 01100101 ausgeben.
Hierzu noch ein Hinweis: Geben Sie jedes einzelne Bit mit einem printf aus. Sie müssen also printf acht mal aufrufen, um jedes einzelne Bit der Variablen auszugeben.

40. Was würde das folgende Programm ausgeben?

Listing C.8:

```
#include  <stdio.h>

int main()
{
   int  a=10, b=20, c=10;

   a=++b+c;
   printf("a=%d\n", a);
   printf("b=%d\n--------\n", b);

   b--;
   c*=--b;
   printf("a=%d\n", a);
   printf("b=%d\n", b);
   printf("c=%d\n", c);

   return(0);
}
```

41. Kennzeichnen Sie im folgenden Programm alle nicht erlaubten Ausdrücke. Sie sollten diese Aufgabenstellung zunächst ohne Zuhilfenahme des Compilers lösen.

Listing C.9:

```
#include  <stdio.h>

int main()
{
     int    a, b;
     float  x, y;

     a = -2;
     x = 10.5;
     y = -2.5;
     y += x++;
     y /= --x;
     a <<= a;
     x = 3.5<<2;
     y = ~x;
     x = 4.7|3;
     y = x%5;
     a &= 3;
     x &= 3;

     return(0);
}
```

42. Was würde das folgende Programm ausgeben?

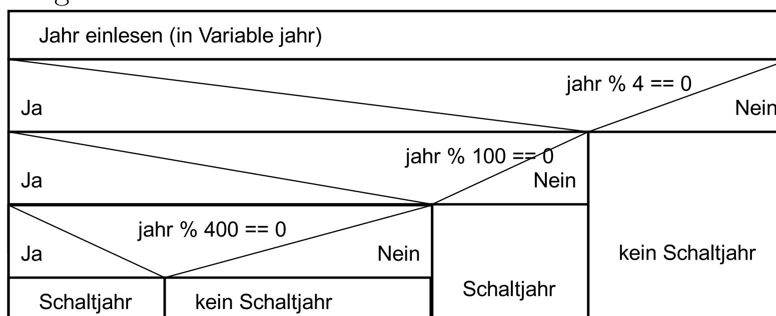
Listing C.10:

```
#include <stdio.h>

int main()
{
   int a,b,c;

   a=b=c=1;
   ++a || ++b && ++c;     printf("a=%d, b=%d, c=%d\n", a, b, c);

   a=b=c=1;
   ++a && ++b && ++c;     printf("a=%d, b=%d, c=%d\n", a, b, c);

   a=b=c=1;
   ++a && ++b || ++c;     printf("a=%d, b=%d, c=%d\n", a, b, c);

   a=b=c=1;
   ++a || ++b || ++c;     printf("a=%d, b=%d, c=%d\n", a, b, c);

   a=b=c=-1;
   ++a || ++b && ++c;     printf("a=%d, b=%d, c=%d\n", a, b, c);

   a=b=c=-1;
   ++a && ++b && ++c;     printf("a=%d, b=%d, c=%d\n", a, b, c);

   a=b=c=-1;
   ++a && ++b || ++c;     printf("a=%d, b=%d, c=%d\n", a, b, c);

   a=b=c=-1;
   ++a || ++b || ++c;     printf("a=%d, b=%d, c=%d\n", a, b, c);

   return(0);
}
```

C.5. Übungen zu Verzweigungen

43. Schreiben sie ein Programm, das zwei int-Werte von der Tastatur einliest und mit Hilfe der if-Anweisung das Maximum dieser zwei Werte bestimmt und ausgibt.

44. Erstellen sie in Programm, das eine Jahreszahl einliest und dann ausgibt, ob es sich bei diesem Jahr um ein Schaltjahr handelt oder nicht. Das nachfolgende Struktogramm zeigt die Regeln für ein Schaltjahr. Setzen Sie nun diese Struktogramm als Programm um.



45. Ein Versandgeschäft berechnet bei Aufträgen bis zu 500EUR einen Verpackungszuschlag von 10EUR und Versandkosten von 15EUR. Bei Rechnungsaufträgen von 500EUR bis 2000EUR liefert es ohne Berechnung von Versandkosten, berechnet aber einen Verpackungszuschlag von 7EUR. Bei Rechnungen über 2000EUR entstehen dem Kunden keine zusätzlichen Kosten. Erstellen Sie nun ein Programm, das den ursprünglichen Betrag einliest und dann auf diesen Betrag die anfallenden Verpackungs- und Versandkosten aufaddiert.

46. Die Lösung der Gleichung $ax^2+bx+c=0$ soll unter Berücksichtigung aller Fälle für die Koeffizienten a, b, c ermittelt werden. Die Lösungsformeln lauten bekanntlich

$$x_1 = \frac{-b+\sqrt{d}}{2a}$$
$$x_1 = \frac{-b-\sqrt{d}}{2a}$$

wobei $d = b^2 - 4ac$ die Diskriminante ist. Für die Anzahl der Lösungen sind 6 Fälle zu unterscheiden:

- a=0 und b=0 und c=0, dann gibt es unendlich viele Lösungen
- a=0 und b=0 und c!=0, dann gibt es keine Lösung
- a=0 und b!=0, dann gibt es genau 1 Lösung
- a!=0 und d>0, dann gibt es genau 2 Lösungen
- a!=0 und d=0, dann gibt es genau 1 Lösung
- a!=0 und d<0, dann gibt es keine Lösung

Erstellen Sie nun ein Programm, das die Koeffizienten a,b,c einliest und dann die Lösungen ausgibt.

47. Was würde das folgenden Programm am Bildschirm ausgeben?
Sie sollten versuchen, diese Ausgabe zu ermitteln, ohne dass Sie dieses Programm eintippen.

Listing C.11:

```
#include <stdio.h>

int main()
{
   int a,
       b=2,
       c=0;

   if (b)
      c=15;
   printf("c=%d\n", c);

   if (b==0) c=10; else c=20;
   printf("c=%d\n", c);

   a=1; c=50;
   if (a<0) if (b>0) c=100; else c=200;
   printf("c=%d\n", c);

   if (b=a<0)
      c=20;
   else if (c==0)
      c=30;
   else
      c=40;
   printf("a=%d, b=%d, c=%d\n", a, b, c);

   c=0;
   if (a=(c==0)) c=10; c=30;
   printf("a=%d, c=%d\n", a, c);

   a=0;
   c=50;
   if (c=b=a); c=10;
   printf("a=%d, b=%d, c=%d\n", a, b, c);
   return(0);
}
```

48. Erstellen Sie ein Programm, das 5 Zahlen einliest und dann die kleinste und die größte der eingegebenen Zahlen wieder ausgibt.

49. Erstellen Sie ein Programm, welches eine Stunde einliest und abhängig von der Stunde folgendes ausgibt:

Stunde	Ausgabe
23,0,1,2,3,4,5	Gute Nacht
6,7,8,9,10	Guten Morgen
11,12,13	Mahlzeit
14,15,16,17	Schönen Nachmittag
18,19,20,21,22	Guten Abend
sonst	keine erlaubte Stunden-Angabe

50. Erstellen Sie ein Programm, das zwei ganze Zahlen x und y einliest und dann ausgibt, ob x durch y teilbar ist.

51. Eine Gerade wird durch die folgende Gleichung beschrieben: $y = ax + b$
Erstellen Sie ein Programm, das die ganzen Zahlen a und b für die Gerade einliest. Danach soll es noch die Koordinaten eines Punktes einlesen, bevor es ausgibt, ob dieser Punkt auf der vorgegebene Geraden liegt oder nicht.

52. Was würde das folgenden Programm am Bildschirm ausgeben?
Sie sollten versuchen, diese Ausgabe zu ermitteln, ohne dass Sie dieses Programm eintippen.

Listing C.12:

```
#include <stdio.h>

int   main()
{
   int a=1,
       b=1,
       c=1;

   a+=b+=c;
   printf("%d\n", a<b ? a : b);
   printf("%d\n", a<b ? ++a : ++b);
   printf("a=%d, b=%d\n", a, b);

   c += a<b ? a++ : b++;
   printf("a=%d, b=%d, c=%d\n", a, b, c);
   return(0);
}
```

53. Erstellen Sie ein Programm, bei dem der Benutzer wählen kann, welche der folgenden Flächen er sich berechnen lassen möchte:
    - Quadrat: $A = a^2$ (a ist die einzugebende Seitenlänge)
    - Rechteck: $A = a * b$ (a und b sind die einzugebenden Seitenlängen)
    - Kreis: $A = a^2 * PI$ (a ist der Radius des Kreises)
    - Ellipse: $A = a * b * PI$ (a ist der Radius der x-Achse und b der Radius der y-Achse der Ellipse)

54. Man fordert jemandem auf, ausgehend von seinem Geburtsmonat, eine Reihe von Berechnungen durchzuführen. Lässt man sich die berechnete Zahl nenne
    a) Geburtsmonat mit 2 multiplizieren
    b) Auf dieses Ergebnis 5 aufaddieren
    c) neues Ergebnis mit 50 multiplizieren
    d) auf dieses Ergebnis das Alter aufaddieren
    e) vom bisherigen Ergebnis dann den Wert 365 subtrahieren

    Auf das genannte Endergebnis muss der Ratende nun noch den Wert 115 addieren. Die letzten beiden Ziffern dieses Ergebnisses geben dann das Alter und die vordere Ziffer bzw. Ziffern geben den Geburtsmonat an. Schreiben Sie nun ein Programm,

welches dem Benutzer die einzelnen Berechnungen durchführen lässt, das berechnete Ergebnis dann einliest und dann den Geburtsmonat und das Alter des Benutzers errät. Der Monat ist dabei nicht als Zahl, sondern als Name auszugeben.

55. Erstellen Sie ein Programm, das ein Datum einliest und dann das Datum des darauf folgenden Tages ausgibt. Schaltjahre sind dabei zu beachten (siehe auch Aufgabe 44.

56. Es ist ein Programm zu erstellen, welches ein Zeichen einliest und den dazugehörigen Morse-Code ausgibt. Der Morse-Code ist folgender:

A . -	N - .	0 - - - - -
B - . . .	O - - -	1 . - - - -
C - . - .	P . - - .	2 . . - - -
D - . .	Q - - . -	3 . . . - -
E .	R . - .	4 . . . . -
F . . - .	S . . .	5 . . . . .
G - - .	T -	6 - . . . .
H	U . . -	7 - - . . .
I . .	V . . . -	8 - - - . .
J . - - -	W . - -	9 - - - - .
K - . -	X - . . -	
L . - . .	Y - . - -	
M - -	Z - - . .	

- (Strich) steht dabei für einen langen und . (Punkt) für einen kurzen Ton. Neben dieser Form der Ausgabe sollte zusätzlich noch der akustische Morse-Code für das angegebene Zeichen ausgegeben werden. Zur Tonausgabe unter WindowsNT und Visual C++ steht die Funktion Beep() und für eine Pause die Funktion Sleep() in der Bibliothek winbase.h zur Verfügung.

C.6. Übungen zu Wiederholungen

57. Was gibt das folgende Programm auf dem Bildschirm aus? Versuchen Sie die Lösung zuerst ohne den Compiler zu lösen.

Listing C.13: aufgabe

```
#include <stdio.h>

int main()
{
  unsigned short int i=0;

  printf("Beginne Schleife: i = %u\n", i);

  for (i=20; i>=0; i--)
  {
    printf("i = %u\n", i);
  }

  printf("Fertig mit der Schleife: i = %u", i);
  return(0);
}
```

58. Modifizieren Sie Ihr Programm aus Aufgabe 43 derart, dass das Programm nach der Ausgabe des Maximums den Benutzer fragt, ob weitere Werte zu bearbeiten sind, und ggf. wieder von vorn beginnt. Nutzen Sie zur Lösung zunächst die do-Anweisung, dann die while-Anweisung und zum Schluss die for-Anweisung. Machen Sie sich die Unterschiede klar!

59. Erstellen Sie ein Programm, das zyklisch in beliebiger Anzahl int-Werte einliest, dabei intern den arithmetische Mittel bildet und dieses zum Schluss ausgibt. Sie können den Nutzer am Anfang ihres Programms nach der Anzahl der einzulesenden Zahlen fragen oder die Eingabe mit einem Buchstaben (z.B. ‘q’) beenden lassen.

60. Erstellen Sie ein kurzes Programm, das folgende Tabelle erstellt.

z	hex	num	Bit	1	2	3	4	z	hex	num	Bit	1	2	3	4
A	41	65	1000001	@	A	A	A	a	61	97	1100001	‘	a	a	a
B	42	66	1000010	B	@	B	B	b	62	98	1100010	b	‘	b	b
C	43	67	1000011	B	A	C	C	c	63	99	1100011	b	a	c	c
D	44	68	1000100	D	D	@	D	d	64	100	1100100	d	d	‘	d
E	45	69	1000101	D	E	A	E	e	65	101	1100101	d	e	a	e
F	46	70	1000110	F	D	B	F	f	66	102	1100110	f	d	b	f
G	47	71	1000111	F	E	C	G	g	67	103	1100111	f	e	c	g
H	48	72	1001000	H	H	H	@	h	68	104	1101000	h	h	h	‘
I	49	73	1001001	H	I	I	A	i	69	105	1101001	h	i	i	a
J	4a	74	1001010	J	H	J	B	j	6a	106	1101010	j	h	j	b
K	4b	75	1001011	J	I	K	C	k	6b	107	1101011	j	i	k	c
L	4c	76	1001100	L	L	H	D	l	6c	108	1101100	l	l	h	d
M	4d	77	1001101	L	M	I	E	m	6d	109	1101101	l	m	i	e
N	4e	78	1001110	N	L	J	F	n	6e	110	1101110	n	l	j	f
O	4f	79	1001111	N	M	K	G	o	6f	111	1101111	n	m	k	g
P	50	80	1010000	P	P	P	P	p	70	112	1110000	p	p	p	p
Q	51	81	1010001	P	Q	Q	Q	q	71	113	1110001	p	q	q	q
R	52	82	1010010	R	P	R	R	r	72	114	1110010	r	p	r	r
S	53	83	1010011	R	Q	S	S	s	73	115	1110011	r	q	s	s
T	54	84	1010100	T	T	P	T	t	74	116	1110100	t	t	p	t
U	55	85	1010101	T	U	Q	U	u	75	117	1110101	t	u	q	u
V	56	86	1010110	V	T	R	V	v	76	118	1110110	v	t	r	v
W	57	87	1010111	V	U	S	W	w	77	119	1110111	v	u	s	w
X	58	88	1011000	X	X	X	P	x	78	120	1111000	x	x	x	p
Y	59	89	1011001	X	Y	Y	Q	y	79	121	1111001	x	y	y	q
Z	5a	90	1011010	Z	X	Z	R	z	7a	122	1111010	z	x	z	r

In der Tabelle wird für Groß- und Kleinbuchstaben jeweils die dezimale, hexadezimale und binäre Darstellung ausgegeben. Danach wird jeweils mit Hilfe eines Zuweisungsoperators das erste, zweite, dritte oder vierte Bit gelöscht und das Ergebnis als char-Zeichen ausgegeben. Interpretieren Sie das Ergebnis.

61. Schreiben Sie ein Programm, das auf Wunsch Funktionstabellen für (mindestens 3) trigonometrische Funktionen generieren kann. Der Nutzer soll dabei die Funktion, die untere und obere Schranke und die Schrittweite wählen können. Achten Sie auf eine übersichtliche Darstellung auf dem Bildschirm. Sind mehr Daten anzuzeigen, als auf dem Bildschirm passen, so ist der Bildschirm erst nach Tastendruck durch den Nutzer weiter zu scrollen.

62. Schreiben Sie ein C-Programm, welches das „kleine Einmaleins“ in einer Tabelle (wie unten angegeben) vollständig auf dem Bildschirm ausgibt.

	1	2	3	4	5	6	7	8	9	10
1	1	2	3	.	.	.	.	.	.	.
2	2	4	6	.	.	.	.	.	.	.
3	3	6	9	.	.	.	.	.	.	.
4	.	.	.	.	.	.	.	.	.	.
5	.	.	.	.	.	.	.	.	.	.
6	.	.	.	.	.	.	.	.	.	.
7	.	.	.	.	.	.	.	.	.	.
8	.	.	.	.	.	.	.	.	.	.
9	.	.	.	.	.	.	.	.	.	.
10	.	.	.	.	.	.	.	.	.	.

63. Der Computer „merkt“ sich eine Zufallszahl zwischen 1 und 15, die der Spieler (=Benutzer) erraten soll. Der Spieler hat insgesamt drei Versuche. Nach jedem falschen Versuch gibt der Computer an, ob die angegebene Zahl zu klein oder zu groß ist. Ist auch der 3. Versuch erfolglos, wird die gesuchte Zahl ausgegeben. Der Spieler hat gewonnen, wenn er innerhalb der 3 Versuche, die Zahl errät. Er soll das Spiel beliebig oft wiederholen können. Zufallszahlen werden mit der Funktion srand() erzeugt.

64. Erstellen Sie ein Programm, das die harmonische Reihe berechnet, wobei der Endwert n einzugeben ist:
$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ... + \frac{1}{n}$$

65. Erstellen Sie ein Programm, das bei 1 beginnend eine Summe von ungeraden Zahlen berechnet. Bis zu welchem Endwert dabei die ungeraden Zahlen aufzuaddieren sind, ist einzugeben.

66. Dies ist die altbekannte Geschichte vom Händeschütteln. n Personen treffen sich auf einer Party. Jede schüttelt dabei jedem die Hände. Wie oft werden insgesamt Hände geschüttelt. Wenn z.B. 10 Personen auf der Party sind, so schüttelt die erste Person 9 anderen Personen die Hand, die zweite noch 8 usw. Erstellen Sie ein Programm, das einliest, wie viele Personen auf der Party sind und dann ausgibt, wie oft Hände auf dieser Party geschüttelt werden.

67. Erstellen Sie ein Programm, das alle Teiler zu einer Zahl, die einzugeben ist, ermittelt und ausgibt. 1 und die Zahl selbst sollen dabei nicht mit ausgegeben werden.

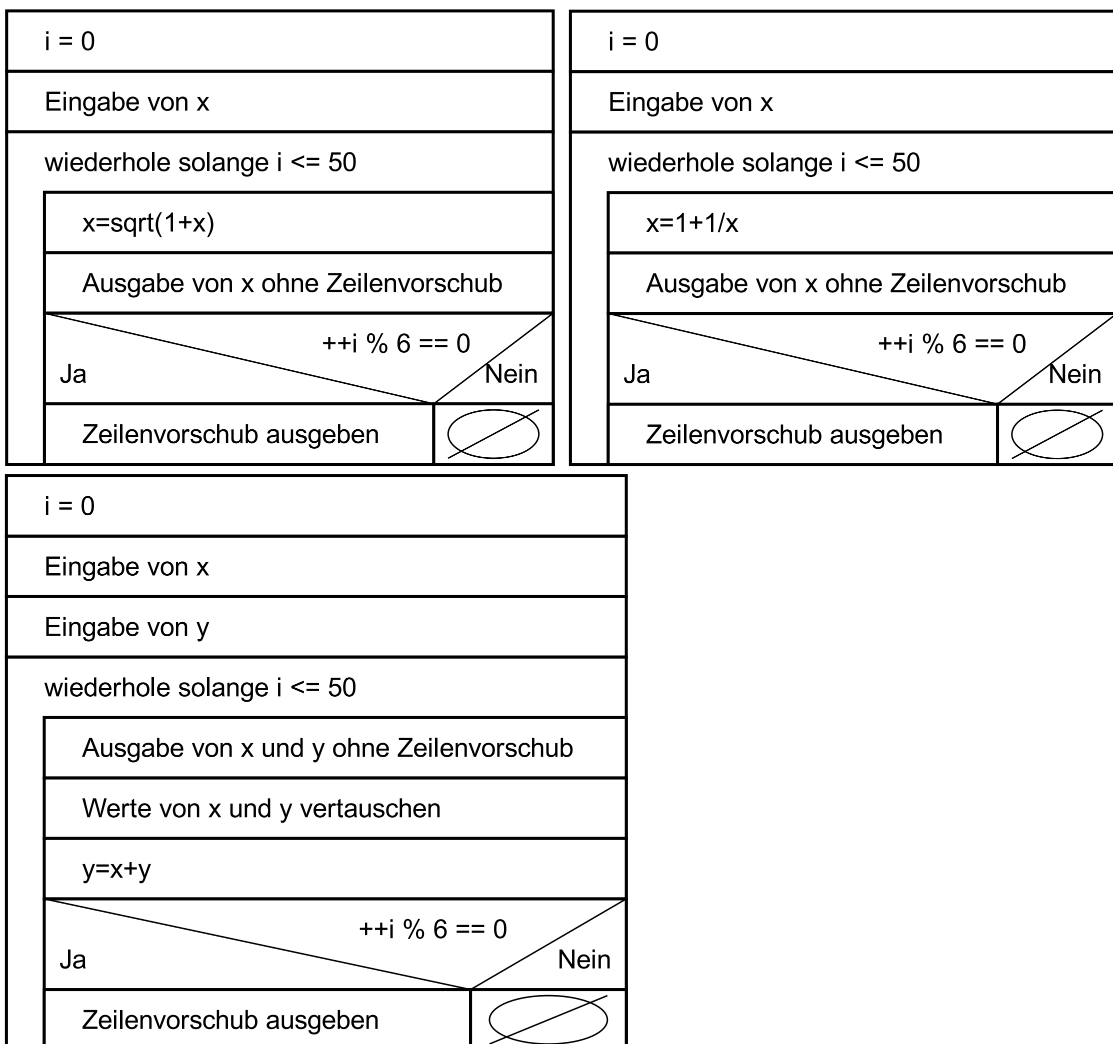
68. Winkelfunktionen können mit Reihen berechnet werden:
$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...$$
$$cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + ...$$
Erstellen Sie ein Programm, das den Benutzer zuerst zwischen Sinus und Cosinus wählen lässt bevor es x einliest. Danach kann der Benutzer noch wählen, wie viele Glieder der Reihe zu berechnen sind. Das Programm berechnet dann die entsprechende Reihe und gibt das Ergebnis aus. Zur Kontrolle soll es auch noch das Ergebnis mit Hilfe den in den Bibliotheken gegebenen Funktionen sin() bzw. cos() berechnen lassen.

69. Erstellen Sie ein Programm, das eine ganze Zahl einliest und dann deren Quersumme ausgibt. Die Quersumme einer Zahl ist die Summe aller ihrer Ziffern.

70. Erstellen Sie in Programm, das eine natürliche Zahl einliest und dann deren Dualdarstellung und deren BCD-Darstellung am Bildschirm ausgibt. Bei der BCD-Darstellung wird jede Ziffer einzeln in 4 Bits codiert, wie z.B. 2365:

    0010 0011 0110 0101
    2 3 6 5

71. Nachfolgend sind 3 Struktogramme vorgegeben, welche Sie in 3 Programme umformen sollen. Führen Sie dann diese Programme aus, überlegen Sie sich zuvor jedoch die einzelnen Ausgaben.



72. Erstellen Sie ein Programm, das sich zufällig eine Zahl aus dem Intervall [1,x] denkt. x ist vom Benutzer einzugeben. Danach soll der Benutzer versuchen, die vom Computer „gedachte“ Zahl zu erraten. Für jeden Rateversuch wird dem Benutzer mitgeteilt, ob seine Zahl zu groß oder zu klein ist.

73. Erstellen Sie ein Programm, das auf einer Maschine, die nur addieren und subtrahieren kann, die ganzzahlige Multiplikation und Division nachbildet. Einzugeben sind dabei 2 Zahlen. Das Programm gibt dann das Ergebnis der vier Rechenoperationen aus.

C.7. Übungen zu Feldern

74. Gegeben ist ein eindimensionales Feld mit n Elementen vom Typ int.

    a) Führen Sie ein einfache Rotation des Feldes nach rechts durch, d.h. das erste Elemente bekommt den Wert den n-ten Elementes zugewiesen, das Zweite den des Ersten usw.

    b) Verallgemeinern Sie ihr Programm dahingehend, dass eine m-fache Rotation (m<n) in eine beliebige Richtung des Feldes möglich ist. Die Rotationsrichtung und die Rotationsweite m sind vom Benutzer abzufragen.

75. Gegeben ist ein eindimensionales Feld der Länge n mit m Elementen (m<n) vom Typ int.

    a) Schreiben Sie ein Programm, das das k-te Elemente (k<=m) aus dem Feld entfernt und die entstandene Lücke durch heranziehen der restlichen Feldelemente schließt.

    b) Erweitern Sie ihr Programm dahingehend, dass hinter dem k-ten Element (k<=m) ein neuer int-Wert eingefügt werden kann.

    Das Programm gibt jeweils den Feldinhalt auf dem Bildschirm aus, fragt den Nutzer, ob ein Element einzufügen oder zu löschen ist und welches Element betroffen ist.

76. Gegeben ist ein eindimensionales Feld mit n Elementen vom Typ int, das bei der Definition initialisiert wird. Schreiben Sie ein Programm, das alle Duplikate aus dem Vektor entfernt, wenn

    a das Feld aufsteigend sortiert ist,

    b das Feld unsortiert vorliegt.

77. In einem eindimensionalen Feld sind n Elemente vom Typ int gespeichert. Schreiben Sie in Programm, das das häufigste Element und dessen absolute Häufigkeit ermittelt, unter der Voraussetzung,

    a das Feld aufsteigend sortiert ist (bei gleichen Häufigkeiten das kleinste Element),

    b das Feld unsortiert vorliegt (bei gleicher Häufigkeit das zuerst aufgetretene Element).

78. Erstellen Sie ein Programm, das zufällig x verschiedene Zahlen aus einem Bereich von 1 bis n ermittelt. x und n sollen dabei eingegeben werden. Für alle Lottospieler sind die Zahlen x=6 und n=49 vielleicht interessant.

79. Erstellen Sie ein Programm, das den Wochentag zu einem bestimmten Datum ausgibt. Zur Lösung dieser Aufgabenstellung gibt es viele Methoden. Eine Form soll hier anhand eines Pseudocodes dargestellt werden:

```
jh_koeff[4] = { 4, 2, 0, 5 }
monat_koeff[12] = { 2, 5, 5, 1, 3, 6, 1, 4, 0, 2, 5, 0 }

Eingabe: tag, monat, jahr (wie z.B. 2.11.2002)

jh = Jahrhundert (vorderen beiden Ziffern der Jahreszahl)
ja = Jahr im Jahrhundert (hinteren beiden Ziffern der Jahreszahl)

if (monat > 2)
    schaltjahr = 0
 else if (jahr%400==0)
    schaltjahr = 1
 else if (jahr%100==0)
    schaltjahr = 0
 else if (jahr%4==0)
    schaltjahr = 1
 else
    schaltjahr = 0

 wochen_tag = (tag+monat_koeff[monat-1]-schaltjahr+jh_koeff[jh%4]+ja+ja/4)%7

 Wochentag ausgeben (0=Sonntag, 1=Montag...)
```

Erstellen Sie zu diesem Pseudocode ein Programm.

80. Modifizieren Sie das Programm aus Aufgabe 56 derart, dass jetzt eine Zeichenkette eingelesen werden kann, die dann Zeichen für Zeichen gemorst wird. Denken Sie dabei an eine kleine Pause zwischen den einzelnen Zeichen.

C.8. Übung zu Zeichenketten

81. In der Headerdatei <string.h> sind viele Funktionen zur String-Verarbeitung bereits vorhanden. Erstellen Sie nun ihre eigenen Funktionen zum Kopieren (strcpy), Vergleichen (strcmp), Anhängen (strcat) von Strings, sowie zur Ermittlung der Länge des Strings (strlen)

82. Erstellen Sie ein Programm, das aus einer Zeichenkette, die einzugeben ist, alle Vertreter eines Zeichens, das ebenfalls einzugeben ist, löscht.

83. Erstellen Sie ein Programm, das eine Häufigkeitsverteilung über die Wortlängen in einem Text ausgibt.

84. Eine zufällige Anordnung von n Buchstaben soll durch „Spielzüge“ in die alphabetische Reihenfolge gebracht werden. Ein „Spielzug“ besteht darin, dass nach Eingabe einer Zahl k die ersten k Buchstaben in der Reihenfolge umgekehrt werden, wie z.B.

```
Wie viele Zeichen: 6
lxavbc
Bis wohin umkehren: 3
axlvbc
Bis wohin umkehren: 6
cbvlxa
Bis wohin umkehren: 2
bcvlxa
...
```

85. Erstellen Sie ein Programm, das in einer Zeichenkette, die einzugeben ist, alle Vertreter eines Wortes, das ebenfalls einzugeben ist, durch ein anderes Wort (auch einzugeben) ersetzt. Dabei sollen nur ganze Wörter ersetzt werden, d.h. dass bei Wörtern, in denen das zu ersetzende Wort nur ein Teilstring ist, keine Ersetzung stattfindet.

C.9. Übungen zu Funktionen

86. Ändern Sie die Aufgabe 56 dahingehend, dass die Ausgabe der einzelnen Buchstaben und die Auswahl des Morsecodes für ein jeweils in einer separaten Funktion stehen. Die Eingabe des Buchstabens soll in der main-Funktion verbleiben. Vermeiden Sie globale Variablen!

87. Schreiben Sie ein Programm, das neben der main-Funktion noch 3 weitere Funktionen enthält, die nacheinander aufgerufen werden. Die erste Funktion lese ein Zahl von der Tastatur ein, die zweite Funktion quadriere die eingelesene Zahl und die dritte Funktion gebe das Ergebnis mit einer Erklärung auf dem Bildschirm aus. Lösen Sie die Aufgabe ohne globale Variablen.

88. Erstellen Sie ein Programm, das auf einer Maschine, die nur addieren und subtrahieren kann, die ganzzahlige Multiplikation und Division nachbildet. Einzugeben sind dabei 2 Zahlen. Das Programm gibt dann das Ergebnis der vier Rechenoperationen aus. Es empfiehlt sich hier, für jede der 4 Rechenoperationen eine eigene Funktion zu erstellen, die dann bei Bedarf aufgerufen wird.

89. Folgender Algorithmus von Legendre liefert die Potenz a^b für reelles a>0 und natürliche Zahlen b.

```
x = a
y = b
z = 1
while (y>0)
  if (y ungerade)
    z = z*x
  y = y/2
  x = x*x
z = Potenz von a^b
```

    Erstellen Sie ein Programm, das diesen Algorithmus von Legendre testet, indem es mit ihm zufällig 10 ganze Zahlen zwischen 0 und 10 potenzieren lässt und das Ergebnis ausgibt.

90. Zuweilen, besonders bei hardwarenaher Programmierung, ist es nützlich, einzelne Bits in einem Byte überprüfen, setzen, rücksetzen und löschen zu können. Schreiben Sie für jede dieser Operationen eine kleine Funktion. Um die Funktionen testen zu können, schreiben Sie ein Programm, das zunächst ein Byte im Binärmuster einliest und die zu ändernde Bitposition abfragt. Je nach gewünschter Operation wird dann das Bitmuster entsprechend verändert.

91. Der Spieler Antoine Gombaud Chevalier de Mère fragte eines Tages den französischen Mathematiker und Philosophen Pascal, ob es wahrscheinlicher sein, mindestens eine 6 bei 4 Würfen mit einem Würfel oder mindestens eine Doppelsechs bei 24 Würfen mit 2 Würfeln zu erhalten. Erstellen sie ein Programm, das über n Simulationen diese Frage beantwortet. Die Anzahl n der Simulationen soll der Benutzer eingeben können.

92. Erstellen Sie ein Programm für die Verwaltung von Gleitzeitkonten, das zwei Uhrzeiten in der Form hh:mm:ss einliest und diese addiert bzw. subtrahiert und die Ergebnisse ausgibt.

93. Erstellen Sie ein Programm, das einen Taschenrechner für Bruch-Operationen nachbildet. Der Taschenrechner soll dabei folgende Operationen beherrschen: Addition, Subtraktion, Multiplikation, Division. Die soeben ausgeführte Berechnung soll dabei stets als gekürzter Bruch ausgegeben werden. Bei Abschluss der Berechnung sollen zusätzlich die ganzen Zahlen vor den Bruch gezogen werden, wie z.B.

```
Bitte 1.Bruch eingeben:
Zaehler (!=0): 3
Nenner  (!=0): 4
Bitte gewünschte Operation eingeben (+,-,*,/,= -> Abschluss der Berechnung)
```

```
Ihre Wahl: *
Zaehler (!=0): 5
Nenner  (!=0): 6
  3     5     5
--- * --- = ---
  4     6     8
Bitte gewünschte Operation eingeben (+,-,*,/,= -> Abschluss der Berechnung)
Ihre Wahl: +
Zaehler (!=0): 1
Nenner  (!=0): 24
  5     1     2
--- + --- = ---
  8    24     3
Bitte gewünschte Operation eingeben (+,-,*,/,= -> Abschluss der Berechnung)
Ihre Wahl: -
Zaehler (!=0): 8
Nenner  (!=0): 9
  2     8    -2
--- - --- = ---
  3     9     9
Bitte gewünschte Operation eingeben (+,-,*,/,= -> Abschluss der Berechnung)
Ihre Wahl: /
Zaehler (!=0): 8
Nenner  (!=0): 9
 -2     1    -2
--- / --- = ---
  9     9     1
Bitte gewünschte Operation eingeben (+,-,*,/,= -> Abschluss der Berechnung)
Ihre Wahl: =
vollstaendig gekürztes Endergebnis mit Ganzen:
 -2
--- = -2
  1
***************************************************************************
Bitte 1.Bruch eingeben:
Zaehler (!=0): -38
Nenner  (!=0): 15
Bitte gewünschte Operation eingeben (+,-,*,/,= -> Abschluss der Berechnung)
Ihre Wahl: +
Zaehler (!=0): 38
Nenner  (!=0): 15
-38    38     0
--- + --- = ---
 15    15   225
Bitte gewünschte Operation eingeben (+,-,*,/,= -> Abschluss der Berechnung)
Ihre Wahl: =
```

```
vollstaendig gekürztes Endergebnis mit Ganzen:
  0
--- = 0
225
***************************************************************************
Bitte 1.Bruch eingeben:
Zaehler (!=0): -61
Nenner  (!=0): 30
Bitte gewünschte Operation eingeben (+,-,*,/,= -> Abschluss der Berechnung)
Ihre Wahl: =
vollstaendig gekürztes Endergebnis mit Ganzen:
-61       1
--- = -2 ---
 30       30
```

C.10. Übungen zu Strukturen

94. Es soll ein Programm erstellt werden, das das Rechnen mit Bruchzahlen ermöglicht. Möglicher Ablauf des Programms:

```
1. Bruch: 2/3
2. Bruch: 3/4

Die beiden Brüche 2/3 und 3/4
1 (addieren), 2 (Subtrahieren), 3 (multiplizieren) 4 (dividieren)? 1
...Ergebnis: 2/3 + 3/4 = 17/12 = 17/12 (gekuerzt)
-------------------------------------------------
3. Bruch (0/0 = Ende): 5/6

Die beiden Brüche 17/12 und 5/6
1 (addieren), 2 (Subtrahieren), 3 (multiplizieren) 4 (dividieren)? 2
...Ergebnis: 17/12 - 5/6 = 42/72 = 7/12 (gekuerzt)
-------------------------------------------------
4. Bruch (0/0 = Ende): 0/0
```

95. Es soll ein Programm erstellt werden, das das Addieren von deutschen Kommazahlen ermöglicht. Der Vorkomma- und der Nachkommateil soll dabei in unterschiedlichen Komponenten einer Struktur gehalten und auch getrennt aufaddiert werden. Vor der Ausgabe ist eventuell ein entsprechender Übertrag aus dem Nachkommateil in den Vorkommateil notwendig.

96. Erstellen Sie ein Programm, das eine arabische Zahl einliest und dann die zugehörige römische Zahl ausgibt. Folgende römische Ziffernkombinationen stehen für die entsprechenden arabischen Zahlen:

I	1
IV	4
IX	9
X	10
XL	40
L	50
XC	90
C	100
CD	400
D	500
CM	900
M	1000

    Die Zuordnung lässt sich leicht mit einem Strukturarray lösen, das schon bei der Definition mit obigen Werten initialisiert wird.

97. Ein Dozent braucht ein Programm, das ihm die Notenübersicht für Klausuren erleichtert. Dieses Programm soll zunächst für alle Schüler eines Kurses den Namen, Vornamen und die in der Klausur erreichte Note einlesen, bevor es dann eine Namensliste mit jeweils erreichter Note, die Durchschnittsnote und einen Notenspiegel in Form eines Histogramms (Häufigkeit der einzelnen Noten) ausgibt.

Literaturverzeichnis

[1] Winfried Bantel. *Strukturiertes Programmieren in C.* Shaker Verlag, Aachen, 2001.

[2] Ulrike Böttcher. *Grundlagen der Programmierung.* HERDT-Verlag, Nackenheim, 1. edition, 2001.

[3] Helmut Erlenkötter. *C Programmieren von Anfang an.* rororo Computer, 4. edition, 2001.

[4] Dr. Wilhelm Hanrath. *Einführung in die Programmiersprache C.*

[5] Helmut Herold. *Die Programmiersprache C.*

[6] Rolf Isernhagen. *Softwaretechnik in C und C++.* Hanser Verlag, München, 2. edition, 2000.

[7] Klaus Schmaranz. *Softwareentwicklung in C.* Springer-Verlag, 2001.