

# **Einführung**

## **in die**

# **Programmiersprache C**

Ergänzendes Skript zur Vorlesung

Verfasser: Gerhard Krucker

Datum: Herbst/ Winter 2003/2004

## Vorwort

Es gehört heute zum Grundwissen eines jeden Ingenieurs, mindestens eine Programmiersprache zu beherrschen. Der Trend der letzten Jahre zeigt eindeutig, dass im technisch -wissenschaftlichen Bereich die Sprache C/C++ führend vertreten ist.

C bietet die Möglichkeiten von Assembler auf der Stufe der Hochsprache. Daher kann man in C sehr maschinenorientiert programmieren. Andererseits kann man aber auch problemorientiert Programme codieren. C++ ist eine grosse Erweiterung von C, dazu gehören auch die objektorientierten Fähigkeiten. Sie ist neben Java die wohl am meisten eingesetzte objektorientierte Sprache im technisch/ wissenschaftlichen Bereich.

Mittlerweile sind heute praktisch alle Compiler für C auf PC-Plattformen eigentlich C++ Compiler. Deshalb werden wir in der Einführung bereits die nicht objektorientierten Erweiterungen der Sprache C++ verwenden. In diesem Sinn werden wir C++ also als 'besseres C' benutzen. Dieses Wissen ermöglicht einen einfachen Einstieg in die Sprache Java, da Java viele Elemente von C++ beinhaltet.

Das Skript soll die wesentlichen Elemente der Sprache zeigen. Weiterführend wird ausdrücklich auf die zahlreichen Lehrbücher verwiesen. Beachten Sie dazu das Literaturverzeichnis am Ende des Skriptums.

Wir werden einerseits die Sprache C auf der Sprachdefinition nach ANSI erlernen. Die praktische Realisation von Programmen wird mit einem modernen Compiler vermittelt. Dazu ist Microsoft Visual C++ oder Borland C++ Builder vorgesehen. Dies sind leistungsfähige Entwicklungswerkzeuge mit integrierter Benutzeroberfläche. Sie bieten alle zur Codierung und Test notwendigen Werkzeuge, inklusive Online-Dokumentation.

Das Erlernen einer Programmiersprache muss zwingend mit entsprechenden praktischen Übungen verbunden sein. Nur so kann das Wissen dauerhaft gefestigt werden und ein Grundstock an Präsenzwissen zur Sprache aufgebaut werden.

Deshalb ist es sicher notwendig die Beispiele sowie die Übungsaufgaben seriös durchzuarbeiten. Dies ist teilweise während des Unterrichtes möglich. Jedoch sind die praktischen Übungen im Unterricht eher weniger dazu gedacht die Beispiele durchzuarbeiten, sondern als Möglichkeit individuell bei praktischen Problemen eine Frage zu stellen. Deshalb ist die Meinung, dass diese Aufgaben ausserhalb des Unterrichtes gelöst werden sollen. Je nach Vorbildung (bereits bekannte Sprachen, Programmiererfahrung, etc.) ist der Aufwand individuell unterschiedlich.

Bei Fragen zum Stoff und zu den Übungen können Sie einen Beratungstermin vereinbaren. Bitte kontaktieren Sie mich per Mail über [krucker@krucker.ch](mailto:krucker@krucker.ch) und Teilen Sie mir mit, worum es geht. Ich bin zwar sicherlich kompetent, kann aber mit einer Vorabinformation besser helfen.

Viele Dokumente wie Übungen, Skript und Vorlesungsbeilagen sind von [www.krucker.ch](http://www.krucker.ch) zugreifbar. In irgendeiner Weise Copyright behaftetes Material ist nur über den Klassenaccount auf meinem persönlichen FTP-Server zugreifbar. Die IP-Nummer und Account mit Passwort wird im Unterricht mitgeteilt.

# INHALTSVERZEICHNIS

<b>1</b>	<b>EINLEITUNG</b>	<b>1-1</b>
1.1	Allgemeines zu Programmiersprachen	1-1
1.1.1	Low Level Language	1-2
1.1.2	Höhere Programmiersprachen	1-3
1.1.3	Evolution der Programmiersprachen	1-3
1.2	Zeilen- und Symbolstruktur eines C-Programmes	1-5
1.3	Darstellung der Syntax	1-6
1.4	Syntaxdiagramme für C	1-8
<b>2</b>	<b>KONZEPT DER SPRACHE C</b>	<b>2-1</b>
2.1	Erstes Beispiel	2-2
2.2	Grundsätzlicher Aufbau von C-Programmen	2-3
<b>3</b>	<b>EINFACHE DATENEIN- UND AUSGABE IN C</b>	<b>3-1</b>
3.1	Datenausgabe mit printf()	3-1
3.2	printf() Formatspezifikationen	3-3
3.3	Spezifikationen für die Feldbreite und Genauigkeit	3-5
3.4	Dateneingabe mit scanf()	3-6
<b>4</b>	<b>DATENTYPEN UND VARIABLEN</b>	<b>4-1</b>
4.1	Definitionen	4-1
4.2	Systemdatentypen in C	<b>Fehler! Textmarke nicht definiert.</b>
4.3	Definition von Variablen	4-3
4.3.1	Schlüsselwörter in C	4-3
4.3.2	Zusätzliche Schlüsselwörter in C++	4-4
4.4	Gültigkeitsbereich von Variablen	4-4
4.5	Speicherklassen	4-5
4.6	Interne Darstellung der Datenwerte	4-6
4.7	Konstanten	4-8
4.7.1	Konstante Zeichen	4-9
4.7.2	Konstante Variablen	4-9
<b>5</b>	<b>AUSDRÜCKE UND OPERATIONEN</b>	<b>5-1</b>
5.1	Hierarchie	5-1
5.2	Operationen	5-2
5.3	Assoziativität	5-3
5.3.1	Primärausdrücke	<b>Fehler! Textmarke nicht definiert.</b>
5.3.2	Unäre Operatoren	5-4
5.4	Wertzuweisung	5-5
5.5	Vergleiche	5-6
<b>6</b>	<b>ABLAUFSTRUKTUREN</b>	<b>6-1</b>
6.1	Strukturdiagramme	6-1
6.2	Das Flussdiagramm	6-2
6.3	Nassi-Shneiderman	6-3
6.4	Anwendung	6-4
6.4.1	Sequenz	6-5
6.4.2	Alternative (Vergleiche) if (..) .. else ..	6-5
6.4.3	Fallunterscheidung (Mehrfachauswahl) switch (..)	6-7
6.4.4	Bedingungsoperator	6-8
6.4.5	Schleifenbefehle	6-9
6.4.6	Vorabprüfende Schleife: while	6-9
6.4.7	Nachprüfende Schleife: do..while	6-11
6.4.8	Kombischleife for(...)	6-13
<b>7</b>	<b>ZEIGER UND SPEICHERPLATZADRESSEN</b>	<b>7-1</b>

7.1	Motivation	7-1
7.2	Grundsätzliches über Zeiger	7-2
7.3	Anwendungen mit Zeiger	7-4
<b>8</b>	<b>BENUTZERDEFINIERTER DATENTYPEN</b>	<b>8-1</b>
8.1	Aufzähltypen (enums)	8-1
8.2	typedef-Anweisung	8-2
8.3	Komplexe Datentypen (Aggregates und Arrays)	8-3
8.4	Arrays	8-3
8.5	Strings	8-4
8.6	Allgemeine Bemerkungen zu Arrays	8-5
8.7	Structs	8-7
8.7.1	Klassische Art der Definition nach Kernighan & Ritchie	8-8
8.7.2	Neue Art nach ANSI	8-9
8.8	Ergänzungen	8-10
8.8.1	Unions	8-11
8.8.2	Bitfelder	8-11
<b>9</b>	<b>FUNKTIONEN</b>	<b>9-1</b>
9.1	Definition von Funktionen	9-1
9.1.1	Typ der Funktion	9-2
9.1.2	Parameter	9-3
9.1.3	Parameterübergabe über Zeiger	9-5
9.1.4	Der Funktionskörper	9-6
9.1.5	Funktionsprototyp	9-6
9.2	Rekursion	9-8
9.3	Arrays an Funktionen übergeben	9-9
9.4	Strings an Funktionen übergeben	9-11
9.5	Wertrückgabe über Referenzparameter	9-12
9.6	Zusammenfassung über das Thema Funktionen	9-12
<b>10</b>	<b>TYPENKONVERSION</b>	<b>10-14</b>
10.1	Implizite Konversionen	10-14
10.2	Arithmetische Konversionen:	10-16
10.3	Explizite Konversionen (Casts)	10-16
<b>11</b>	<b>DER PREPROZESSOR</b>	<b>11-1</b>
11.1	Definieren von Symbolen und Makros (#define)	11-1
11.2	Bedingte Kompilation	11-4
11.3	Einbinden von Sourcefiles (#include)	11-4
11.4	Sonstiges	11-5
<b>12</b>	<b>FILE I/O</b>	<b>12-1</b>
12.1	Zugriffsfunktionen	12-1
12.2	Benutzerdefinierte Files	12-2
12.3	Praktische Arbeit mit benutzerdefinierten Files:	12-2
12.3.1	File Pointer	12-3
12.3.2	Öffnen eines Files	12-3
12.3.3	Daten auf Files schreiben	12-5
12.3.4	Lesen von Daten aus Files	12-6
12.3.5	Lesen aus Binärfiles	12-7
<b>13</b>	<b>LITERATURVERZEICHNIS</b>	<b>13-1</b>

# 1 Einleitung

Dieses Kapitel soll einen ersten Einstieg in die praktische Arbeit mit der Programmiersprache C sowie der Arbeitsumgebung ermöglichen. Ohne vorher viel Theorie zu wälzen, wird rasch versucht ein Minimalbeispiel zum Laufen zu bringen, um so ein Motivationspotenzial anzuregen.

Für den Einstieg werden keine Vorkenntnisse erwartet. Der Einstieg ist aber sicher einfacher, wenn schon ein wenig Programmiererfahrung aus irgendeiner Sprache vorhanden ist. Jedoch sollte es auch für totale Anfänger ein gangbarer Weg sein, auch wenn die Erfolgserlebnisse eine gewisse Reifezeit brauchen und man manchmal das Gefühl hat im Pilgerschritt vorwärts zu schreiten.

Der Kurs verfolgt die Zielsetzung eine fundierte Grundlage in der Sprache C und Praxis im Umgang mit einer Standard-Entwicklungsumgebung zu vermitteln. Sie anschliessend sollen in der Lage sein, einfache Kleinprogramme auf Kommandozeilenbasis routiniert zu erstellen und C-Programme anhand des Listings in ihrem Wirkungsablauf nachzuvollziehen und kompetent zu erklären.

## Anmerkung zur Notation

Die Programmbeispiele sind in ANSI-C formuliert. Neu eingeführte Begriffe werden in ***Kursiv-schrift fett*** dargestellt. Programmcode wird in nicht proportionaler Schrift gelistet. Ausgaben werden entweder als Bildschirmfoto oder in Arial gezeigt.

## 1.1 Allgemeines zu Programmiersprachen

Eine Programmiersprache formuliert Anweisungen an einen Rechner. Geordnet lassen sich nach [KÜV99] Programmiersprachen als ***kontextfreie Sprachen*** mit einer einfachen Grammatik und kleinem Wortschatz spezifizieren.

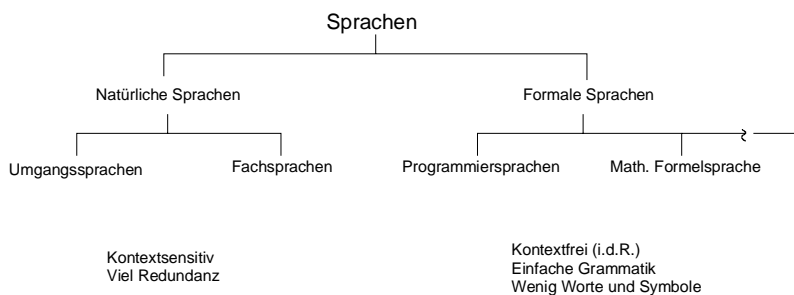


Bild 1-1: Einteilung der Sprachen nach [KÜV99].

Die Programmiersprachen lassen sich ihrerseits wiederum gruppieren:

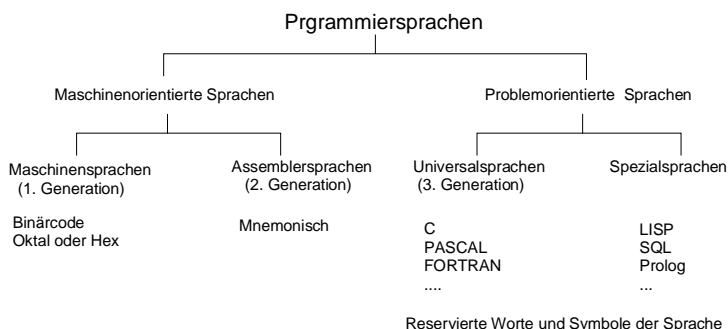


Bild 1-2: Einteilung der Programmiersprachen nach [KÜV99].

Je nach Distanz der Sprache zum **Rechenwerk** oder **Prozessor** spricht man von höheren oder **tieferen Programmiersprachen** und ordnet sie in **Generationen**. Die Generationenbezeichnung ergibt sich aus der zeitlichen Entwicklung (vgl. Tabelle 1).

Kennzeichen einer 1./2. Generationssprache ist, dass pro Anweisung genau ein Maschinenbefehl ausgeführt wird. Bei einer höheren Programmiersprache werden pro Anweisung mehrere Maschinenbefehle ausgeführt.

### 1.1.1 Low Level Language

Der Begriff „tiefe Programmiersprache“ ist im deutschen Sprachraum nicht geläufig. Hingegen ist „Low level language“ im englischsprachigen Raum sehr wohl ein Begriff. Statt dessen spricht man von 1./ 2.-Generationssprachen. Beide sind **maschinenorientierte Sprachen**, weil sie direkt mit dem Befehlsatz des Prozessors einhergehen. Vertreter dieser Gruppe sind der **Maschinencode** als Sprache der 1. Generation und **Assembler** als Sprache der 2. Generation.

Mit diesen Sprachen können die effizientesten Programme codiert werden. Der Instruktionssatz eines Prozessors kann voll ausgenutzt werden und man hat vollständige Kontrolle über alle Speicherbereiche.

Leider haben 1. und 2. Generationssprachen auch erhebliche Nachteile:

- Alle Programme sind maschinenspezifisch, d.h. sie sind an Hardware und Prozessorarchitektur gebunden. Ein Übertragen auf andere Prozessortypen und Hardware bedingen erhebliche Anpassungen bis hin zu einem Neudesign der Programme.
- Zur Programmierung steht nur Prozessorbefehlssatz mit einfachen Datentypen zur Verfügung. Viele Prozessoren verfügen nur über Ganzzahlrechnung. Gleitkommarechnung muss über Software erfolgen, was Speicherplatz und Rechenzeit benötigt.
- Moderne Prozessoren haben eine Vielzahl von Registern und Adressiermöglichkeiten. Die Codierung in Assembler mit mnemonischen Code wird sehr aufwändig.
- Maschinencodeprogrammierung hat heute keine Bedeutung mehr. Assemblerprogramme werden vor allem im Mikrocontrollerbereich noch verwendet. Auf PC- und Workstationplattformen ist Assemblerprogrammierung fast ganz verschwunden. In seltenen Fällen wird Sie noch für Bootstrap-Code oder Gerätetreiber eingesetzt.
- Assemblerprogramme sind ausserordentlich dokumentationsintensiv, wenn man später ohne Aufwand Änderungen vornehmen möchte.

Assemblerprogramme werden mit einem Editor durch Formulierung der Prozessorbefehle und Steueranweisungen in mnemonischen Code erstellt. Der Assembler erzeugt aus dem mnemonischen Code einen Objektmodul. Es enthält im Prinzip ein Maschinencode ohne feste Speicheradressen. Der Linker fasst mehrere Objektmodule zusammen und ordnet feste Speicheradressen zu, so dass ein ausführbarer Maschinencode entsteht.

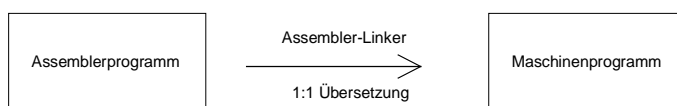


Bild 1-3: Umsetzungsprozess eines Assemblerprogrammes in ausführbaren Maschinencode.

Assembler für einfache Prozessoren können auch direkt einen ausführbaren Maschinencode erstellen. Das ist aber eher die Ausnahme. Moderne Assembler und Linker sind hochkomplexe Produkte. Sie haben eine Vielzahl von Anweisungen und Konfigurationsmöglichkeiten. Sie gehören meist zum Lieferumfang der Entwicklungsumgebungen für höhere Programmiersprachen, wie MS-Visual C++, Borland C++ Builder, MS Fortran Power Station, und andere.

### 1.1.2 Höhere Programmiersprachen

Programmiersprachen der 3. Generation nennt man *höhere Programmiersprachen*. In diese Gruppe fallen z.B. C, PASCAL, FORTRAN, BASIC und viele andere. Sie abstrahieren im Sinne einer Idealisierung den Rechner. Zur Programmierung stehen hier mächtige Instruktionen, Funktionen und Datentypen zur Verfügung. Sie sind daher *problemorientierte Sprachen*. Die Lösung einer Programmieraufgabe muss nicht mehr an den Gegebenheiten des Rechners orientieren, sondern der Rechner stellt in gewissem Sinne eine ideale Maschine dar. Programme in höheren Programmiersprachen sind in Grenzen auf andere Maschinen direkt übertragbar.

Die Programmierung erfolgt im Quellcode mit einem Editor. Komfortabler sind integrierte Entwicklungsumgebungen. Diese werden meist als IDE (Integrated Development Environment) bezeichnet. Sie Enthalten alle zum Codierprozess notwendigen Werkzeuge wie Editor, Compiler, Linker und Debugger.

Der Compiler wandelt den Quellcode in den vom Prozessor ausführbaren Maschinencode um. Ein anderes Prinzip verfolgt die *Interpreter-Lösung*. Das Programm läuft auf der Ebene des Quellcodes ab. Ein Interpreter wandelt dabei jede Anweisung in für den Prozessor ausführbare Instruktionen um. Es existieren auch Hybrid-Lösungen. Der Quellcode wird einmalig in einen kompakten Zwischencode kompiliert (p-Code, Bytecode). Beim Programmstart wird der Zwischencode über einen Interpreter abgearbeitet. Typische Vertreter sind Microsoft-BASIC und JAVA.

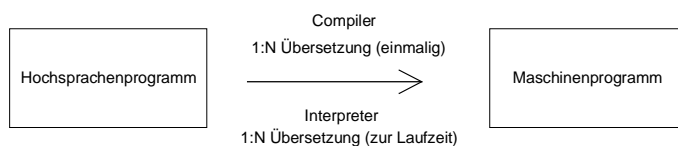


Bild 1-4: Umsetzungsprozess eines Hochsprachenprogrammes in ausführbaren Maschinencode.

### 1.1.3 Evolution der Programmiersprachen

Programmiersprachen haben sich von den Anfängen der Rechnertechnik stark entwickelt. Zu Beginn der Computerära in den 50er Jahren wurde eine effiziente Programmausführung mit wenig Rechner-Ressourcen gefordert. Damals war die Hardware extrem teuer. Die Kosten für die Programmierer waren im Verhältnis zur Hardware klein.

Mitte der 60er Jahre erkannte man, dass für die immer komplexer werdenden Programme neue Anforderungen notwendig wurden. Das Ziel war die Entwicklung korrekter Programme. Die Grundlage des Software-Engineerings, als Disziplin für methodische Software-Entwicklung, wurde geboren.

Heute sind die Verhältnisse so, dass Hardware praktisch nichts mehr kostet, die Programmentwicklung aber ist extrem teuer. Die Anforderungen an Programme sind stark gestiegen. Softwareprodukte sind äusserst komplex und man braucht auf jeder Stufe des Entwicklungsprozesses ein methodisches, dem Problem angepasstes, Vorgehen.

Aus dieser Sicht ist eine Programmiersprache nur ein (kleines) Werkzeug in einer langen Kette von Aktivitäten in einem Entwicklungsprojekt. Es beginnt mit einer sauberen, vollständigen Analyse mit Pflichtenheft, gefolgt einem Design einer hinreichend guten Lösung. Die Codierung setzt das Design in eine ablauffähige Lösung um. Der Test verifiziert die Erfüllung der Anforderungen und dokumentiert nachvollziehbar die Korrektheit. Selbstverständlich sind alle Phasen von Analyse bis Test den Vorgaben entsprechend zu dokumentieren.

Die Evolution von Hard und Software kann gemäss [PRA98] in Fünfjahresschritten beschrieben werden:

<b>Fünfjahresspanne</b>	<b>Einflüsse und Technologien</b>
1951-1955	<i>Hardware:</i> Computer, aufgebaut aus Elektronenröhren; Quecksilber- Laufzeitspeicher <i>Methoden:</i> Assemblersprachen; Grundlagenkonzepte: Unterprogramme, Datenstrukturen <i>Sprachen:</i> experimentelle Nutzung von Ausdruckscompilern
1956-1960	<i>Hardware:</i> Magnetbandspeicherung; Kernspeicher; Transistorschaltungen <i>Methoden:</i> Erste Compilertechnologien; BNF-Notationen; Code-Optimierung; Interpreter; dynamische Speichermetoden und Listenverarbeitung <i>Sprachen:</i> FORTRAN, ALGOL 58, ALGOL 60, COBOL, LISP
1961-1965	<i>Hardware:</i> Familien kompatibler Architekturen; Magnetplattenspeicher <i>Methoden:</i> Programmübergreifende Betriebssysteme; syntaxorientierte Compiler <i>Sprachen:</i> COBOL-61, ALGOL 60 (revidiert), SNOBOL, JOVIAL, APL-Notation
1966-1970	<i>Hardware:</i> Zunehmende Grösse und Geschwindigkeit bei sinkenden Preisen; Minicomputer; Mikroprogrammierung; integrierte Schaltungen <i>Methoden:</i> Time-Sharing und interaktive Systeme; optimierende Compiler <i>Sprachen:</i> APL, FORTRAN 66, COBOL 65, ALGOL 68, SNOBOL4, BASIC, PLII, SIMULA 67, ALGOL-W
1971-1975	<i>Hardware:</i> Mikrocomputer; Zeitalter der Minicomputer; kleine Massenspeichersysteme; Aufkommen von Halbleiterspeichern und Ablösung der Kernspeicher <i>Methoden:</i> Programmverifikation; strukturierte Programmierung; erstes Erscheinen von Software-Engineering als Studienrichtung <i>Sprachen:</i> Pascal, COBOL 74, PL/1 (Standard), C, Scheme, Prolog
1976-1980	<i>Hardware:</i> Mikrocomputer in kommerzieller Qualität; grosse Massenspeichersysteme; verteilte Rechnerumgebungen <i>Methoden:</i> Datenabstraktion; formale Semantik; parallele, eingebettete und Echtzeitprogrammierung <i>Sprachen:</i> Smalltalk, Ada, FORTRAN 77, ML
1981-1985	<i>Hardware:</i> Personalcomputer; erste Workstations; Videospiele; lokale Netzwerke (LANs); Arpanet <i>Methoden:</i> Objektorientierte Programmierung; interaktive Umgebungen, syntaxorientierte Editoren <i>Sprachen:</i> Turbo Pascal, Smalltalk-80, Wachstum von Prolog, Ada 83, PostScript
1986-1990	<i>Hardware:</i> Alterung der Mikrocomputer; erste Engineering- Workstation; RISC-Architekturen; globale Vernetzung; Internet <i>Methoden:</i> Client/Server-Architektur <i>Sprachen:</i> FORTRAN 90, C++, SML (Standard ML)
1991-1995	<i>Hardware:</i> Sehr schnelle preisgünstige Workstations und Mikrocomputer; massive parallele Architekturen; Sprache, Video, Fax, Multimedia <i>Methoden:</i> Offene Systeme; Umgebungsrahmen; <i>National Information Infrastructure</i> (Datenautobahn) <i>Sprachen:</i> Ada 95, Prozeßsprachen (TCL, PERL),

Tabelle 1: Wichtige Einflüsse zur Entwicklung auf Programmiersprachen nach [PRA98], S 32-34.

## 1.2 Zeilen- und Symbolstruktur eines C-Programmes

Das folgende kleine C-Programm bewirkt die Ausgabe der Summe 10+12 auf dem Bildschirm:

```
#include <stdio.h>

main()
{ printf("Summe= %d \n",10+12);
  return 0;
}
```

So kurz dieses Trivialprogramm auch ist, es erlaubt doch einige wesentliche Eigenschaften der Sprache C aufzuzeigen.

Wir erkennen eine Zeilenstruktur im Programm. Die einzelnen Anweisungen im Programm werden zeilenweise notiert. Wie bei einem Blatt Papier, dessen Breite begrenzt ist, wird der Text über mehrere Zeilen formuliert.

Die Aussage, die sog. Semantik, wird durch die Zeilenstruktur, bis auf wenige Ausnahmen, nicht beeinflusst. Wir könnten unser Summenprogramm also auch so schreiben:

```
#include <stdio.h>
main(){printf("Summe= %d \n",10+12);return 0;}
```

Obwohl auch dieses Programm syntaktisch korrekt ist, hat es massiv an Übersichtlichkeit verloren. Für C-Programme wählt man deshalb eine Darstellung, die leicht lesbar ist.

Eine weitere Feststellung ist, dass das Programm aus einer Menge von Symbolen besteht. Ein Symbol kann sein:

- ein (reserviertes) Wort (Bsp.: main)
- eine Zahl (Bsp.: 10)
- eine Textkette (Bsp.: "Summe = %d \n")
- ein Spezialzeichen (Bsp.: ; oder {})

Aus dem Beispiel können wir uns für später bereits merken:

1. Ein Symbol muss immer zusammengeschrieben werden. Es darf also keine Leerschläge enthalten und darf nie über zwei Zeilen verteilt werden.
2. Zwischen Symbolen darf beliebig viel Platz sein. Unter Platz, dem *white space*, verstehen wir Leerschläge, Tabulatoren oder Leerzeilen.

Wir werden diese etwas vagen Formulierungen später präzisieren.

### 1.3 Darstellung der Syntax

Programmiersprachen sind so genannte **formale Sprachen**. Formale Sprachen sind kontextfreie Kunstsprachen die eine genau definierte Syntax haben. **Kontextfreiheit** besagt, dass die Semantik nicht von der Syntax abhängig ist.

Allgemein stellen Programmiersprachen bezüglich der syntaktischen Formulierung hohe aber klare Anforderungen. Eine Redundanz wie in der menschlichen Sprache, wo die Bedeutung eines Satzes nach Weglassen eines oder mehrerer Worte immer noch klar ist, gibt es in diesen formalen Sprachen nicht.

Zur Definition der Syntax verwenden wir Syntaxdiagramme. Sie wurden von J. Backus und P. Naur zur Definition der Sprache ALGOL 60 verwendet. Syntaxdiagramme können grafisch (wie hier verwendet) oder textuell dargestellt werden.

Zur Illustration der Syntaxdiagramme wollen wir die Bedienung eines einfachen Taschenrechners mit den Grundrechenoperationen definieren:

Als erstes definieren wir eine ganze Zahl, die eingetippt werden kann:

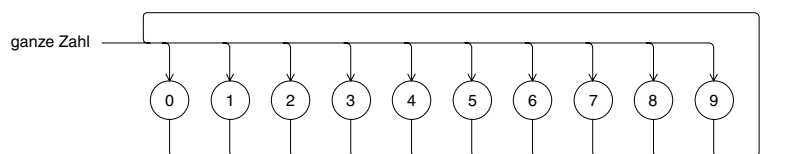


Bild 1-6: Syntaxdiagramme für eine ganze Zahl.

Die Kreise bedeuten '**atomare Elemente**'. Sie sind die Grundbausteine für die gesamte Syntax. Alle komplexeren Ausdrücke lassen sich in solche atomare Bausteine zerlegen (rekursive Definition). Die Rückführung am Schluss bewirkt eine Wiederholung falls das syntaktische Element noch fertig analysiert wurde, beispielsweise erst eine Ziffer einer dreistelligen Zahl bearbeitet.

Eine 'ganze Zahl' ist also gemäss Diagramm eine Ziffernfolge von 1..n Ziffern '0'..'9'.

Wir durchlaufen übungshalber das Diagramm für die Zahl 124. Dazu nehmen wir zuerst die Ziffer 1 und laufen durch das Diagramm:

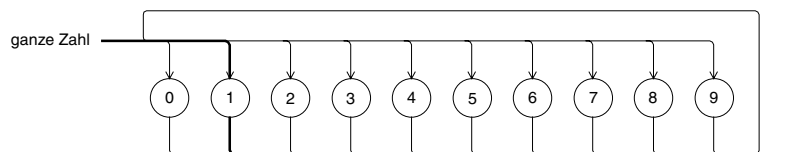


Bild 1-5: Prüfung der Zahl 124 anhand des Sytaxdiagrammes beginnend mit Ziffer 1.

Da noch nicht alle Ziffern der Zahl bearbeitet wurden, kehren wir über die Verzweigung an den Eingang zurück und wiederholen dir Durchläufe für die beiden anderen Ziffern.

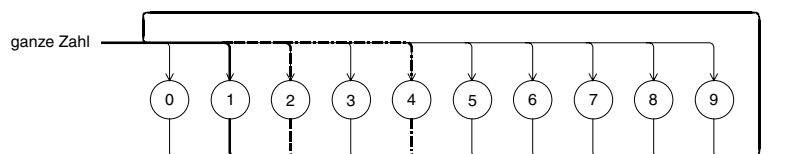


Bild 1-7: Prüfung der Zahl 124 anhand des Syntaxdiagrammes für die restlichen Ziffern 2 und 4.

Sind wir für alle Ziffern der Zahl konfliktfrei durch das Diagramm gegangen, so haben wir Gewähr dass 124 eine korrekte Zahl gemäss der syntaktischen Definition ist.

Dieses Verfahren zur Prüfung mag kompliziert erscheinen. Es ist aber relativ einfach zu implementieren weil es hierarchisch sauber gegliedert und somit übersichtlich ist. Alle gängigen Compiler führen die Syntaxprüfung aufgrund solcher Definitionen durch, so ist es wertvoll wenn man dieses Konzept verstanden hat.

**Aufgabe:** Versuchen Sie dasselbe mit 0x23 und 3.14

Für beide Zahlen findet man keinen Weg durch das Syntaxdiagramm 'ganze Zahl'. Somit sind beide keine gültigen ganzen Zahlen im Sinne der syntaktischen Definition.

Wir gehen einen Schritt weiter und betrachten das Syntaxdiagramm für eine beliebige Zahl in unserem Taschenrechner:

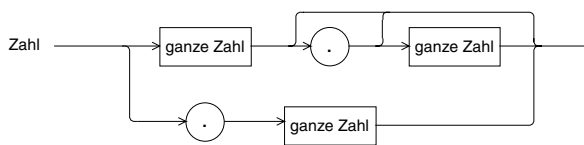


Bild 1-8: Syntaxdiagramm für eine Gleitkomma- oder eine ganze Zahl.

Die Rechtecke besagen, dass hier das entsprechend benannte Syntaxdiagramm (als Subroutine) durchlaufen wird. Wir haben somit eine **hierarchische Syntaxdefinition**.

Beurteilen Sie nun welche der folgenden Konstruktionen gültige Zahlen, gemäss unserem Syntaxdiagramm 'Zahl' sind:

35	38a7
621.007	43.
43.0	.5
20.10.1994	123456789012345656778.123
19,72	52=4
0.3	3BC

Zur vollständigen Beschreibung des Taschenrechners müssen nun noch die Operationen definiert werden, d. h. die Syntax einer ganzen Rechnung:

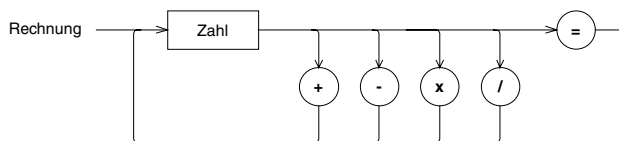


Bild 1-9: Syntaxdiagramm für eine einfache Rechnung.

Das Diagramm besagt, dass die Rechnung aus einer Zahl, gefolgt von einer beliebig langen Sequenz von jeweils einer Rechenoperation und einer Zahl besteht. Die Rechnung wird immer durch Drücken der =-Taste abgeschlossen.

## 1.4 Syntaxdiagramme für C

Syntaxdiagramme zur Sprachdefinition in C sind in gängigen Lehrbüchern nicht unbedingt üblich. Viele Sprachen z.B. ALGOL und PASCAL wird ausschliesslich mit Syntaxdiagrammen gearbeitet. Dies ist (vermutlich) in den Wurzeln der Sprache begründet: C ist eine Sprache zur effizienten Implementierung von Systemsoftware, die aus Bedürfnissen von Profis entstanden ist. Diese Leute waren ausgebildete Informatiker und hatten diese Konzepte selbstverständlich intuitiv. PASCAL hingegen ist von der Idee her gesehen eine Lernsprache für angehende Informatiker, die dazu entwickelt wurde um Algorithmen möglichst sauber strukturiert zu codieren. Also zwei ganz verschiedene Bedürfnisse.

Wir setzen voraus, dass mit Syntaxdiagrammen gearbeitet werden kann. So sollte es möglich sein aufgrund eines Syntaxdiagrammes einen Algorithmus zu entwerfen, der prüft, ob eine Gleitkommazahl gültig ist.

Ein vollständiges Syntaxdiagramm zu C nach Kernighan & Ritchie liegt dem Buch [KER83] als Faltblatt in Grösse DIN A2 bei.

## 2 Konzept der Sprache C

Die Sprache C selbst ist relativ einfach, obwohl das auf den ersten Blick nicht gerade so erscheint. Dies begründet darauf, dass die Sprachdefinition selbst recht einfach ist. So hatte C in der Urversion nur 13 Befehle. Diese Befehle dienen hauptsächlich dazu elementare Datentypen zu definieren und den Programmablauf zu steuern. Alles andere wird über den sog. **Preprozessor** und **Library-Funktionen** realisiert.

Der Preprozessor ist ein historisches 'Überbleibsel' der Sprache C. Er ist der erste Teil des Kompilationsvorganges und hat mit der Sprache C nicht direkt etwas zu tun. So gelten für den Preprozessor kurioserweise auch andere Syntaxregeln. Der Preprozessor führt die an ihn gerichteten Anweisungen aus. Preprozessoranweisungen beginnen immer mit einem Doppelkreuz-Zeichen (#). Der Preprozessor wird in Kapitel 11 detailliert behandelt.

Die sog. **Library Funktionen** sind tragendes Element der Sprache C. Die Sprachdefinition von C verfügt über keine I/O-Funktionen, komplexere Operationen oder Funktionen. Dies würde die Sprache nur unnötig aufblähen. Alle diese fallweise dennoch benötigten Funktionen sind in der sog. **Runtime-Library** abgelegt. Diese Library ist sehr umfangreich. Fast für jedes Bedürfnis kann eine entsprechende Funktion gefunden werden.

Ein klassischer C-Compiler besteht also aus 3 Teilen:

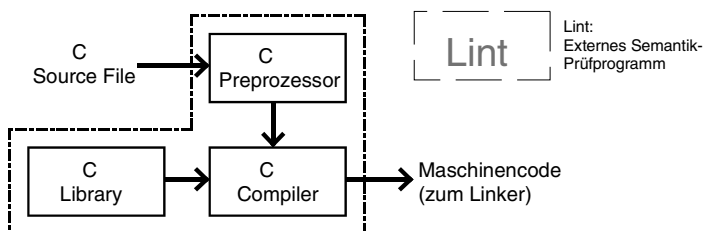


Bild 2-1: Funktionsblöcke des C-Compilers.

Diese Dreiteilung ist in modernen Compilern immer noch genauso vorhanden. In einer integrierten Arbeitsoberfläche ist dies aber nicht mehr unbedingt ersichtlich. **Lint** gehört heute normalerweise nicht mehr zum Lieferumfang von C/C++ Entwicklungsumgebungen, weil eine Typen- und Semantikprüfung von jedem C++ Compiler durchgeführt wird.

Für tiefgehende Semantikprüfungen und portabilitätsaspekte wird Lint immer noch eingesetzt. Es wird von mehreren Herstellern angeboten. Ein Standard ist meines Wissens „PC-Lint“ von Gimpel Software..

Das Compilieren eines C-Programmes erfolgt dann also in drei Stufen:

Preprozessor -> C-Compiler -> Linker

Gründe für dieses aufgesplittete Konzept waren Speicherplatzprobleme auf den ersten Maschinen, wo C implementiert wurde. Dies waren PDP-11 Rechner mit 16KB Speicher. Deshalb wurde alles weggelassen, was nicht unbedingt nötig war. So verfügten die ersten Compiler über praktisch keine Typenprüfung bezüglich Datentypen und andere Komfortfunktionen. Das brachte der Sprache C einen etwas strengen Ruf ein: 'C liefert immer ein Resultat, aber oft das falsche'.

Diese eher humoristischen Aussagen haben heute keine Gültigkeit mehr. C wurde stark weiterentwickelt und in den Jahren 1983-88 durch ANSI genormt. Dieser Normung verdankt C sicher den grossen Verbreitungsgrad im technisch/ wissenschaftlichen- sowie Systembereich.

1986 wurde C durch B. Stroustrup (AT&T) nochmals im Sprachumfang erweitert und als C++ zur

Normung vorgeschlagen. Die wichtigsten neuen Eigenschaften sind sicher die objektorientierten Fähigkeiten. C++ liegt heute in der dritten Revision vor und es ist anzunehmen, dass diese Definition als Standard ca. 1996 verabschiedet wird.

Wir werden (müssen) uns in der Einführung auf ANSI-C resp. nicht objektorientierten Erweiterungen von ++ beschränken.

## 2.1 Erstes Beispiel

Wir wollen nun ein Minimalprogramm erstellen, welches folgende Ausgabe auf den Bildschirm bewirkt:

```
Hallo, mein erstes C-Programm
```

C selbst kennt aus der Sprachdefinition keine Ein- und Ausgabefunktionen (I/O). Dies ist kein Nachteil, da I/O's immer maschinen-, resp. architekturenspezifisch sind. Aus diesem Grund sind die maschinenabhängigen Teile von der Sprache abzukoppeln und nur diejenigen Teile in die Sprachdefinition eingebunden, die auf allen Plattformen verfügbar sind. So ist ein C-Programm prinzipiell auf allen Maschinen (8-Bit Mikrocontroller bis zur Supercomputer) verfügbar. I/O-Funktionen sind zwar nicht Bestandteil der Sprache C, sind aber dennoch als Library-Funktionen standardisiert. So werden I/O's auf allen Systemen in C genau gleich codiert.

Die I/O-Funktionen sind in der sog. **Runtime-Library** abgelegt. Die Runtime-Library stellt alle Funktionen der Sprache C zur Verfügung. Aus dieser umfangreichen Bibliothek werden während des Linkvorganges die tatsächlich benötigten Funktionen extrahiert und zum Programm dazugebunden. Der Linkprozess läuft in Wirklichkeit etwas komplizierter ab. Dies ist aber für unsere Arbeit nicht weiter von Interesse.

Die Standard-Ausgabefunktion ist `printf()` (Print to File). Diese Funktion schreibt sequenziell Zeichen auf den Bildschirm (Konsolenfenster):

```
printf("Hallo\n");
```

Mit dieser Anweisung der **String** `Hallo` auf den Bildschirm ausgegeben. Strings sind Textzeichenketten. Das `\n` ist eine sog. **Escape-Sequenz**. Diese bewirkt die Ausgabe eines Zeilenvorschubes auf die neue Zeile.

Library-Funktionen werden immer mit `#include` zu Beginn deklariert werden. Für `printf()` ist dies `stdio.h` (Standard I/O):

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

Das Programm selbst besteht aus der (Haupt-) Funktion `main()`. Der Programmstart beginnt immer bei `main()`. `main()` ist somit ein reservierter Funktionsname. Der Code selbst, wird gemäss Syntaxdiagramm im Funktionskörper, welcher durch `{ }` begrenzt ist, eingebettet.

Fertiges Programm:

```
#include <stdio.h>
main()
{ printf("Hallo, mein erstes C-Programm\n");
  return 0;
}
```

## 2.2 Grundsätzlicher Aufbau von C-Programmen

Wir betrachten vorab in einer Übersicht den grundlegenden Aufbau von C/ C++ -Programmen. Eine Übersicht deshalb, damit möglichst rasch konkret, d. h. mit Beispielen, gearbeitet werden kann. Dies hat natürlich zur Folge, dass vorab nicht jedes Detail in voller Tiefe behandelt werden kann. Die wesentlichen Begriffe werden jedoch später präzisiert.

Wir gehen von der Syntaxdefinition eines C/ C++ Programmes aus:

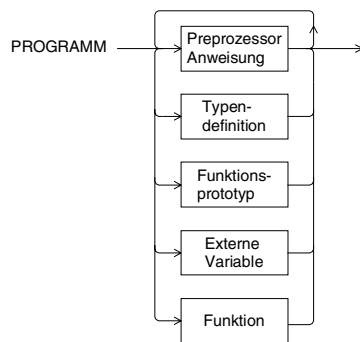


Bild 2-2: Vereinfachtes Syntaxdiagramm eines C-Programmes.

Wir sehen bereits hier den freien Programmaufbau unter C. So ist keine feste Reihenfolge der obigen Blöcke vorgeschrieben. Ebenso können Blöcke fehlen. Für ein lauffähiges Programm muss nur mindestens eine Funktion mit dem Namen `main()` existieren.

C- Programme bestehen normalerweise aus einer Vielzahl von **Funktionen**. Eine Funktion wird folgendermassen definiert:

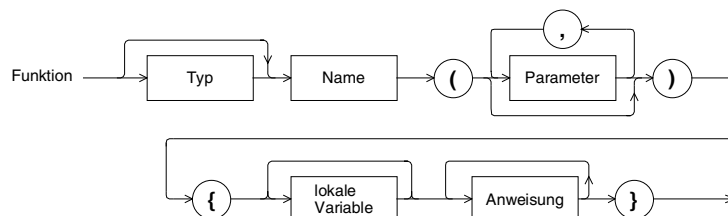


Bild 2-3: Syntaxdiagramm einer Funktion in C/ C++.

In C/C++ ist der Standarddatentyp für eine Funktion `int`. Dieser Typ wird für das Resultat angenommen, wenn kein expliziter Typ dafür angegeben wird. Der Name der Funktion ist frei wählbar mit der Bedingung, dass es ein gültiger Name im Sinne der C-Syntax ist.

Die **Parameter**, also Argumente an die Funktion werden in runden Klammern notiert. Falls die Funktion keinen Parameter hat, müssen die Klammern gleichwohl aufgeführt werden, sowohl bei der Funktionsdefinition wie auch beim Aufruf!

Der **Funktionskörper** folgt als Block, eingekleidet mit `{ }`-Klammern. Diese Klammern werden auch **Blockklammern** genannt. Im Block werden zuerst, falls nötig, die **lokalen Variablen** definiert, dann folgt der eigentliche Funktionscode. Mit der Einführung von C++ können Variablen an beliebiger Stelle definiert werden, trotzdem sollten die Definitionen aus Gründen der Übersichtlichkeit zu Beginn eines Blockes erfolgen.

### 3 Einfache Datenein- und Ausgabe in C

Obwohl die Ausgabefunktion vom Thema her in das Kapitel der Files gehört, ist es selbst für einfache Programme unerlässlich über ein Grundwissen zu verfügen. Deshalb wird das Prinzip der Datenein- und Ausgabefunktionen `scanf()` und `printf()` vorgezogen. Die Sprache C++ verwendet mit den Streams ein grundsätzlich anderes Ein- Ausgabekonzept. Das ist einer der grossen Unterschiede zwischen C und C++.

Alle Datenein- und Ausgaben erfolgen in C über **Datenkanäle** (Files). Beim Programmstart werden die **Standardfiles** `stdin`, `stdout` und `stderr` automatisch bereit gestellt. Ohne weitere Vorkehrungen können nachher Programm Daten von der Tastatur eingelesen werden und Datenausgaben auf den Bildschirm erfolgen.

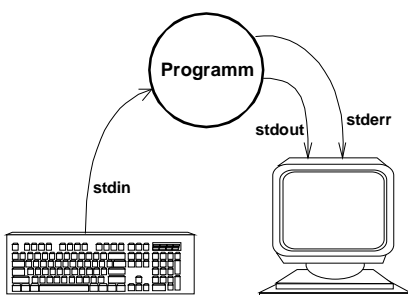


Bild 3-1: Standard Ein- und Ausgabekanäle in C.

Das Standardfile `stdin` stellt den Dateneingabekanal bereit. Die Datenausgabe zum Bildschirm erfolgt über `stdout`. Beide Kanäle arbeiten mit gepufferter Datenübertragung, d.h. die Ein- und Ausgaben erfolgen über einen Zwischenspeicher. Der Kanal `stderr` ist Standard-Fehlerausgabekanal. Er ist ungepuffert und kann nicht umgeleitet werden. Die Kanäle `stdin` und `stdout` können von der Kommandozeile her mit den Piping-Operatoren (`<`, `>`) umgeleitet werden.

#### 3.1 Datenausgabe mit `printf()`

Die Ausgabefunktion `printf()` ist die universelle Ausgabefunktion für die **formatierte Ausgabe** von Datenwerten auf `stdout`. Durch die vielfältigen **Formatschlüssel** sind umfangreiche Arten der Formatierung von Text und Datenwerten möglich.

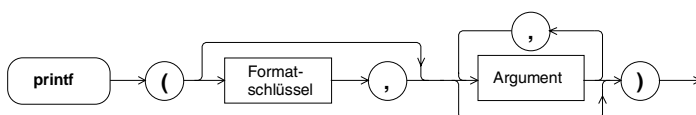


Bild 3-2: Syntaxdiagramm für `printf()`-Funktion.

Der Formatschlüssel definiert die Darstellung der auszugebenden Grössen. Er wird durch einen Textstring mit Formatieranweisungen dargestellt. Die Argumente sind die eigentlich auszugebenden Grössen. Sie werden gemäss Formatieranweisung in die Ausgabe eingebettet. Ausgegeben werden können alle Systemdatentypen und Strings in Form von Null-terminierten Character-Arrays.

Bis jetzt haben wir mit `printf()` nur eine Textzeile auf dem Bildschirm ausgegeben:

```
printf("Mein erstes C-Programm\n");
```

Die Funktion `printf()` erhält hier als Argument den String "Mein erstes C-Programm\n". Dieser String kann auch **ASCII-Steuerzeichen** beinhalten. Diese werden in spezieller Weise codiert. Steuerzeichen, sog. Escape-Sequenzen, werden mit dem `\` (Backslash) eingeleitet:

Steuerzeichen	Bedeutung
\a	Klingelzeichen (bell)
\b	Rückschritt (backspace)
\f	Seitenvorschub (form feed)
\n	Zeilenwechsel (new line)
\r	Wagenrücklauf (carrige return)
\t	Tabulator (horizontal tabulator)
\v	Vert. Tabulator
\\	Backslash-Zeichen
\'	Hochkomma
\"	Anführungszeichen
\?	Fragezeichen
\xnnn	ASCII-Wert hexadezimal
\nnn	ASCI-Wert oktal

Tabelle 2: Steuerzeichen in C/C++.

Die Steuerzeichen werden direkt in den String eingebracht. So wird die Ausgabe des Textes:

```
Dies ist Zeile 1          mit Tabulator  
"Dies ist Zeile 2 in Anführungszeichen"  
Durch den String
```

```
"Dies ist Zeile 1\tmit Tabulator\n\"Dies ist Zeile 2 in Anführungszeichen\"\n"
```

wird die Ausgabe in dieser Form bewirkt.

Der **horizontale Tabulator** \t bewegt den Schreibcursor in die Tabulatorstopp-Position. Den **vertikalen Tabulator** \v ist zwar ein gültiges Steuerzeichen, jedoch verstehen es nur die wenigsten Peripheriegeräte. (Es ist ein Relikt aus der Zeit der Fernschreibterminals.)

Der **Wagenrücklauf** \r bewegt den Cursor auf Spalte 1 der aktuellen Zeile. Wurde diese Zeile bereits beschrieben kann ein Überschreiben erfolgen.

Der **Zeilenwechsel** \n bewegt den Cursor in Spalte 1 der nächsten Zeile. Dies ist das übliche Zeichen, um auf die nächste Zeile zu wechseln.

**Backspace** \b bewegt der Cursor um ein Zeichen zurück. So kann beispielsweise das letzte ausgeschriebene Zeichen nochmals überschrieben werden. Bei Interaktionen mit dem Benutzer kann diese Möglichkeit praktisch sein, wenn man die jeweils letzte Eingabe einfach überschreiben kann.

Der **Seitenvorschub** /f bewirkt beim Drucker die Ausgabe auf einer neuen Seite fortzusetzen. Bei Bildschirmausgaben ist dieses Zeichen wirkungslos, hier muss das Löschen über eine spezielle Funktion erfolgen.

Das **Klingelzeichen** \a bewirkt die Ausgabe eines Tones auf dem eingebauten Lautsprecher im PC. Die Tonausgabe ist aber nicht in allen Umgebungen unterstützt. So kann in Visual C++ keine Tonausgabe mittels \a erfolgen.

Soll das **Backslashzeichen** selbst oder ein **Anführungszeichen** in den String eingebunden werden, so erfolgt dies durch \\ resp. \".

### 3.2 printf() Formatspezifikationen

Die wahre Stärke der printf-Funktion liegt aber in der formatierten Ausgabe von Daten. Über den Formatschlüssel, der im String angegeben wird, können auszugebende Datenwerte sehr komfortabel ausgerichtet und in den verschiedensten Darstellungen ausgegeben werden.

Formatspezifikationen sind Platzhalter im String, an deren Stelle bei der Ausgabe der Datenwert eingesetzt wird. Sie beginnen immer mit % und können für alle Ganzzahl, Gleitkomma und Textausgaben verwendet werden.

Obwohl der Zahl der Argumente zum Zusammenstellen der Formatschlüssel relativ klein ist, sind verschiedenste Kombinationen möglich und üblich. Wir werden die am häufigsten verwendeten kurz betrachten. Für den Rest sei auf die Referenz (rosa Blatt verwiesen).

Beispiel:

```
printf("Dies ist Kapitel %d",4);
```

Die Formatspezifikation ist hier %d. %d ist ein 'Platzhalter' für die Ausgabe einer vorzeichenbehafteten dezimalen Ganzzahl (int). Bei der Ausgabe wird also genau an der Stelle wo '%d' im String steht der Zahlenwert eingesetzt.

Ein etwas komplexeres Beispiel zum Verständnis:

```
main ()  
{ printf("%d + %d = %d",2, 3, 2+3);  
  return 0;  
}
```

Dies ergibt die Bildschirmausgabe:

2 + 3 = 5

In diesem Beispiel stehen im String lediglich Leerschläge, das Additionszeichen, das Gleichheitszeichen sowie den Formatschlüsseln, jedoch keine Rechnung in irgendwelcher Art.

Wie wir im Beispiel sehen, kann printf nicht nur Werte, sondern auch Ergebnisse von Operationen ausgeben, wie hier die Summe 2+3:

Generell werden Argumente in der gleichen Reihenfolge eingesetzt wie sie als Formatschlüssel im String erscheinen:

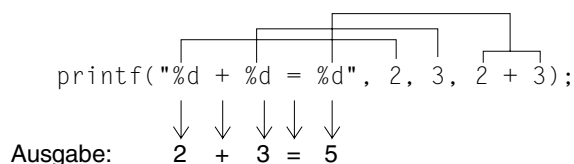


Bild 3-3: Assoziation der Formatschlüssel zu den Argumenten.

Gleitkommawerte können nicht mit %d ausgegeben werden. Ein guter Compiler wird beim Kompilieren hier eine Fehlermeldung ausgeben. Ältere Produkte geben einfach ein falsches Resultat auf den Bildschirm.

Die Ausgabe von Gleitkommawerten (`float`, `double`) erfolgt mit `%f`, `%g` für Fixkommadarstellung und `%e` für Exponentendarstellung. Der Formatschlüssel `%f` gibt den Wert immer in Fixkommadarstellung aus.

```
main()
{ float a = 3.3;

  printf("float Datenwerte in verschiedener Darstellung:\n%f\t%e\t%g", a, a, a);
  return 0;
}
```

float Datenwerte in verschiedener Darstellung:

3.300000 3.300000e+000 3.3

Die Feldbreite besagt wie viel Platz im String für die auszugebende Zahl mindestens reserviert wird. Wird keine Feldbreite vorgegeben, so werden standardmässig 6 Nachkommastellen gerundet ausgegeben. Ebenso kann die Präzision (Anzahl Nachkommastellen) spezifiziert werden:

```
main()
{ float a=3.35123;

  printf("%3.2f\t%3.1f\t%3.1e\t%3.1g", a, a, a, a);
  return 0;
}
```

3.35 3.4 3.4e+000 3

Ferner sind für Ganzzahlen auch Ausgaben in oktaler oder hexadezimaler Basis möglich. Dies wird mit den Schlüsselwörtern `%x`, `%X` resp. `%o` erreicht. Die grossgeschriebene Variante zeigt die Hexzahlen a..f in Grossbuchstaben während `%x` die Buchstaben in Kleinschrift ausgibt:

```
main()
{ int i;

  for (i=0; i<=15;i++)
    printf("Dezimal: %d\tOktal: %o\tHex (gross): %X\tHex(klein): %x\n", i, i, i, i);
  return 0;
}
```

Dezimal: 0	Oktal: 0	Hex (gross): 0	Hex(klein): 0
Dezimal: 1	Oktal: 1	Hex (gross): 1	Hex(klein): 1
Dezimal: 2	Oktal: 2	Hex (gross): 2	Hex(klein): 2
Dezimal: 3	Oktal: 3	Hex (gross): 3	Hex(klein): 3
Dezimal: 4	Oktal: 4	Hex (gross): 4	Hex(klein): 4
Dezimal: 5	Oktal: 5	Hex (gross): 5	Hex(klein): 5
Dezimal: 6	Oktal: 6	Hex (gross): 6	Hex(klein): 6
Dezimal: 7	Oktal: 7	Hex (gross): 7	Hex(klein): 7
Dezimal: 8	Oktal: 10	Hex (gross): 8	Hex(klein): 8
Dezimal: 9	Oktal: 11	Hex (gross): 9	Hex(klein): 9
Dezimal: 10	Oktal: 12	Hex (gross): A	Hex(klein): a
Dezimal: 11	Oktal: 13	Hex (gross): B	Hex(klein): b
Dezimal: 12	Oktal: 14	Hex (gross): C	Hex(klein): c
Dezimal: 13	Oktal: 15	Hex (gross): D	Hex(klein): d
Dezimal: 14	Oktal: 16	Hex (gross): E	Hex(klein): e
Dezimal: 15	Oktal: 17	Hex (gross): F	Hex(klein): f

Die Funktion `printf()` liefert als Funktionswert die Anzahl ausgegebener Zeichen als `int`-Grösse zurück. Normalerweise ist dieser Rückgabewert nicht von Interesse. Erfolgt aber eine Umleitung des Standard-Ausgabekanals `stdout` z.B. auf einen Drucker oder Disk, kann dies zur Fehlererkennung benutzt werden.

### 3.3 Spezifikationen für die Feldbreite und Genauigkeit

Wir haben bereits gesehen, dass im Formatschlüssel auch die Feldbreite spezifiziert werden kann. Die Feldbreite definiert wie viel Platz für die auszugebende Zahl inklusive eventueller Vorzeichen und Exponenten zur Verfügung gestellt wird.

Die Feldbreite: wird generell durch die ganze Zahl die nach dem % Zeichen folgt definiert. Sie gibt an wie viele Zeichen (Spalten) für das auszugebende Zeichen reserviert werden.:

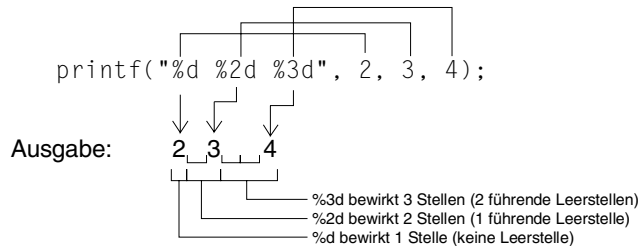


Bild 3-4: Feldbreiten bei Formatschlüssel.

#### Achtung

Benötigt der auszugebende Wert mehr Platz als die Feldbreite zur Verfügung stellt, wird dieser automatisch genommen. `printf` kennt keine Definition einer maximalen Feldbreite. Die Feldbreite ist daher immer als Mindestfeldbreite zu verstehen.

Die Feldbreitenspezifikation kann man für Ganzzahlwerte wie auch für Gleitkommawerte benutzen. Gleitkommawerte können zusätzlich in ihren Nachkommastellen spezifiziert werden.

Die Feldbreite ist die Mindestbreite für die gesamte auszugebende Zahl. Die zweite Zahl nach dem Punkt gestattet es die Anzahl Dezimalstellen festzulegen. Wird die Ausgabe beschnitten, so erfolgt immer eine Rundung auf die entsprechende Anzahl Stellen.

Beispiel:

```
main()
{ printf("%6.2f %6.4f %8.4f", 12.3456, 12.345, 12.3456);
  return 0;
}
```

12.35 12.3450 12.3456

Es ist zu beachten:

Die Feldbreite umfasst alle Zeichen, die zur Darstellung der Zahl benötigt werden. Also auch ev. Vorzeichen und den Dezimalpunkt.

Wir analysieren die Ausgabe im obigen Programm:

Wert	Spezifikation	Ausgabe	Kommentar
12.3456	%6.2f	12.35	Der Wert wird auf 2 Dezimalstellen nach dem Komma gerundet.
12.345	%6.4f	12.3450	Null wird hinzugefügt, wegen der Genauigkeit 4. Die Mindestbreite (6) wird überschritten.
12.3456	%8.4f	12.3456	Ein führendes Leerzeichen zum Erreichen der Mindestbreite wird eingefügt.

Bei Ganzzahlen wird anstatt der Präzision die Anzahl führende Nullen zum Auffüllen des Feldes angegeben. Häufig ist jedoch die Angabe einer führenden Null in der Feldbreite. Dies bewirkt ein Auffüllen mit führenden Nullen bis zur Mindestfeldbreite.

```
main()
{
    printf("%5.3d %5.5d %05d", 3, 3, 3);
    return 0;
}
```

003 00003 00003

Printf zeigt das Vorzeichen nur an wenn es negativ ist. Soll das Vorzeichen immer angezeigt werden, kann dies mit dem Pluszeichen in dem Formatschlüssel erreicht werden:

```
main()
{
    printf("%+3d %+3d", -7, 5);
    return 0;
}
```

-7 +5

Die letzte Option ist ein Minuszeichen im Formatschlüssel. Es bewirkt, dass der Wert linksbündig in das Feld eingesetzt wird:

```
main()
{
    printf("%-3d %-3d", -7, 5);
    return 0;
}
```

### 3.4 Dateneingabe mit scanf()

Die Einlesefunktion scanf() ist das Gegenstück zu printf(). Sie erlaubt das formatierte Einlesen von Text, einzelner Zeichen und Zahlen vom Standard-Eingabekanal stdin.

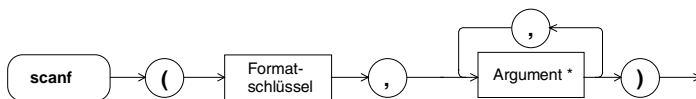


Bild 3-5: Syntaxdiagramm für scanf().

Der Formatschlüssel beschreibt das Format und den Typ der einzulesenden Größen. Einlesbar sind alle Systemdatentypen und Textstrings. Bei Gleitkommazahlen ist auch Exponentialdarstellung erlaubt. Werden mehrere Größen gleichzeitig eingelesen, müssen sie durch eine White Space getrennt sein. Obwohl die Formatieranweisungen praktisch gleich wie bei printf() sind, können über den Formatschlüssel keine Textausgaben erfolgen.

Die Argumente bezeichnen die Speicherstelle, wo die eingelesene Größe abgespeichert werden soll. Dies wird mit einer Zeigerreferenz angegeben. Der Zeiger beschreibt die Speicherplatzadresse der Variablen.

Die Funktion liefert als Funktionswert eine Ganzzahl als Resultat. Sie zeigt an, wie viele zu den Formatschlüsseln passende Größen eingelesen werden konnten.

Beispiel:

```
#include <stdio.h>

int main()
{
    int count;
    double gleitkommaZahl;
    char string[30];
    int ganzzahlWert;

    printf("Bitte Gleitkommazahl, String und Ganzzahl eingeben: ");
    count=scanf("%lf%s%d",&gleitkommaZahl,string,&ganzzahlWert);

    printf("Eingelesen wurden %d Groessen:\n%f %s d",count,gleitkommaZahl,string,ganzzahlWert);
    return 0;
}
```

Ein Programmlauf zeigt beispielsweise:



Die Gleitkommazahl wurde in diesem Beispiel mit %lf eingelesen, aber mit %f ausgegeben. Das ist korrekt, denn mit %f kann eine doppelt genaue Gleitkommazahl zwar ausgegeben, aber nicht eingelesen werden.

## 4 Datentypen und Variablen

Datenverarbeitung und somit Programmierung allgemein, beschäftigt sich mit dem systematischen Verarbeiten von Daten. Die Daten werden in Programmen in Variablen gehalten. Diese Variablen sind typisiert, d.h. jede Variable kann nur Daten eines bestimmten Typs aufnehmen. So gibt es Variable die können nur Ganzzahlen aufnehmen, andere für Gleitkommazahlen und andere für ASCII-Zeichen.

Diese Einschränkung bezüglich Typ ist sinnvoll da intern in der Maschine jeder Datentyp andere Vorschriften zu Verarbeitung der Daten erfordert. So erfolgt intern eine Addition von zwei Ganzzahlen absolut anders als die Addition von zwei Gleitkommazahlen, obwohl nachher vom numerischen Wert dasselbe Resultat erscheint.

Ohne uns näher mit diesen Hintergründen zu beschäftigen, möchten wir den Begriff des Datentyps charakterisieren als Sorte von Daten, die in eine Variable untergebracht werden können. Somit ist der Begriff Datentyp sehr eng mit dem Begriff Variablen assoziiert.

### 4.1 Definitionen

Wir wollen nachfolgend einige Begriffe definieren, die später mehrfach verwendet werden. Sie gehören zum normalen Sprachschatz des Programmierers.

**Variable:** Abstraktes Element (sog. Datenobjekt) dem während des Programmlaufes einen Wert zugewiesen und belesen werden kann. Variablen haben eine strikte Typenbindung an den Datentyp auf den sie definiert worden sind. Jede Variable hat einen Namen (Identifizier) und einen Wert (Value).

**Datentyp:** 'Sorte' der Daten, die in Variablen untergebracht werden können. Wir unterscheiden zwischen Systemdatentypen, die immer standardmässig verfügbar sind (int, char, float, double) und benutzerdefinierten Datentypen wie Array und Struct's, etc.  
Funktionen sind auch an einen Datentyp gebunden. Sie beziehen sich auf den Typ des Resultates, das sie zurückliefern.

**Geltungsbereich: (Scope)** Jede Variable hat einen gewissen Geltungsbereich. Je nach Sorte der Variablen kann sie in gewissen Programmabschnitten gesehen (d.h. zugegriffen) werden in anderen hingegen nicht.

**Lebensdauer: (Lifetime)** Jede Variable hat eine bestimmte Lebensdauer. So gibt es Variablen die während des gesamten Programmes leben und andere, die nur in einem kleinen Block existieren.  
(Anm: Das hat nichts mit globalen / lokalen Variablen zu tun!)

**Globale Variable:** Variable, die überall im Programmmodul zugreifbar ist.

**Lokale Variable:** Variable, die nur im Block in dem sie definiert wurde zugreifbar ist.

## 4.2 Systemdatentypen in C

C (wie auch C++) kennt, wie die meisten Programmiersprachen, nur wenige *Standarddatentypen*. Dies ist jedoch kein Mangel, da in umfangreichen Programmen sowieso meist benutzerdefinierte Datentypen verwendet werden. Die benutzerdefinierten Typen sind aber schlussendlich aus Systemtypen zusammengesetzt, so dass sie die primitiven oder atomaren Datentypen der Sprache verkörpern.

Der Datentyp ist das Konstruktionsmuster oder -Vorschrift für eine Variable. Die Variablen verkörpern technisch gesehen Speicherplatz im RAM-Speicher des Rechners. Durch den Datentyp wird nun festgelegt, wieviel Speicherplatz für eine Variable eines Typs bei der Definition reserviert werden muss.

Systemdatentypen der Sprache C:	Bedeutung
char	Ein Byte zur Speicherung eines Zeichens
int	Ein Wort zur Speicherung einer Ganzzahl
float	Gleitkommazahl in einfacher Genauigkeit
double	Gleitkommazahl in doppelter Genauigkeit

Tabelle 3: Systemdatentypen C/C++.

Der Typ **int** speichert eine ganze Zahl. Der Wertebereich einer **int** ist aber maschinenabhängig. Normalerweise hat **int** die Breite eines Maschinenwortes oder Prozessorregisters. So ist auf einem PC unter DOS und Windows 3.X eine **int** 16 Bit breit. Auf Win32 Plattformen jedoch 32 Bit. Die **int** ist eine vorzeichenbehaftete Zahl mit dem Wertebereich von `INT_MIN` . . `INT_MAX`. Diese Konstanten sind im Headerfile `LIMITS.H` definiert. Der Wertebereich ist  $[-2^{15}-1, 2^{15}]$  für 16 Bit und  $[2^{31}-1, 2^{31}]$  für 32 Bit Wortbreite angenommen werden.

Eine **float** speichert eine Gleitkommazahl mit einfacher Genauigkeit. Der Wert wird nach IEEE-Norm in Vorzeichen, Mantisse und Exponent zerlegt und in 4 Bytes abgespeichert.

Eine **double** speichert eine Gleitkommazahl mit doppelter Genauigkeit. Der Wert wird nach IEEE754-Norm in Vorzeichen, Mantisse und Exponent zerlegt und in 8 Bytes abgespeichert. **double** -Werte haben eine wesentlich grössere Präzision in der Mantisse und einen grösseren Exponenten gegenüber der **float**.

Der Typ **char** ist die kleinste adressierbare Einheit. Er besteht in der Regel aus einem Byte und dient normalerweise zur Speicherung eines Zeichens (ASCII) oder von kleinen Ganzzahlen. **char** ist eine vorzeichenbehaftete Grösse.

Der Ganzzahltyp **int** kann zusätzlich noch mit **long**, **short**, **signed** und **unsigned** modifiziert werden. Dabei wird der Wertebereich des Typs beeinflusst. Nachfolgend alle zulässigen Bezeichnungen mit ihren Äquivalenten:

```
char
signed char
unsigned char

signed, int, signed int
short, short int, signed short, signed short int
long, signed long, signed long int

unsigned, unsigned int
unsigned short, unsigned short int
unsigned long, unsigned long int
float
double
long double
```

Nicht alle Systeme und Compiler kennen den `long double`-Typ. Er ist eine mehrfachgenaue `double` und wird vor allem in internen Berechnungen verwendet. Die Werte sind 10 Byte IEEE754-codierte Gleitkommazahlen, so wie sie direkt in den Numerikprozessoren verwendet werden. Viele Rechner kennen `long double` auch unter dem Typ `extended`.

Nachfolgend eine Zusammenstellung der Wertebereiche für Systemdatentypen (Richtlinie)

Typ	Platzbedarf	Wertebereich
<code>char</code>	1 Byte	-128..+127
<code>unsigned char</code>	1 Byte	0..+255
<code>short</code>	2 Bytes	-32768..+32767
<code>int</code>	4 Bytes (auf 32Bit Masch)	$-2^{31}..+(2^{31}-1)$
<code>unsigned int</code>	4 Bytes	$0..+(2^{32}-1)$
<code>long</code>	4 Bytes	$-2^{31}..+(2^{31}-1)$
<code>unsigned long</code>	4 Bytes	$0..+(2^{32}-1)$
<code>float</code>	4 Bytes	$\pm 3.4 E \pm 38$
<code>double</code>	8 Bytes	$\pm 1.7 E \pm 308$
<code>long double</code>	10 Bytes	$\pm 3.4 E -4932.. \pm 1.1 E +4932$
<code>bool</code>	4 Bytes (int)	0 = FALSE 1= TRUE

Tabelle 4: Wertebereich und Speicherplatzbedarf von Systemdatentypen in C/C++ auf einer Win32 Plattform.

### 4.3 Definition von Variablen

Variablen werden durch Angabe des Datentyps und Namens definiert. Der Name ist ein benutzerdefiniertes Symbol und wird auch *Identifizier* genannt. Er kein reserviertes Wort der Sprache C/C++ gemäss Tabelle 5 sein (C ist eine kontextfreie Sprache). Ist der Name bereits vordefiniert, geht die vordefinierte Funktionalität im Lokalitätsblock verloren. C/C++ unterscheidet Gross- und Kleinschrift.

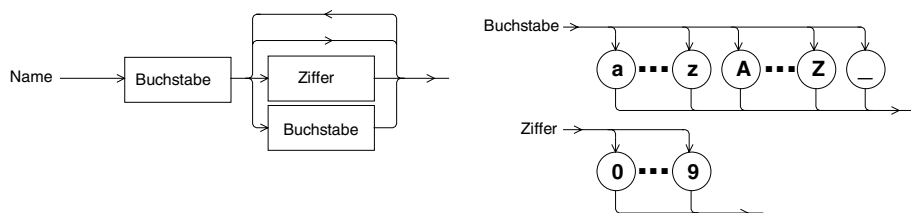


Bild 4-1: Syntaxdiagramm für ein benutzerdefiniertes Symbol. Hierunter fallen Namen für Funktionen, Variablen und Konstanten.

Beispiel:

```
int count;
float f;
char zeichen;
```

Die Regeln zur Namensbildung können dem Syntaxdiagramm entnommen werden. Die Länge ist 1 bis 31 Zeichen (minimal nach Standard). Konventionsgemäss werden Variablennamen normalerweise in Kleinschrift gehalten. Der Unterstrich sollte nach Möglichkeit vermieden werden. Für selbst erklärende Variablennamen sollte besser eine gemischte Klein-Grossschrift<sup>1</sup> verwendet werden, z.B. `meineErsteVariableInC`.

Unterstriche als erstes Zeichen eines Symbols oder Funktionsnamens zeigt meist, dass es sich hierbei um eine plattform- oder produktespezifische Erweiterung handelt. Sie gehört nicht zur Standardbibliothek und wird auf anderen Plattformen kaum verfügbar sein (z.B. `_kbhit()`).

#### 4.3.1 Schlüsselwörter in C

<sup>1</sup> Sprachwissenschaftler nennen diese gemischte Gross-Kleinschrift *Binnenmajuskel*. Sie wurde Anfangs der 80er-Jahre eingeführt. Das Ziel war die Überwindung des generischen Maskulinums und der sprachlichen Ungleichbehandlung von Mann und Frau. Später verselbständigte sich die Sache für die Beschreibung verschiedenster Produkte und Leistungen wie CorelDraw, Hypo Vereinsbank, etc.

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

4.3.2 Zusätzliche Schlüsselwörter in C++			
asm	export	private	true
bool	false	protected	try
const_cast	friend	public	typeid
catch	inline	reinterpret_cast	typename
class	mutable	static_cast	using
delete	namespace	template	virtual
dynamic_cast	new	this	
explicit	operator	throw	

Tabelle 5: Reservierte Wörter der Sprache C /C++.

## 4.4 Gültigkeitsbereich von Variablen

Variablen können in C immer zu Beginn eines Blockes, also nach einem { definiert werden. Bei C++ ist sogar überall im Block eine Definition möglich. So in einem Block definierten Variablen sind **lokale Variablen**. Sie sind nur in diesem Block und hierarchisch tiefer liegenden Blöcken zugreifbar :

Beispiel:

```
main()
{ int i=2;

    print("Wert i:%d",i);
    { int j=5;

        printf("Wert j: %d",j );
    }
    printf("Wert j: %d",j);
}
```

Es zeigt wie immer nach einer Blockklammer eine Variable definiert werden kann. Hier wird jedoch in der dritten printf()-Anweisung auf die im inneren Block definierte Variable j zugegriffen. Dies geht nicht, da sie nicht mehr im Gültigkeitsbereich Scope liegt. Der Compiler wird dies bereits beim Compilieren mit einer Fehlermeldung bemerken.

Generell gilt in einem hierarchisch untergeordneten Block kann auf eine im übergeordneten Block definierte Variable zugegriffen werden:

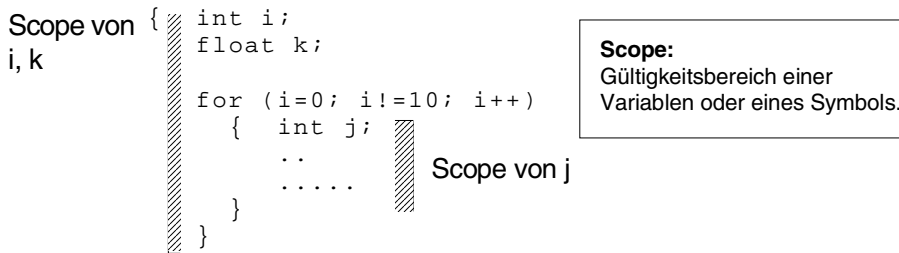


Bild 4-2: Geltungsbereich (Scope) von lokalen Größen in C/C++.

Variablen werden nach dem Verlassen des Gültigkeitsbereiches aufgelöst und sind nicht mehr existent. Eine Ausnahme bilden **static-Variablen**. Sie bleiben auch nach Verlassen des Gültigkeitsbereiches erhalten. Sie sind vom Prinzip her als eine Art „lokale Globalvariable“ zu verstehen.

Variablen können bei der Definition auch gleich Initialwerte erhalten. Dies geschieht durch direkte Zuweisung eines konstanten Wertes bei der Definition:

```
int i=0, j=5, k=-1, q=7823;
float f=0.0, a=1.3E-3, b=4.27;
char c='z', d=74;
```

Beachten Sie, dass der char-Typ nur einen Spezialfall des int-Typs darstellt. Generell sind Zeichen (ASCII) in C/C++ nur eine Untermenge der ganzen Zahlen und werden immer so behandelt.

## 4.5 Speicherklassen

Sie sind ein Präfix (Attribut) bei der Variablendefinition. Die Speicherklasse legt fest wo die Variablen im Speicher gehalten werden. Dadurch wird auch die Lebensdauer der Variablen festgelegt:

- auto**            Standard der Compiler entscheidet selbst ob er die Variable in einem CPU Register oder im Arbeitsspeicher halten soll. **auto** ist Standard für C.
- static**          Die Variable wird im Arbeitsspeicher gehalten und verliert selbst als Lokalvariable nie Ihren Wert. **static**'s werden zu Beginn des Programmes standardmässig immer mit 0 initialisiert.
- register**        Die Variable wird in einem CPU-Register gehalten. Dies erlaubt den schnellsten Zugriff auf Daten. Die meisten Compiler ignorieren aber schlichtweg diese Anweisung.(Grund: Der Compiler macht sowieso die bessere Entscheidung als Sie!)
- volatile**        Spezifiziert, eine häufig geänderte Grösse die immer im Arbeitsspeicher gehalten wird. Sie wird bei jedem Zugriff neu aus dem Hauptspeicher geladen (forced refetch). Damit werden Variablen berücksichtigt, welche nicht nur durch unser Programm verändert werden, sondern auch durch programmunabhängige Instanzen (Bsp: Interrupts, DMA, oder andere Prozesse).
- extern**          Definiert automatisch eine statische Variable. Zusätzlich wird eine externe Bindung definiert, so dass die Variable auch in anderen Modulen zugreifbar wird. **extern** definiert also eine Referenz zu anderswo definierten Variablen. Der Linker löst dann die entsprechenden Referenzen auf.

Beispiele:

```
static int i;
register long k;
volatile int v;
extern int k;

void counter
{ static int count=0;

  count++;
  print("Funktionsaufruf #: %d\n",count);
}
```

## 4.6 Interne Darstellung der Datenwerte

Obwohl wir uns selbst kaum darum kümmern müssen wie der Compiler das System (Compiler) die Organisation des Datenspeicherplatzes macht, ist es sicher sinnvoll, wenn man weiss, wie die Datentypen sich intern im Speicher präsentieren.

Jede Variable verkörpert im Prinzip eine bestimmte Menge physikalischen Speicherplatz. Die Menge (Anzahl Bytes) ist vom Datentyp abhängig. Die Werte werden entweder direkt (Ganzzahlen) oder nach einer bestimmten Codierungsvorschrift (Gleitkommazahlen) im Speicher abgelegt:

Von jeder Variablen oder Datentyp kann immer mit der `sizeof()`-Funktion die Grösse in Bytes bestimmt werden. Somit liefert auf einem PC:

```
sizeof(int)          = 2
sizeof("Hallo")     = 6  (Warum nicht 5?)
```

Nachfolgend eine Zusammenstellung der Speicherformate für Systemdatentypen in C /C++ nach [THI85], S.105 für Gleitkommaformate :

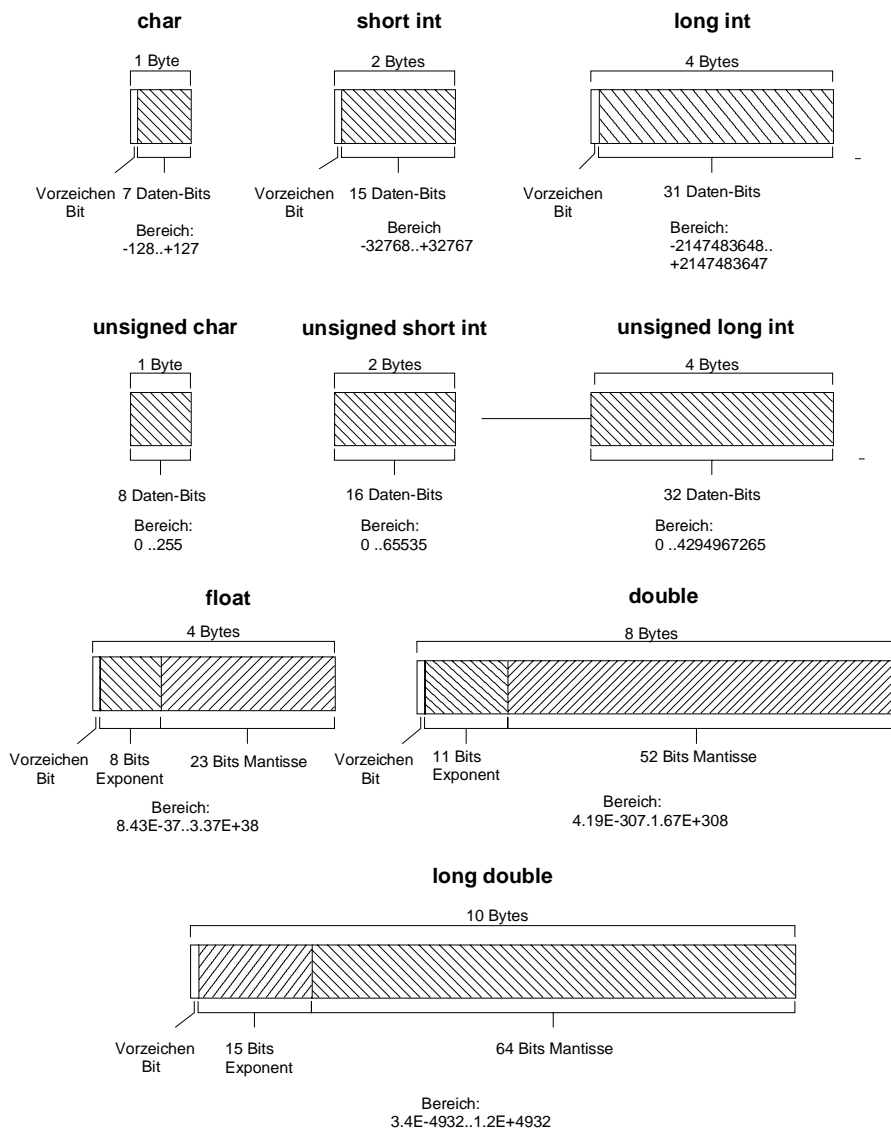


Bild 4-3: Speicherformate für Systemdatentypen in C/C++ auf PC-Plattformen.

## 4.7 Konstanten

Konstanten sind spezielle Werte im Programm. Sie verändern während der gesamten Programmausführung ihren Wert nie. Konstanten können Zahlenwerte, wie auch Zeichen oder Zeichenketten (Strings) sein.

Für Konstanten gibt es bestimmte Vorschriften wie die Werte bezüglich des Datentyps zu formulieren sind:

### **int - Konstanten**

signed:	normaler Ganzzahlwert
unsigned:	Ganzzahl, gefolgt von u oder U
long:	Ganzzahl, gefolgt von l oder L
unsigned long:	Ganzzahl, gefolgt von ul, resp. UL

Ebenso können Konstanten direkt auch oktal oder hexadezimal formuliert werden. Hex-Zahlen beginnen immer mit 0x. Das 0x ist nicht Bestandteil des Zahlenwertes selbst, sondern dient nur dazu die Zahl als hex zu identifizieren.

Beispiel:

```
0x3a
0xfd3a
0xd5a312aa
```

Nun eine C-Spezialität, die sicher noch manchem Kopfzerbrechen bei der Fehlersuche bereiten wird:

### **Oktale Zahlen beginnen immer mit einer 0!**

Sobald eine Ganzzahl mit einer 0 (Ziffer 0) beginnt, wird sie als oktale Zahl verstanden.

Beispiel:

```
010
017
```

Somit ist also 010 != 10.

### **Gleitkommakonstanten**

Gleitkommawerte können in Exponential- oder Fixkommenschreibweise notiert werden. Zulässige Zeichen hierzu sind:

```
'0'..'9', '.', '-', '+', 'e', 'E'
```

Bei diesen Konstanten muss mindestens ein . (Dezimalpunkt) in der Zahl vorhanden sein damit der Compiler dies als Gleitkommakonstante erkennt (sonst nimmt er eine int an, was im Programmlauf jedesmal zu einer Konversion führt).

Ohne weitere Zusätze werden alle Gleitkommakonstanten als double aufgefasst. Wird der Zahl ein f oder F nachgestellt wird der Wert als float aufgefasst und mit l, resp. L als long double. Der Typ long double ist zwar ein erlaubter Datentyp, ist aber synonym zu double. Dies ist zumindest bei MS-Visual C++ sowie bei Borland C++ Builder der Fall.

Beispiel:

```
1.23E3L
0.457e-14f
```

### Textkonstanten (Strings)

Strings sind Zeichenketten, also eine Folge von einzelnen Zeichen (`char`). Intern werden Strings in einem Array (Vektor, d.h. Folge von aufeinander folgenden Speicherstellen) abgelegt.

Strings werden in doppelten Hochkommata formuliert.

Beispiel: `"Dies ist ein String"`

In Strings sind alle Zeichen des auf 8Bit erweiterten ANSI-Zeichensatzes, ausser das Nullzeichen `\0` erlaubt. Strings dürfen leer `" "` sein. Die maximale Länge ist auf 16Bit Systemen 64k-1 Zeichen. Auf 32 Bit Plattformen ist die maximale Länge durch maximale Länge des Datensegmentes begrenzt.

Strings werden intern mit einem Null-Zeichen `\0` terminiert. Aufgrund dieser Terminierung weiss der C-Compiler, wo der String im Speicher zu Ende ist.

Ein leerer String `" "` besteht bereits aus einem Zeichen, dem Nullzeichen und belegt ein Byte Speicherplatz.

#### 4.7.1 Konstante Zeichen

Textzeichen (ASCII) werden normalerweise dem Datentyp `char` zugeordnet. `char` hat eine Speicherplatzbedarf von einem Byte und ist deshalb prädestiniert solche Zeichen zu speichern. Zeichenkonstanten werden immer in einfachen Hochkommata notiert:

```
char c='a';  
char d='\t';
```

Zeichen können selbstverständlich auch `long` oder `int` zugewiesen werden. Die meisten C-Compiler kennen auch 'Wide Characters'. Dies ist ein 16-Bit Datentyp (`wchar_t`), der vor allem zur Aufnahme der neuen, in Zukunft verwendeten, Codes gedacht ist (UNICODE). Dort erfolgt die Zuweisung eines Zeichens in zwei Bytes:

```
#include <stddef.h>      /* Definition fuer wchar_t */  
  
wchar_t c=L'a';  
wchar_t msg[]=L"Hallo";
```

Beachten Sie bitte generell den Unterschied bei der Zuweisung eines Zeichens (einfache Hochkommata: `' '`) und einem String (doppelte Hochkommata: `" "`).

#### 4.7.2 Konstante Variablen

ANSI-C wie auch C++ kennen das `const`-Präfix zur Definition von Konstanten. Dieses Präfix bewirkt, dass die Variable nur einmalig bei der Definition einen Wert erhalten darf, und nachher nur noch lesend zugreifbar ist. Ein Versuch eine solche konstante Variable zu beschreiben wird bereits beim Compilieren mit einer Fehlermeldung kritisiert.

Das Präfix `const` wird vor allem dazu verwendet den konstanten Charakter der Grösse zu unterstreichen und damit sicherzustellen, dass kein Überschreiben möglich ist.

Beispiel:

```
const int k=3;  
const long N=1000L;
```

Konventionsgemäss werden Konstanten in C Programmen oft in Grossbuchstaben geschrieben.

## 5 Ausdrücke und Operationen

**Ausdrücke** sind in C/C++ Konstrukte die einen Wert ergeben. Ein Ausdruck kann seinerseits wieder einen Ausdruck enthalten.

Beispiel: 
$$a + 7$$
$$x * x + 3 * x / (4 + a)$$

In Ausdrücken finden **Operationen** statt. Operationen werden mit **Operatoren** ausgeführt. C/C++ stellt einen Satz von Operatoren zur Verfügung. Diese können in Gruppen eingeteilt werden:

- arithmetische und logische Operationen
- Vergleiche
- Zuweisungen
- Funktionsaufrufe

Wesentlich ist, dass ein Ausdruck immer einen Wert (numerisch oder einen Zeigerwert) liefert. Wie in der Mathematik gelten für Operatoren auch gewisse Regeln:

- Hierarchie, Rang (precedence, rank)
- Assoziativität

### 5.1 Hierarchie

Die Hierarchiestufe, d.h. der **Rang** eines Operators besagt, welche Priorität die entsprechende Operation in einem Ausdruck hat. Aus der Mathematik wissen wir, dass beispielsweise die Multiplikation höher priorisiert ist als Addition. Da C mehr Operatoren kennt, ist die Hierarchieübersicht etwas umfangreicher.

Generell gelten für Ausdrücke in C algebraische Konventionen. Andere Operationen wie logische oder Dereferenzierung haben gewisse spezielle Regeln.

Bei gewissen Operatoren könnte man die Hierarchie intuitiv anders vermuten als sie tatsächlich ist. So ist beispielsweise die Bindekraft des \*-Operators in C schwächer als die Indizierung.

$$*b[4] \quad != \quad (*b)[4]$$

## 5.2 Operationen

C/C++ stellt standardmässig einen recht umfangreichen Satz an arithmetischen und logischen Operationen zur Verfügung. Sie sind mit gewissen Einschränkungen auf alle Systemdatentypen anwendbar.

Weitere Operationen können über Bibliotheksfunktionen (runtime library functions) durchgeführt werden.

Zusammengefasst kennt C folgende Operationen:

Operator	Beschreibung	Notation	Rang
[ ]	Arrayindex	a[b]	1
()	Klammer	(a+b)*c	1
->	Selektor	a->b	1
.	Selektor	a.b	1
-	Vorzeichen Minus	a = -b;	2
+	Vorzeichen Plus	a = +b;	2
++	Inkrement	a++ und ++a	2
--	Dekrement	a-- und --a	2
!	Logisches nicht	!flag	2
~	Bitweises Komplement	a = ~b;	2
&	Adresse	a = &b;	2
*	Dereferenzierung	a = *ptr;	2
sizeof	Speicherplatzbedarf	a = sizeof b;	2
( )	Cast(b)	(typ)a	2
*	Multiplikation	a = b * c;	3
/	Division	a = b / c;	3
%	Modulo Division	a = b % d;	3
+	Addition	a = b + c;	4
-	Subtraktion	a = b - c;	4
>>	Rechtsschieben	a = b >> c;	5
<<	Linksschieben	a = b << c;	5
>	Gösser als	a > b	6
>=	Grösser oder gleich	a >= b	6
<	Kleiner als	a < b	6
<=	Kleiner oder gleich	a <= b	6
==	Gleich	a == b	7
!=	Ungleich	a != b	7
&	Bitweises UND	a = b & c;	8
^	Bitweises EXOR	a = b ^ c;	9
	Bitweises ODER	a = b   c;	10
&&	Logisches UND	a = b && c;	11
	Logisches ODER	a = b    c;	12
?	Bedingte Zuweisung	a?b:c	13
=	Zuweisung	a = b;	14
,	Mehrfachausdruck	a <op>= b;	15
		a, b, c;	

Tabelle 6: Operationen und Rangfolge der Sprache C /C++.

### 5.3 Assoziativität

Die Assoziativität der Operatoren besagt ob bei Operatoren der gleichen Hierarchiestufe von links nach rechts oder von rechts nach links gruppiert wird.

Die Assoziativität der Operatoren kann in nachfolgender Tabelle gelesen werden. Man beachte das in C++ (und nur in C++) gewisse Operatoren überladen (d.h. für eine Klasse ( $\approx$ Datentyp)) umdefiniert werden können, so dass die Assoziativität gemäss Tabelle unsicher ist.

Kategorie	Operatoren	Assoziativität	Rang
primär	() [] ->	(l -> r)	1
unär	! ~ ++ -- (type) * & sizeof	r -> l (mit Ausn.)	2
unär	- +	r -> l	2
binär	* / %	l -> r	3
binär	+ -	l -> r	4
binär	>> <<	l -> r	5
binär	< <= > >=	l -> r	6
binär	!= ==	l -> r	7
binär	&	l -> r	8
binär	^	l -> r	9
binär		l -> r	10
binär	&&	l -> r	11
binär		l -> r	12
bedingt	? :	r -> l	13
Zuweisung	= += -= *= /= %= >>= <<= ^= &=  =	r -> l	14
Komma	,	l -> r	15

Tabelle 7: Assoziativität und Rangfolge der Operatoren in C.

Mit Klammern () kann, wie in der Mathematik, die Priorisierung der Auswertung beeinflusst werden.

### 5.3.1 Unäre Operatoren

Unäre Operatoren beziehen sich im Gegensatz zu binären Operatoren nur auf einen Operanden. Unäre Operatoren assoziieren rechts nach links. Die Operatoren ++,-- können von links nach rechts assoziieren.

Speziell sind die Inkrement- / Dekrementoperatoren ++ und -- in C. Diese Operatoren erhöhen / erniedrigen den Wert einer Variablen um eine Einheit. Diese Einheit ist Normalerweise die Zahl 1 (Ausnahmen werden später gezeigt).

```
++x;  
--y;
```

ist äquivalent zu

```
x = x + 1;  
y = y - 1;
```

Der -- / ++ Operator assoziiert beidseitig. Je nachdem wo der Operator steht (links oder rechts der Variablen) wird festgelegt, wann genau die Variable inkrementiert / dekrementiert wird. Dies spielt für unser obiges Beispiel eigentlich keine Rolle. Der Normalfall der Anwendung des ++ oder -- Operators ist jedoch in Ausdrücken wie

```
array[i++] = 0;
```

i++ sagt, dass zuerst der aktuelle Wert der Variablen i für die Indizierung benutzt wird und nachher wird i um eine Einheit inkrementiert. Während

```
array[++i] = 0;
```

++i besagt, dass zuerst der Wert der Variablen i um eine Einheit inkrementiert wird und nachher die Indizierung erfolgt.

Der C-Programmierer spricht je nachdem wann das Inkrement / Dekrement erfolgt von:

**Prefix-Notation**      Inkrement/ Dekrement erfolgt vor dem Zugriff auf die Variable (Bsp: ++i).  
**Postfix-Notation**     Inkrement / Dekrement erfolgt nach dem Zugriff auf die Variable (Bsp: i++).

Beispiele : Was ergibt konkret? (Vorgaben für jede Rechnung: a = 3; i = 0;)

q = --a + i++;      a =                  i =                  q =

p = a++ \* ++i;      a =                  i =                  p =

r = a+++i;          a =                  i =                  r =

## 5.4 Wertzuweisung

Die wesentliche Operation in Ausdrücken ist die Wertzuweisung. Sie erfolgt mit dem `=`-Zeichen:

```
int a;  
struct { int x;  
        int y;  
    } s, t;  
  
.....  
a=-10;  
s1.x=3;  
s2.y=4;  
t = s;
```

Es lassen sich alle Datentypen (ausser Arrays) direkt zuweisen. Es muss aber dafür Sorge getragen werden, dass nur zuweisungskompatible Datentypen einander zugewiesen werden. Notfalls ist eine entsprechende Typenkonversion durchzuführen.

Beispiel: Der ganzzahlige Wert (-23.78) einer Gleitkommazahl `f` soll der Ganzzahlvariablen `c` vom Typ `char` und `i` vom Typ `int` zugewiesen werden.

```
float f=-23.78f;  
char c;  
int i;  
.....  
c=(char) f;  
i=(int) f;
```

Der Nachkommaanteil wird durch diesen Cast abgeschnitten. Der Ganzzahlteil wird der entsprechenden Variablen als Ganzzahlwert zugewiesen.

C kennt auch mit Operatoren kombinierte Wertzuweisungen. Sie dienen hauptsächlich der vereinfachten Schreibweise. Sie bringen bezüglich der Laufzeiteffizienz meist nichts. Jedoch ist die Verwendung solcher Notationen in C mehr als üblich. So ist

```
x=x + 1;
```

äquivalent zu

```
x += 1;
```

Wir sehen es gibt in C viele Formulierungen um genau dasselbe zu formulieren.

## 5.5 Vergleiche

In C++ können alle Grössen auf Gleichheit und Ungleichheit geprüft werden. Numerische Datentypen (`char`, `extended`) können sogar mit Relationaloperatoren (`>`, `>=`, `<`, `<=`) geprüft werden. Eine detaillierte Betrachtung der Vergleiche erfolgt in Kapitel 6.4.2

Das Resultat eines Vergleiches ist immer 0 oder 1 vom Typ `int`:

- 1: Der Vergleich hat ergeben, dass die Vergleichsbedingung erfüllt wurde, also 'wahr' ist.
- 0: Die Vergleichsbedingung wurde nicht erfüllt. Die Vergleichsaussage ist also 'falsch'.

C kennt keine Datentypen für logische Werte, wie beispielsweise PASCAL mit dem `BOOLEAN`-Typ. In C werden logische Werte immer als `int` verkörpert mit den Werten:

- 0: Logisch FALSCH (`FALSE`)
- `≠0`: Logisch WAHR (`TRUE`)

Vergleichsergebnisse werden fast ausschliesslich als Entscheidungsgrundlage für Verzweigungen im Programm gebraucht.

## 6 Ablaufstrukturen

Ablaufstrukturen sind Anweisungen der Programmiersprache zur Steuerung des Programmlaufes. Da zu gehören Befehle der Gruppen:

- **Sequenz:** Nacheinander Ausführen von Anweisungen.
- **Alternative:** Bedingte Verzweigungen im Programmlauf aufgrund einer Bedingung.
- **Iteration:** Wiederholung von Befehlen oder ganzen Blöcken aufgrund eine Bedingung.

Zusätzlich werden wir in diesem Kapitel ein Hilfsmittel zum Darstellen von solchen Ablaufstrukturen kennen lernen: Die Strukturdiagramme. Wir zeigen die Verwendung von Strukturdiagrammen sinnvollerweise gerade direkt mit der Verwendung der entsprechenden C-Konstrukte.

### 6.1 Strukturdiagramme

Strukturdiagramme zeigen grafisch auf einer höheren Ebene den Programmablauf. Sie eignen sich hervorragend für den Programmwurf und Dokumentation. Wesentlich ist das Darstellen der Funktionalität im Programm. Deshalb sollten Strukturdiagramme nicht einzelne Instruktionen dokumentieren (die sieht man im Programm sowieso), sondern auf die Wirkungsweise auf einer höheren Stufe zeigen, ähnlich einem Blockschaltbild in der Elektrotechnik.

Die Praxis kennt unzählige verschiedene grafische Strukturdiagramme zum Darstellen des Programmablaufes. Viele von ihnen sind sehr speziell. Für den praktischen Nutzen kann man in einem ersten Ansatz drei Forderungen aufstellen:

1. Verbreitungsgrad: Ist diese Darstellung weit verbreitet und genormt (ANSI, DIN, ISO, etc.)?
2. Maschinentauglichkeit: Existieren Hilfsmittel (Programme) zum Erstellen solcher Diagramme?
3. Unabhängigkeit: Darstellung unabhängig vom zu lösenden Problem und Programmiersprache.

Wir zeigen hier die zwei wichtigsten Vertreter welche die drei oberen Anforderungen erfüllen:

- **Flussdiagramme** nach DIN
- Strukturdiagramme nach **Nassi-Shneidermann**

## 6.2 Das Flussdiagramm

Das Flussdiagramm zeigt mit einfachen Elementen den Programmablauf ähnlich einem elektrischen Schaltschema. Vom Konzept her stammt es aus der Lochkartenzeit-Ära und hat gewisse Mängel. So existiert beispielsweise beim Entwurf mit Flussdiagrammen keine Kontrolle bezüglich Realisierbarkeit.

Schulmässig werden jedoch oft andere, übersichtlichere Diagramme propagiert. Vor allem zum Entwurf und Beschreibung komplexer Abläufe wird das Flussdiagramm schnell unübersichtlich. Jedoch kann damit jede, noch so verworrene, Struktur beschrieben werden. Es ist daher nicht unbedingt das ideale Hilfsmittel zur strukturierten Programmierung.

Aus der Sicht der 'reinen Informatiklehre' sind Flussdiagramme nicht besonders gern gesehen. Der Informatiker sagt: Flussdiagramme unterstützen die 'strukturierte Programmierung'<sup>2</sup> nicht.

Der grosse Vorteil des Flussdiagrammes ist seine weltweite Verbreitung. Es ist DIN genormt und wird nicht nur zur Darstellung von Programmabläufen verwendet. Obwohl es vielerorts als Spagettidiagramm dargestellt wird, ist es international ohne grosse Erklärungen verstanden.

Für die manuelle Erstellung von Flussdiagrammen werden Schablonen angeboten. Besonders interessant sind aber die computerunterstützten Hilfsmittel zur Diagrammerstellung. Sie erlauben ein effizientes Erstellen und Verwalten von komplexen Diagrammen auf dem PC mit grafischer Unterstützung. Beispiele solcher Hilfsmittel sind: ABC, RFFlow, Micrografx Flowchart, etc.

### Anwendung:

Es wird von oben nach unten den Verbindungslinien entlang gelesen. Als Elemente zur grafischen Darstellung sind folgende Symbole empfohlen:

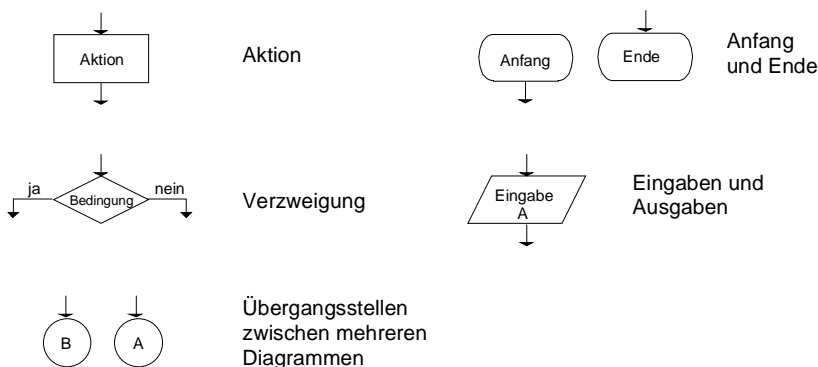


Bild 6-1: Grundelemente des Flussdiagrammes. (Auszug)

Nachstehend ein Musterbeispiel eines Flussdiagrammes. Es beschreibt ein Programm zur Erzeugung von Primzahlen:

<sup>2</sup> Strukturierte Programmierung:

Festlegung bei der Programmierung auf einen Satz von Regeln (Einschränkungen) die Übersichtlichkeit und Sicherheit erhöhen. Im wesentlichen umfasst dies:

1. Nur die Verwendung der Ablaufstrukturen Sequenz, Alternative, Iteration.
2. Begrenzte Datenverfügbarkeit. Daten soweit möglich lokal halten. Keine globalen Daten.
3. Hierarchischer Programmaufbau mit schrittweiser Verfeinerung.

Wesentlich ist Forderung, dass keine unbedingten Sprungbefehle verwendet werden (goto).

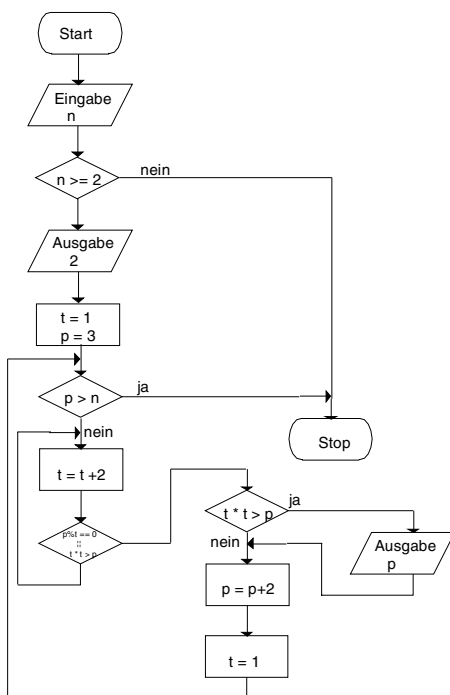


Bild 6-2: Beispiel eines einfachen Flussdiagrammes.  
(Primzahlen bestimmen)

### 6.3 Nassi-Shneiderman

Vielfach wird unter dem Begriff *Struktogramm* ein Diagramm nach *Nassi-Shneiderman* verstanden. Sein Vorteil: Durch klare Regelungen in der Symbolik sind undisziplinierte Sprünge fast unmöglich.

Nassi-Shneiderman-Diagramme sind in den 70er Jahren als dasjenige Werkzeug zur methodischen (strukturierten) Entwicklung von Mikroprozessorsoftware vorgestellt worden. Dieser Zeit haftet das Manko an, dass sehr viel Material entwickelt wurde, das schlecht dokumentiert war. Selbst kleinste Änderungen waren ein Ding der Unmöglichkeit, weil weitgehend alles in Assembler codiert wurde. So suchte man Lösungen, um die Wirkungsweise auf einem höheren Niveau darzustellen. So erlebte dieses Diagramm eine gewisse Blütezeit, vor allem in Bereichen der Mikroprozessorsoftware. In anderen Bereichen war die Akzeptanz eher gering.

Der grosse Vorteil von Nassi-Shneiderman gegenüber von Flussdiagrammen ist, dass eine strukturierte Programmierung erzwungen wird. Ein Problem dieser Methode darf jedoch nicht verschwiegen werden: Alles was interessant ist, liegt unten rechts, was zwangsläufig zu Platzproblemen führt.

Klarer Nachteil ist jedoch die weitgehende Maschinenuntauglichkeit dieser Methode. Die Diagramme müssen meist in mühsamer Zeichenarbeit erstellt werden, was der Effizienz der Arbeit nicht gerade förderlich ist. Mit den heutigen Zeichenprogrammen ist dieser Nachteil wieder kleiner geworden. Desgleichen existieren verschiedene Programme zur Arbeit mit Nassi-Shneiderman-Diagrammen ( Easy-Case von Siemens, u.a.).

Die Diagramme haben folgende Grundelemente:

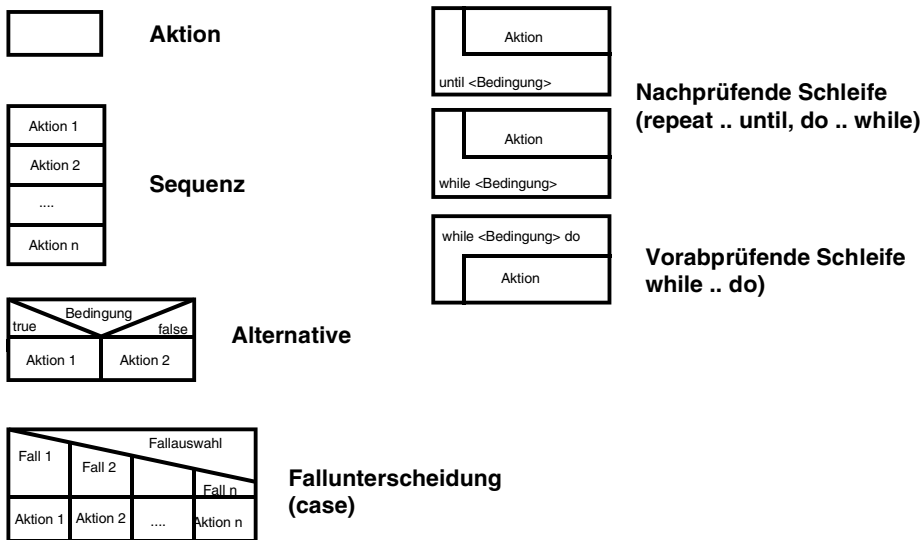


Bild 6-3: Elemente des Nassi-Shneiderman-Diagrammes.

Die Grundelemente lassen sich in Gruppen mit den drei **Grundablaufstrukturen** einteilen. Jede höhere Programmiersprache bietet heute eigentlich in der Sprachdefinition exakt die Elemente an, so dass Diagramme nach Nassi-Shneiderman sehr bequem zu handhaben sind. Sie eignen daher gut für **Nachdokumentationen**.

Struktogramme nach Nassi-Shneiderman erlauben eine grössere Darstellungsfreiheit als dies mit den konventionellen 3 Grundstrukturen möglich ist. Diese Erweiterungen sind aber immer wieder auf diese drei Elementarstrukturen Sequenz, Alternative und Iteration zurückzuführen.

## 6.4 Anwendung

Es wird von oben nach unten gelesen und von links nach rechts. Jede einzelne Aktion wird in einen Strukturblock eingetragen.

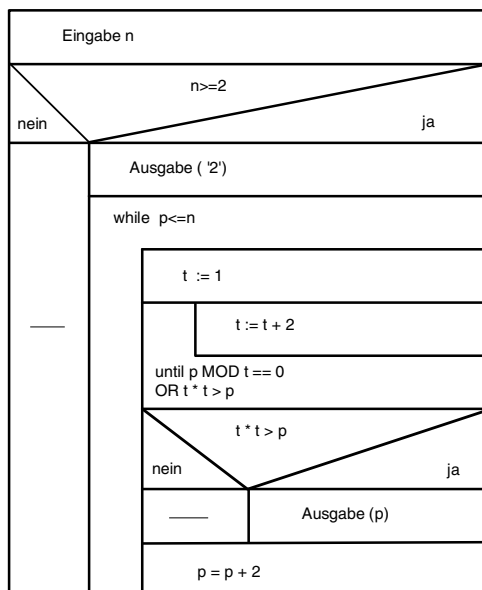


Bild 6-4: Gleiches Programm wie Bild 6-2, aber mit Nassi-Shneiderman dargestellt.

### 6.4.1 Sequenz

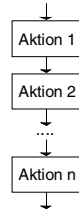
Von den vorgehenden Beispielen ist bereits bekannt, dass aufeinander folgende Anweisungen mit Semikolon getrennt werden.

C-Code:

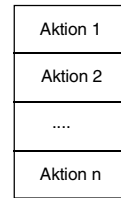
```
Aktion1;
Aktion2;
...
Aktionn;
```

Bsp.: `j = k/4;`  
`i = 2 + wert;`

Flussdiagramm



Nassi-Shneiderman



**Sequenz**

Das Semikolon bewirkt, dass die Anweisung abgeschlossen wird und damit im Rechner ausgeführt wird. In anderen Sprachen, wie beispielsweise PASCAL hat das Semikolon syntaktisch eine andere Bedeutung (Anm.: Es separiert dort Blöcke) obwohl die Verwendung sehr ähnlich erscheint.

Ein Spezialfall von C ist der so genannte **Kommaoperator**. Er dient dazu mehrere Anweisungen syntaktisch zu einer einzelnen Anweisung zusammenzufassen, ähnlich einem Block {}:

```
for (i=0; i <10; i++, j++, k++) ...;
```

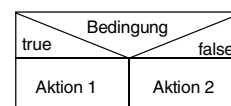
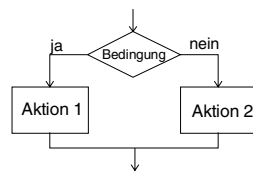
Er bewirkt hier, dass am Schluss eines Schleifendurchlaufes eine ganze Sequenz ausgeführt wird, nämlich `i++, j++, k++`. Grundsätzlich kann ohne grossen Aufwand eine Komma-Lösung immer kommalos formuliert werden, indem man die auszuführende Sequenz geeignet in einen Block packt oder so. Er wird deshalb eher selten verwendet.

### 6.4.2 Alternative (Vergleiche) `if (...) .. else ..`

**Alternativen** verzweigen den Programmfluss aufgrund einer logischen Bedingung. Diese Verzweigungsbedingung wird meist durch einen Vergleich gewonnen. Wenn die Vergleichsbedingung erfüllt ist, wird die nachfolgende Anweisung ausgeführt. Ist sie nicht erfüllt, wird der `else`-Zweig ausgeführt. Bei solchen 'if .. then..else'-Strukturen spricht man auch von **echten Alternativen**.

Syntax:

```
if (Bedingung)
    Statement1;
else
    Statement2;
```

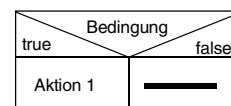
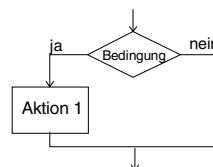


**Echte Alternative**

C kennt auch die unechte Alternative. Sie besitzt keinen `else`-Zweig:

Syntax:

```
if (Bedingung)
    Statement1;
```



**Unechte Alternative**

Der Entscheid welcher Zweig ausgeführt wird, erfolgt immer durch Auswerten der Bedingung. Diese Bedingung muss immer in Klammern stehen und wird als Wert vom Typ `int` interpretiert. Ist der Wert ungleich 0 so ist die Bedingung erfüllt und es wird die Anweisung direkt nach der Bedingung ausgeführt. Ist der Wert der Bedingung gleich 0 so wird, falls vorhanden, der `else` Zweig ausgeführt.

Ein Beispiel mit einer gemischten Verwendung von echten und unechten Alternativen zeigt das nachfolgende Programm:

```
/* Arbeiten mit Vergleichsbefehlen und boolschen Groessen. File: Aternative.C
   Obwohl C standardmaessig keine boolschen Datentypen kennt, werden oft
   TRUE und FALSE sowie der Typ BOOL definiert. Neuere Compiler haben
   vielfach bool als boolschen Typ synonym zu int vordefiniert.

   Autor: Gerhard Krucker
   Sprache: Intel C++ V7.1
   Datum: 17.10.2003
*/
#include <stdio.h>

#define TRUE 1
#define FALSE 0
typedef int BOOL;      /* Wahrheitswerte sind in C immer int's */

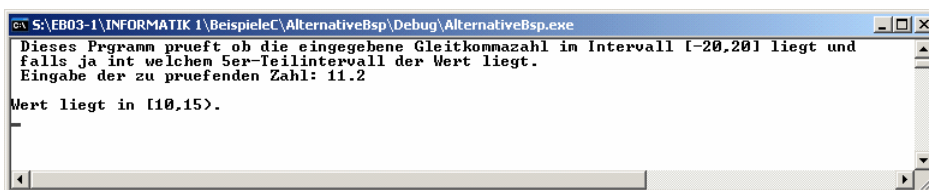
main()
{ double wert;        /* Zu testende Zahl */
  BOOL  imIntervall; /* TRUE=Zahl war im Intervall [-20,20] sonst FALSE */

  printf(" Dieses Prgramm prueft ob die eingegebene Gleitkommazahl im Intervall [-20,20] "
        " liegt und\n"
        " falls ja int welchem 5er-Teilintervall der Wert liegt.\n"
        " Eingabe der zu pruefenden Zahl: ");
  scanf("%lf",&wert);

  if ((wert < -20) || (wert > 20))
    imIntervall = FALSE;
  else
    imIntervall = TRUE;

  if (imIntervall)
  { if (wert < -15) printf("\nWert liegt in [-20,-15).\n");
    if ((wert >= -15) && (wert < -10)) printf("\nWert liegt in [-15,-10).\n");
    if ((wert >= -10) && (wert < -5)) printf("\nWert liegt in [-10,-5).\n");
    if ((wert >= -5) && (wert < 0)) printf("\nWert liegt in [-5,-0).\n");
    if ((wert >= 0) && (wert < 5)) printf("\nWert liegt in [0,5).\n");
    if ((wert >= 5) && (wert < 10)) printf("\nWert liegt in [5,10).\n");
    if ((wert >= 10) && (wert < 15)) printf("\nWert liegt in [10,15).\n");
    if (wert >= 15) printf("\nWert liegt in [15,20).\n");
  }
  else printf("Wert liegt nicht im Intervall [-20,20]\n");

  return 0;
}
```

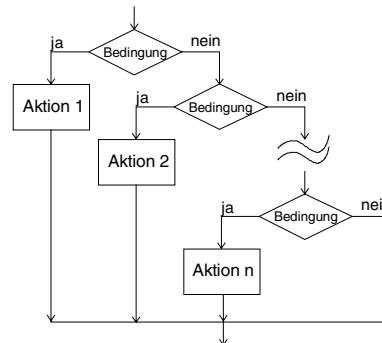


### 6.4.3 Fallunterscheidung (Mehrfachauswahl) `switch (..)`

C/C++ kennt, wie viele andere Programmiersprachen auch, die mehrfache Alternative. Hierbei können aufgrund eines ganzzahligen *Selektors* verschiedene Alternativzweige gewählt werden.

Syntax:

```
switch (Ausdruck)
{ case Konst1:
  Aktion1; break;
  case Konst2:
  Aktion 2; break;
  ...
  case Konstn:
  Aktion n;
}
```



### Fallunterscheidung (Case)

	Fallauswahl		
Fall 1	Fall 2	...	Fall n
Aktion 1	Aktion 2	...	Aktion n

Mit `switch (Ausdruck)` wird aufgrund des Wertes von *Ausdruck* einer der folgenden Fälle angewählt die mit den `case`-Selektoren spezifiziert werden.

Für die Wahl der Selektoren sowie für den Wert von *Ausdruck* gelten gewisse Einschränkungen:

*Ausdruck* (muss übrigens immer in Klammern stehen) muss ein Wert vom Typ `char` sein. Neuere Compiler erlauben auch `int`. (MS-Visual C++ und Borland C++ Builder erlauben `int`.)

Die `case`-Selektoren (*Konst1..Konstn*) müssen Konstanten sein und im Wertebereich von `char` (resp. `int`) liegen. Für jeden Fall, der auftreten kann, muss ein Selektor vorhanden sein, sonst ist das Verhalten undefiniert. C bietet jedoch für alle Fälle, die nicht durch `Cases` abgedeckt werden, den Meta-Selektor `default` an. Er wird gewählt wenn kein `case` passt.

Sollen mehre Selektoren zum gleichen Ereignis triggern, so sind alle Selektoren nacheinander aufzuführen. Eine Bereichsformulierung (ähnlich PASCAL) ist nicht möglich.

#### Beispiel:

```
switch (eingabe)
{ case 'a':
  case 'b':
  case 'c':
    printf("Eingabe war a,b oder c!\n");
    break;
  case 'd': case 'z':
    printf("Eingabe war d oder z!\n");
    break;
  default:
    printf("Eingabe war sonstwas!\n");
}
```

Wir haben in allen `switch()` am Ende einer Fallbearbeitung die Anweisung `break` gesehen. `break` bewirkt ein Beenden der aktuellen Struktur. Hier heisst das, dass `switch()` sofort beendet wird und mit der nächsten Anweisung nach dem `switch()` weitergefahren wird. Würde man das `break` weglassen, so würden **alle nachfolgenden Anweisungen** nach dem ersten passenden Selektor ausgeführt (inklusive `default`).

`break` ist also ein sog. Sprungbefehl. Er wird verwendet um eine `switch()`-Anweisung zu verlassen oder um Schleifen zu beenden.

### 6.4.4 Bedingungsoperator

Er ist ein Grenzfall zwischen Operator und Steueranweisung. Er dient hauptsächlich dazu um zwei verschiedene Werte aufgrund einer Bedingung zuzuweisen. Der gesamte Ausdruck hat die Form  $?$  : und liefert, je nach Bedingung, entweder den Wert von Ausdruck1 oder Ausdruck2.

$(\text{Bedingung}) ? \text{Ausdruck1} : \text{Ausdruck2};$

Bei der Auswertung wird zuerst die Bedingung ausgewertet. Sie wird auf ihren logischen Wert (0 oder  $\neq 0$ ) geprüft. Ist sie logisch wahr (d.h.  $\neq 0$ ) wird *Ausdruck1* genommen, ist die Bedingung logisch falsch (d.h. 0) wird *Ausdruck2* genommen.

#### Beispiel:

$(i < 0) ? 0 : 1;$

Ist  $i$  kleiner als 0 so wird der Wert des ganzen Ausdrucks 0, ist hingegen  $i$  grösser oder gleich 0 so ist der Wert des Ausdrucks 1.

Wie bereits erwähnt, werden solche Konstrukte meist für bedingte Zuweisungen genutzt. So könnte beispielsweise die Vorzeichenfunktion

$$\text{sgn}(x) := \begin{cases} -1 & x < 0 \\ 1 & x \geq 0 \end{cases}$$

so realisiert werden:

$\text{sgn} = (x < 0) ? -1 : 1;$

Bedingte Zuweisungen sind immer vereinfachte `if .. else`-Konstrukte. Sie stammen aus den Anfängen der C-Sprachdefinition und wurden bis nach C++ 'weitergepflegt'. Grundsätzlich können alle Ausdrücke mit dem Bedingungsoperator in eine `if .. else`-Form gebracht werden, so dass die Formulierung auch in andere Sprachen leicht zu übertragen ist:

```
if (x < 0)
    sgn = -1;
else
    sgn = 1;
```

### 6.4.5 Schleifenbefehle

Schleifenbefehle bewirken, dass Anweisungen oder ganze Gruppen aufgrund einer Bedingung eine bestimmte Anzahl Male wiederholt werden. Diese Wiederholungen von Befehlen werden im Jargon 'Schleifen' genannt.

Die logische Bedingung, aufgrund derer eine Schleife durchläuft, wird Laufkriterium genannt. Die Schleife läuft also solange das Laufkriterium erfüllt ist (also logisch wahr ist). Ist das Laufkriterium nicht mehr erfüllt, wird die Schleife beendet und es wird mit der nächsten Anweisung nach der Schleife weitergefahren.

C kennt drei Sorten von Schleifen, die jeweils unterschiedlich ihr Laufkriterium prüfen.

<code>while() Anweisung;</code>	Vorabprüfende Schleifen. Das Laufkriterium wird jeweils vor dem Ausführen von <i>Anweisung</i> geprüft.
<code>for (Start; Laufkriterium; Ausdruck) Anweisung;</code>	
<code>do Anweisung while(Bedingung);</code>	Nachprüfende Schleife. Zuerst wird <i>Anweisung</i> ausgeführt und nachher wird das Laufkriterium ( <i>Bedingung</i> ) geprüft.

Schleifen bewirken Wiederholungen von Anweisungen und werden deshalb auch oft **Iterationen** genannt. Die Iteration, verkörpert durch die `while()`-Anweisung, gehört zu den drei Grundablaufstrukturen. Mit diesen 3 Grundablaufstrukturen (Sequenz, Alternative, Iteration) lassen sich alle Programmkonstrukte im Sinne der strukturierten Programmierung realisieren. Daraus folgt, dass mit der `while()`-Schleife auch alle anderen Schleifen realisiert werden können.

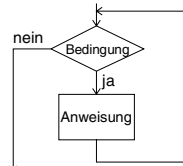
### 6.4.6 Vorabprüfende Schleife: while

Sie verkörpert den klassischen Typ der vorab prüfenden Schleife. Die allgemeine Form und die Strukturdiagramme präsentieren sich die `while()`-Schleife:

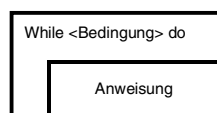
**C-Code**

```
while (Bedingung)
    Anweisung;
```

**Flussdiagramm**



**Nassi-Shneiderman**



**Vorabprüfende Schleife (While .. do)**

Die in der Schleife enthaltene Anweisung wird solange wiederholt wie die Bedingung logisch wahr ist. Die Bedingung muss, wie bei den Vergleichen, zwingend immer in Klammern stehen. Die Anweisung im Schleifenkörper kann dabei eine einzelne Anweisung oder ein ganzer Block sein.

Die Bedingung verkörpert das Laufkriterium und ist immer ein logischer Wert vom Typ `int`. Sie ist erfüllt, wenn der Wert wahr ist, d.h.  $\neq 0$  ist. Bei diesem Schleifentyp wird immer zuerst das Laufkriterium geprüft und falls erfüllt, wird die enthaltene Anweisung ausgeführt.

**Beispiel:** Wir wollen aufeinander folgende Ganzzahlen von 0 bis 100 auf dem Bildschirm ausgeben.

```
#include <stdio.h>

main()
{ int z;          /* Aktuelle Ganzzahl */

  z=0;           /* Beginne mit der ersten Zahl 0 */
  while (z <= 100) /* Wiederhole solange z <= 100 */
  { printf("%d\n",z);
    z++;
  }

  return 0;
}
```

**Weiteres Beispiel:** Umwandlung von Kleinbuchstaben in Grossbuchstaben.

In diesem etwas umfangreicheren Beispiel wird zuerst zeichenweise mit einer `while()`-Anweisung eine Textzeile mit der `getchar()`-Funktion eingelesen und in einem Array abgespeichert. Nachher wird mit einer weiteren `while()`-Schleife jedes Zeichen auf Grossschreibung geprüft und falls nein, wird der entsprechende Buchstabe mit der `toupper()`-Funktion in Grossschrift gewandelt (Anm.: Geht nicht für Umlaute). In einer letzten Schleife wird die gewandelte Textzeile zeichenweise mit der `putchar()`-Funktion ausgegeben.

```
/* Benutzung von While-Schleifen:                               File: WHILE2.C
   Textzeile zeichenweise einlesen und in Grossbuchstaben wandeln
   mit anschliessender Ausgabe auf den Bildschirm.

   Autor: Gerhard Krucker
   Datum: 16.10.2003
   Sprache: Intel C++ V7.1
*/
#include <stdio.h>
#include <ctype.h>

#define EOL '\n'

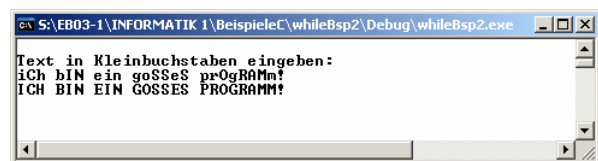
main()
{ char zeile[80]; /* Array fuer Textzeile mit max. 80 Zeichen */
  int anzahl;    /* Anzahl Zeichen im Array */
  int zaehler;   /* Laufvariable fuer Schleife */

  /* Textzeile zeichenweise einlesen und in das Array abspeichern. */
  printf("\nText in Kleinbuchstaben eingeben:\n");
  anzahl = 0;
  while((zeile[anzahl]=getchar()) != EOL) ++anzahl; /* Zeichen einlesen, ins Array speichern,
                                                    solange nicht End Of Line gelesen wird */

  /*Text in Grossbuchstaben wandeln */
  zaehler = 0;
  while (zaehler < anzahl)
  { if (islower(zeile[zaehler])) /* Wenn das Zeichen ein Kleinbuchstabe ist dann wandeln
*/
    zeile[zaehler] = toupper(zeile[zaehler]);
    zaehler++;
  }

  /* Text zeichenweise ausgeben */
  zaehler = 0;
  while (zaehler < anzahl)
    putchar(zeile[zaehler++]);

  return 0;
}
```

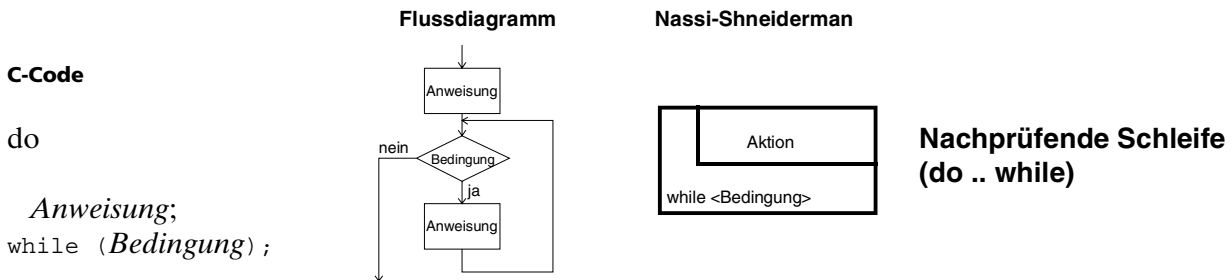


### 6.4.7 Nachprüfende Schleife: do..while

Bei dieser Schleife wird das Laufkriterium erst am Schluss des Schleifendurchlaufes geprüft. Das heisst, die jeweils in der Schleife enthaltene Anweisung zuerst ausgeführt und zwar unabhängig davon, ob das Laufkriterium erfüllt ist oder nicht. Erst nachher wird entschieden, ob der Durchlauf wiederholt werden soll.

Dieser Schleifentyp eignet sich besonders da, wo im Schleifenkörper das Laufkriterium selbst erzeugt wird, also von der Art 'ich wiederhole solange bis eine Abbruchbedingung erreicht wird.'

In C-Code und den Strukturdiagrammen sehen die Schleifen so aus:



Zu beachten sind hier die unterschiedlichen Formulierungen der Laufbedingung: In Nassi-Shneiderman hat die Schleife gemäss Konvention ein **Abbruchkriterium**. Die Schleife läuft also solange bis eine Abbruchbedingung erfüllt ist. In C haben wir hingegen eine **Laufbedingung**, hier läuft die Schleife solange wie die Laufbedingung erfüllt ist, also eine genau inverse Bedingung.

**Beispiel:** Ausgeben der ganzen Zahlen von 1 bis 10 auf den Bildschirm mit do..while-Schleife.

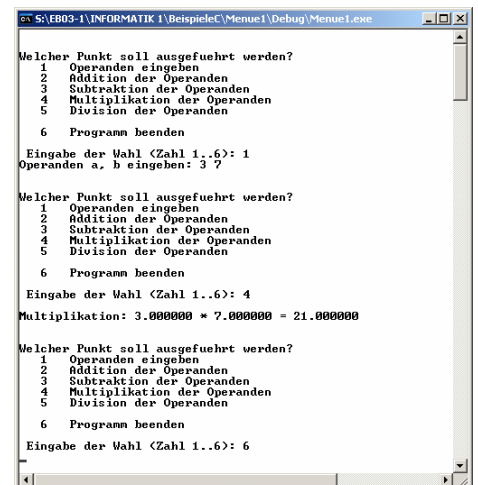
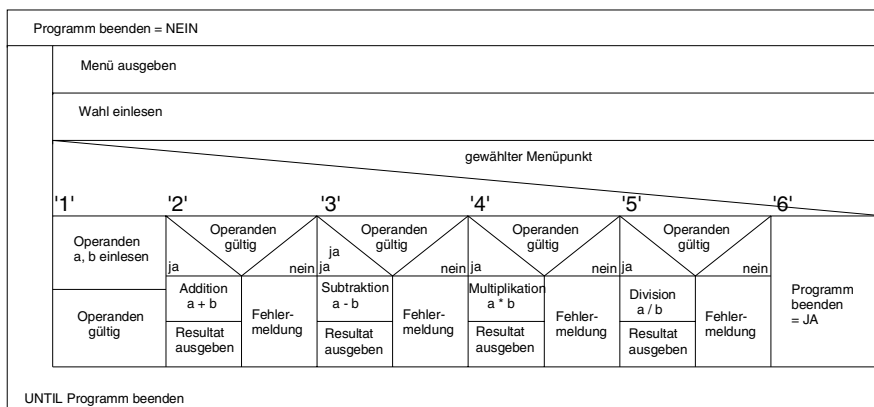
```
#include <stdio.h>
main()
{ int zahl =1;

  do
    printf("%d\n", zahl++);
  while (zahl <= 10);

  return 0;
}
```

**Beispiel:**

Ein etwas umfangreicheres Beispiel ist die nachfolgende 'menügesteuerte' Berechnung. Hierbei können über Menüpunkte verschiedene Aktivitäten gewählt werden:



```

/* Einfaches Programm mit Menuesteuerung
   Ueber Menuepunkte kann gewaehlt werden ob:
   - Operanden eingegeben werden sollen
   - Operation +,-,/,* ausfuehren
   - Programm beenden

   Autor: Gerhard Krucker
   Datum: 16.10.2003
   Sprache: Intel C++ V7.1
*/
#include <stdio.h>
#include <ctype.h> /* fuer isalnum() */

main()
{
    int wahl; /* Gewaehlter Menuepunkt */
    int op_da; /* 0= Keine Operanden in a, b; 1=Operanden wurden eingegeben */
    int fertig; /* 0=Programm beenden, 1 = Weiterfuehren */
    double a, b; /* Operanden a und b als Gleitkommazahl */
    double c; /* Resultat der Berechnung */

    op_da = 0; /* Noch keine Operanden eingegeben */
    fertig = 0; /* Programm noch nicht beenden */

    do {
        printf("\n\nWelcher Punkt soll ausgefuehrt werden?\n"
            " 1 Operanden eingeben\n"
            " 2 Addition der Operanden\n"
            " 3 Subtraktion der Operanden\n"
            " 4 Multiplikation der Operanden\n"
            " 5 Division der Operanden\n"
            " 6 Programm beenden\n\n"
            " Eingabe der Wahl (Zahl 1..6): ");
        do wahl=getchar(); while (!isalnum(wahl)); /* Zeichen aus dem Puffer lesen dabei
                                                    whitespace ueberlesen */
        if ((wahl <'1')|| (wahl >'6')) /* Wurde etwas unerlaubtes gedruickt? */
            printf("\nFehler: Nur Menuepunkte 1..6 erlaubt!\n");
        else
            {
                switch (wahl) {
                    case '1':
                        printf("Operanden a, b eingeben: ");
                        scanf("%lf %lf",&a,&b);
                        op_da=1; /* Operanden wurden eingegeben */
                        break;
                    case '2': /* Addition durchfuehren */
                        if (op_da)
                            {
                                c = a+b;
                                printf("\nAddition: %f + %f = %f\n",a,b,c);
                            }
                        else printf("\nKeine Operanden eingegeben!");
                        break;
                    case '3': /* Subtraktion durchfuehren */
                        if (op_da)
                            {
                                c = a-b;
                                printf("\nSubtraktion: %f - %f = %f\n",a,b,c);
                            }
                        else printf("\nKeine Operanden eingegeben!");
                        break;
                    case '4': /* Multiplikation durchfuehren */
                        if (op_da)
                            {
                                c = a*b;
                                printf("\nMultiplikation: %f * %f = %f\n",a,b,c);
                            }
                        else printf("\nKeine Operanden eingegeben!");
                        break;
                    case '5': /* Division durchfuehren */
                        if (op_da)
                            {
                                c = a/b;
                                printf("\nDivision: %f / %f = %f\n",a,b,c);
                            }
                        else printf("\nKeine Operanden eingegeben!");
                        break;
                    case '6':
                        fertig = 1;
                } /* switch (wahl) */
            } /* else */
        } while (!fertig);
    }
    return 0;
}

```

### 6.4.8 Kombischleife for(...)

Die `for( . . )`-Schleife ist eine Kombischleife mit Vorabprüfung des Laufkriteriums. Sie ist aufgrund ihrer einfachen Formulierung die wahrscheinlich meistverwendete Schleife in C. Ein Beispiel:

- ①      ②      ③

```
for(i = 0; i < 10; i++) printf("%d");
```

Die Bedingung in Klammern für die `for`-Schleife besteht aus drei Teilen.

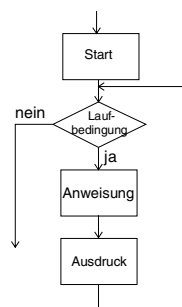
1. Einem Ausdruck für den Anfangswert. Er wird vor dem ersten Durchlauf ausgewertet. Hier wird also die Variable `i` auf Null gesetzt.
2. Dem Laufkriterium, hier `i < 0`. Die Schleife läuft solange durch, wie das Laufkriterium wahr ist.
3. Aktion am Ende jedes Schleifendurchlaufes, hier `i++`. Am Ende jedes Durchlaufes wird die Variable `i` inkrementiert.

Für diesen Schleifentyp existieren keine direkten Diagrammelemente im FD und Nassi-Shneiderman. Sie müssen aus den Grundelementen zusammengesetzt werden:

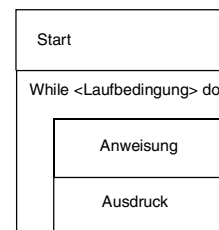
#### C-Code

```
for (Start;Bedingung; Ausdruck)  
Anweisung;
```

#### Flussdiagramm



#### Nassi-Shneiderman



**for - Schleife**

Start ist der Ausdruck, der vor dem ersten Schleifendurchlauf ausgewertet wird. Normalerweise wird hier eine Laufvariable initialisiert. Bedingung ist das logische Laufkriterium. Die Schleife wird also solange ausgeführt wie hier ein Wert `!= 0` steht. Ausdruck ist Aktion, die jeweils am Ende jedes Durchlaufes ausgeführt wird. Normalerweise wird hier die Laufvariable inkrementiert.

Zwei Dinge sind speziell zu beachten:

1. Für *Start*, *Bedingung* und *Ausdruck* sind nur einzelne Anweisungen erlaubt, also keine Blöcke. Müssen ausnahmsweise mehrere Anweisungen formuliert werden, muss dies mit dem Kommaoperator erfolgen.
2. Einzelne Felder in der Klammer dürfen leer sein:  

```
for (i=0; i < 10 ; ) {Anweisung; i++;}
```

  
ist dasselbe wie  

```
for (i = 0; i < 10; i++ ) Anweisung;
```

  
Ein Spezialfall ist:  

```
for (; ) Anweisung;
```

  
Diese Schleife terminiert nie. Dass heisst Anweisung wird immer wiederholt.

### Beispiel:

Mittelwert von n Zahlen berechnen:

```
/* Mittelwert von n Zahlen berechnen                               File: ForBsp1.C
   Ganzzahlen von der Tastatur einlesen, aufsummieren
   und anschliessend Mittelwert berechnen und ausgeben.

   Autor: Gerhard Krucker
   Datum: 16.10.2003
   Sprache: Intel C++ V7.1
*/

#include <stdio.h>

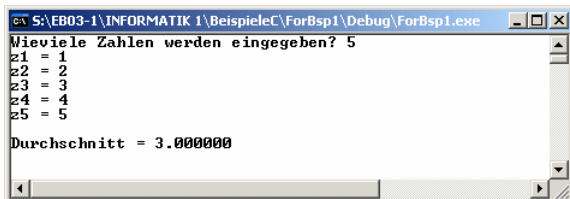
main()
{ int n, zaehler;
  float x, durchschnitt, summe;

  /* Initialisieren und die Anzahl einzugebender Zahlen n einlesen */
  summe=0;
  printf("Wieviele Zahlen werden eingegeben? ");
  scanf("%d",&n);

  /* Zahlen einlesen */
  for (zaehler=0; zaehler < n; zaehler++)
    { printf("z%d = ",zaehler+1);
      scanf("%f",&x);
      summe += x;
    }

  /* Mittelwert berechnen und das Resultat ausgeben */
  durchschnitt = summe / n;
  printf("\nDurchschnitt = %f\n",durchschnitt);

  return 0;
}
```



```
ex S:\EB03-1\INFORMATIK 1\BeispieleC\FörBsp1\Debug\FörBsp1.exe
Wieviele Zahlen werden eingegeben? 5
z1 = 1
z2 = 2
z3 = 3
z4 = 4
z5 = 5
Durchschnitt = 3.000000
```

## 7 Zeiger und Speicherplatzadressen

Zeiger dienen allgemein zur Verwaltung von Speicherplatz. Über Zeiger können Variablen oder sogar Funktionen angesprochen werden. Am besten ist das Zeigerprinzip mit der indirekten Speicheradressierung in Assembler zu beschreiben.

Dieses Kapitel ist vielleicht neben den Grunddatentypen das wichtigste Kapitel. Obwohl vom Prinzip her einfach, wird anfangs das Arbeiten mit Zeiger als recht belastend empfunden. Dies äussert sich dann auch in den praktischen Übungen, die zu Beginn oft ein wenig frustrierende Resultate liefern.

### 7.1 Motivation

Trotzdem ist die Arbeit mit Zeigern und Adressen etwas enorm praktisches und effizientes.

Am besten vergleicht man es so: Sicher ist Autofahren zu lernen mit mehr Aufwand verbunden als Velofahren. Beherrscht man es aber, so kann man in viel kürzerer Zeit grössere Strecken überwinden.

In C braucht man Zeiger. Deshalb ist es besser man befasst sich von Anfang an gründlich damit.

In C++ wurde mit der Einführung von Referenzen das fehlerträchtige Zeigerkonzept abgelöst. Man hat daher in C++ die Möglichkeit zeigerfreie Programme zu realisieren.

Zuerst einige Definitionen. Am besten lernt man sie auswendig und verwendet sie genauso. Dies ist anfangs sicher die beste Methode, bis man das Zeigerkonzept in seiner ganzen Tiefe erfasst hat.

**Zeiger:** **Zeiger sind Variablen die eine (Speicher-) Adresse beinhalten.**  
Sie zeigen also auf Speicherstellen, d.h. Variablen oder Funktionen. Zeiger sind typisiert, d. h. ein Zeiger 'weiss' auf welchen Datentyp er zeigt.

**Zeigervariablen:** **Variable, die einen Zeiger enthält.**

**&** **Adressoperator**  
Er liefert die Adresse des entsprechenden Elementes. Er wird meist zum Bilden eines Zeigers auf eine Variable benutzt.

**\*** **Dereferenzierungsoperator**  
Wird \* auf einen Zeiger angewandt, so wird der Inhalt der durch den Zeiger adressierten Variablen geliefert.

In C werden Zeiger zur Verwaltung von Datenstrukturen verwendet. So ist es beispielsweise viel einfacher ein Array mit 1000 Elementen über einen Zeiger zu verwalten als jedesmal die Gesamtheit aller 1000 Elemente anzusprechen.

Weiter werden Zeiger zur Rückgabe von Funktionsresultaten über Parameter benutzt. Mehr hierzu im Kapitel 9-

## 7.2 Grundsätzliches über Zeiger

Zeiger werden hauptsächlich verwendet um Speicherplatz zu verwalten. So kann jede Variable über ihre Speicherplatzadresse angesprochen werden. Sie ist vor allem beim Verschieben von sehr grossen Datenobjekten ein Vorteil, weil nur die Zeiger verschoben werden können und nicht das ganze Datenobjekt selbst.

Jeder Variable ist durch vier Dinge beschrieben: Speicherplatzadresse, Name, Datentyp und Inhalt

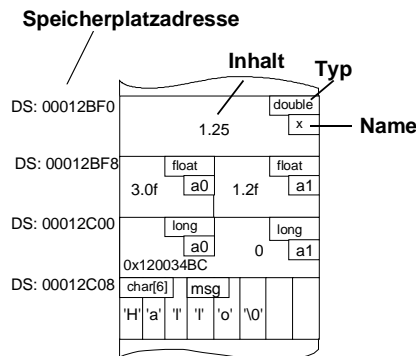
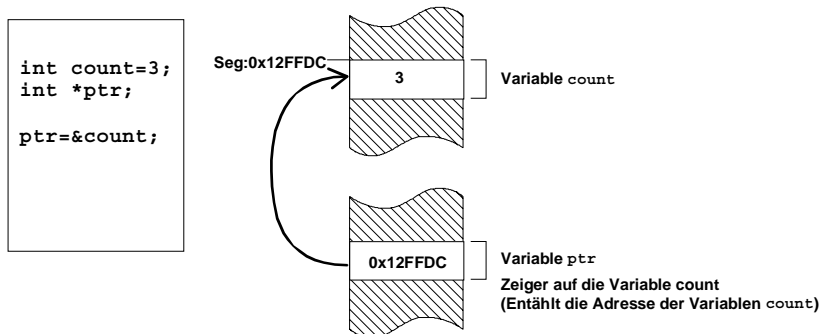


Bild 7-1: Beschreibung von Variablen mit Speicherplatz, Datentyp, Name und Inhalt.

Zeigervariablen beinhalten Zeiger, also Speicherplatzadressen. Sie werden definiert, indem der Variablendefinition ein Stern vor den Namen gesetzt wird:d:

Bsp: `int *a; /* Die Variable a ist ein Zeiger auf eine int */`

Bildlich können wir uns die Zeiger so vorstellen:



Seg:  
DS bei globalen und statischen Variablen  
SS bei lokalen Variablen

Bild 7-2: Zeiger und Speicherplatzadressen.

Zeiger haben eine starke Typenbindung zum Typ, auf den sie zeigen. So kann beispielsweise ein Zeiger auf eine `int` nur Adressen von `int`-Variablen aufnehmen. Der Versuch Adressen von anderen Typen zuzuweisen wird bereits beim Kompilieren mit einer Fehlermeldung quittiert.

Aufgrund dieser Typenbindung 'weiss' der Zeiger beim Erhöhen des Zeigerwertes um wieviele Bytes er inkrementiert werden soll.

Zeiger können für alle Datentypen definiert werden. Selbst auf Zeiger können Zeiger definiert werden.

Beispiel:

```
int a,*p; /* Variable a vom Typ int und p ist ein Zeiger auf eine int */
float *pf, *x;
FILE *fp;
char *pc;
int **a; /* a ist ein Zeiger auf einen Zeiger auf eine int */
```

Mit dem Dereferenzierungsoperator `*` kann auf den Inhalt der Speicherstelle auf den der Zeiger

zeigt, zugegriffen werden.

Beispiel:

```
double *a;
...
*a = 3.14159;      /* 3.14159 einschreiben wo a hinzeigt */
```

Mit dem **Adressoperator** & die Adresse einer beliebigen Variablen bestimmt werden. Der Adressoperator liefert immer einen Zeigerwert (also eine Speicheradresse). Diese können zur Zuweisung an Zeigervariablen benutzt werden.

Beispiel:

```
int *a;
int i;
...
a = &i;           /* a erhält die Adresse der Variablen i */
```

C kennt für Zeiger den Standardwert (Standardkonstante) NULL. Er ist vordefiniert und wird für spezielle Anwendungen verwendet: NULL zeigt an, dass hier keine gültige Adresse vorliegt.

NULL verkörpert in den meisten Systemen die Speicheradresse 0. Sie ist aber über Dereferenzierung in den meisten Systemen beschreibbar, was aber nie geschehen darf. Bei Windows 3.x beispielsweise war dies fatal, weil die ersten 16 Bytes des Datensegmentes intern zur Speicherverwaltung benutzt werden. Windows 95, Windows NT/2000 reservieren die ersten 4K resp. 64K des Segmentes. Ein Zugriff auf diesen Bereich wird mit einer Schutzverletzung quittiert.

Nachfolgend ein Programm, das den Zeigerwert, also die Speicheradressen, zweier Variablen a und b, ausgibt. Die Zeigerwerte werden über den Adressoperator bestimmt und zwei (typisierten) Zeigervariablen zugewiesen. Diese werden dann anschliessend ausgegeben.

```
#include <stdio.h>

int a;           /* a ist eine globale Variable. Sie wird im Datensegment gehalten*/

main()
{int b;         /* a ist eine lokale Variable. Sie wird im Stacksegment gehalten*/
 int *pa, *pb;

 pa = &a;
 pb = &b;

 printf("Adresse der Variablen a: %p\n",pa);  /* Zeigerwert ausgeben */
 printf("Adresse der Variablen b: %p\n",pb);

 return 0;
}
```

C kennt auch untypisierte, sogenannte **generische Zeiger** vom Typ void \*. Generische Zeiger können auf alle Datentypen zeigen. Da sie aber keinen Bezug zu einem Stammtyp haben, sollten sie nicht inkrementiert oder anderswie bearbeitet werden. Generische Zeiger werden meist von Library-Funktionen als Funktionswert retourniert. Der Benutzer muss (oder sollte) dann bei der Zuweisung über einen 'Cast' den Zeiger konvertieren:

```
#include <stdio.h>
#include <stdlib.h> /* Fuer malloc() und exit() */

main()
{ double *pd;      /* pd ist ein Zeiger auf eine double */

  /* Speicherplatz fuer eine double allozieren. malloc() liefert als Resultat
     einen (void *), der nach (double *) gecastet wird */
  pd = (double *) malloc(sizeof(double));
  if (pd == NULL) /* Fehler: Kein Speicherplatz konnte bereitgestellt werden */
  { fprintf(stderr, "Fehler: Zuwenig Speicher!\n");
    exit (1);      /* Programm mit Errorlevel 1 beenden */
  }

  *pd = 1.234567E-124; /* Wert zuweisen */
  printf("Wert von *pd: %E\n", *pd); /* Wert ausgeben */

  free(pd); /* allozierten Speicherplatz wieder zurueckgeben */

return 0;
}
```

### 7.3 Anwendungen mit Zeiger

Nachdem wir bereits eine ganze Reihe von Anwendungen von Zeigern anhand von Beispielen gesehen haben, wollen wir die ganze Thematik zusammenfassen. Es gibt drei Bereiche wo Zeiger vorzugsweise verwendet werden:

1. Aus Effizienzgründen.  
Komplexe Datenstrukturen (Arrays, Structs, etc.) können über Zeiger einfacher verwaltet werden als über Direktzugriff.
2. Verwaltung von dynamischen Speicherplatz.  
Zur Programmlaufzeit kann über Betriebssystemfunktion Speicherplatz angefordert (alloziert) werden. Dieser Speicherplatz wird vollständig über Zeiger verwaltet. Die Nutzung von dynamischen Speicher ist üblich bei der Arbeit mit grossen Datenmengen.
3. Parameterübergabe bei Funktionen.  
- Übergabe von strukturierten Datentypen an Funktionen.  
- Rückgabe von Resultaten über Parameter indem der Parameter über Zeigerreferenz übergeben wird. (Sog. 'Call by Reference'-Simulation)

Nachfolgend ein Beispiel zu 1. und 2.: Verwaltung von komplexen Datentypen und dynamischen Speicherplatz:

```
/* Das Programm liest 5 Zeilen mit max. 79 Zeichen Text ein und
   weist jede Textzeile einem String zu, dessen Speicherplatz mit
   malloc() in passender Groesse alloziert wurde.
*/
#include <stdio.h>
#include <malloc.h>      /* Fuer malloc() */
#include <string.h>     /* Fuer strlen(), strcpy() */

main()
{ char *msg[5];          /* Array mit 5 Zeigern auf einen String */
  char zeile[80];      /* Zwischenspeicher (String) fuer eine Zeile */
  int i;

  printf("Geben Sie 5 Zeilen Text ein:\n");
  for (i=0; i < 5; i++)
  { scanf("%s", zeile);
    msg[i] = (char *) malloc(strlen(zeile)+1); /* Speicher fuer den String allozieren */
    if (!msg[i])
      { fprintf(stderr, "Memory allocation error..\n");
        return 1;
      }

    strcpy(msg[i], zeile); /* Text umkopieren */
  }

  /* Jetzt alle 5 Textzeilen anzeigen */
  for (i=0; i < 5; i++)
    printf("%s \n", msg[i]);

  /* Belegten Speicherplatz wieder zurueckgeben */
  for (i=0; i < 5; i++)
    free(msg[i]);

  return 0;
}
```

### Beispiel zu 3.: Parameteruebergabe an Funktionen.

```
void str_display(char *string) /* Zeiger auf die Zeichenkette */
{
  while ( *string != '\0')
    putchar(*string++);
}

#include <stdio.h>
main()
{
  str_display("Message to display\n");
  return 0;
}
```

Ganze Arrays koennen nur ueber Zeiger an Funktionen uebergeben werden. Einzelelemente hingegen koennen aber direkt als Argument uebergeben werden.

## 8 Benutzerdefinierte Datentypen

Benutzerdefinierte Datentypen werden durch den Benutzer in ihrer Gestalt definiert. Benutzerdefinierte Datentypen dienen der problemorientierten Programmierung mit der Idee logisch zusammengehörende Daten auch unter einem gemeinsamen Namen abzuspeichern.

C (und C++) kennt folgende benutzerdefinierte Typen:

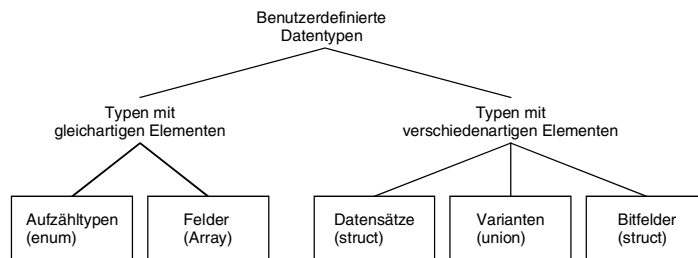


Bild 8-1: Gliederung der benutzerdefinierten Datentypen in C/C++.

### 8.1 Aufzähltypen (enums)

Enum's sind Aufzähltypen. Sie werden durch Aufzählung der Menge aller ihrer Elemente definiert. Beispiel:

```
enum wochentage
{ montag, dienstag, mittwoch, donnerstag,
  freitag, samstag, sonntag };
```

In obigem Beispiel ist `wochentage` der sog. *Tag* des Aufzähltyps. Er dient dazu um bei der Definition einer Variablen mit solchen Aufzähltypen konkret Bezug zu nehmen, ist also eine Art implizite Typdefinition. `montag` . . . `sonntag` sind die **Enumeratoren**. Sie sind Konstante Ganzzahlwerte und können im Programm auch als solche verwendet werden.

Intern werden die Enumeratoren als `int`-Werte 0, 1, 2,... gespeichert. Dies ist eine standardmässige Zuordnung, die mit einer Initialisierung geändert werden kann.

Eine Variable `heute`, die nun solche Aufzähltypen aufnehmen kann wird wie folgt definiert:

```
enum wochentage heute;
```

Ebenso kann die Definition der Variablen direkt mit der Aufzählung verbunden werden:

```
enum wochentage
{ montag, dienstag, mittwoch, donnerstag,
  freitag, samstag, sonntag } heute;
```

Der Tag kann weggelassen werden wenn er nicht benutzt wird. Allerdings kann dann keine zweite Variable mit genau dieser Aufzählung instanziiert werden.

Beachten Sie auch, dass zwei textuell gleiche Aufzählungen nicht dieselben Typen erzeugen. Es gilt generell eine Aufzählung erzeugt einen separaten Typ.

Da C doch eine recht konservative Sprache ist, werden enum's nicht gerade ausschweifend verwendet. Dies liegt daran, dass alte C-Compiler (vor ca. 1988) überhaupt keine enum's kannten. Das Aufzählprinzip wurde dort durch Preprozessoranweisungen (`#define`) realisiert, indem alle Symbole mit ihrem Wert konkret definiert wurden und die Werte dem Systemtyp `int` zugewiesen wurden. Diese Altlast wird heute noch praktiziert und es zeigt sich in Zukunft keine rasche Änderung der Methoden.

Alternative (klassische) Aufzählung über Preprozessoranweisungen:

```
#define rot 0
#define gelb 1
#define gruen 2

main()
{  int ampel;

    for (ampel = rot; ampel <= gruen; ampel++)
        printf("Ample hat Farbe Nummer: %d\n",ampel);

    return 0;
}
```

Als letzte Variante können alle Typen, also auch Aufzählungen, über die Typedef-Anweisung definiert werden.

## 8.2 typedef-Anweisung

Das Wort `typedef` ist ein reserviertes Schlüsselwort in ANSI-C (C++). Es dient dazu explizit Datentypen zu definieren.

Über Tags konnte schon vorher auf mehr oder weniger transparente Art und Weise eine Menge von Datentypen definiert werden, aber nicht alle. ANSI-C liefert die Möglichkeit einer expliziten und vor allem für alle Datentypen gleiche Art der Typendefinition. `typedef` geht also viel weiter als das Konzept der Tags für Struktur- und Aufzähltypen.

Technisch gesehen wird mit `typedef` das 'Konstruktionsmuster' für eine später zu erzeugende Variable festgelegt.

Merke:            Eine Typendefinition erzeugt selbst noch keine Variable, d.h. braucht noch keinen Speicherplatz im Programm!

Beispiele für Typendefinitionen:

```
typedef enum {rot, gelb, gruen} ampel_typ;
typedef double gleitkomma;
typedef unsigned long int UBIG;
```

Das `typedef` ist heute die Standardanweisung für Datentypen zu definieren. Da aber in Dokumentationen und Beispielen alle Möglichkeiten angetroffen werden, wurden (fast) alle gezeigt.

## 8.3 Komplexe Datentypen (Aggregates und Arrays)

Sie dienen vor allem der problemorientierten Programmierung mit der Idee, dass logisch zusammengehörende Daten auch unter einem Namen (Symbol) zusammengefasst werden können.

In C (und C++) unterscheidet man grundsätzlich zwischen zwei verschiedenen Sorten von komplexen Datentypen:

Array:	Folge von $n$ gleichartigen Datentypen
Struct:	Verbund von verschiedenen Datentypen (Aggregates)

Jeder komplexe Datentyp kann wiederum aus komplexen Datentypen bestehen. In der tiefsten Ebene sind aber alle komplexen Datentypen mit Systemtypen definiert. (Prinzip der rekursiven Definition.)

Wir werden komplexen Typen in einer ersten Annäherung in den Arrays und Structs betrachten. Später erfolgt eine Festigung mit den Spezialitäten Union's und Bitfelder.

## 8.4 Arrays

Arrays sind Folgen von  $n$  gleichartigen Elementen. Man nennt sie in anderen Gebieten auch Vektoren, Matrizen oder Felder. Alle Elemente eines Arrays werden fortlaufend in einem zusammenhängenden Speicherblock abgelegt.

Die Arrays können eindimensional (Vektor) oder mehrdimensional (Matrix,) sein.

Arrays können aus System- und benutzerdefinierten Typen definiert werden. Die Definition erfolgt durch Angabe der Anzahl Elemente in eckigen Klammern:

```
int a[10];           /* Ein Array mit 10 int-Elementen */
float m[3][5];      /* Matrix mit 3 Zeilen und 5 Spalten */
char c[6]="Hallo";  /* Array mit 6 char's, das gerade initialisiert wird */
char c[]="Hallo";   /* dito, der Compiler berechnet aber selbst die Groesse */
```

Das Ansprechen eines Elementes aus einem Array erfolgt immer über **Indizierung** mittels Indexklammer. Der Index ist immer eine positive Ganzzahl von  $0..n-1$ , wobei  $n$  die Anzahl Elemente in der entsprechenden Komponente sind.

```
/* Musterbeispiel zum Arbeiten mit Arrays */
#include <stdio.h>

#define DIMENSION 102           /* Anzahl Elemente im Array */
#define ANZ_PRO_ZEILE 5        /* Anzahl Elemente die bei der Ausgabe einer Zeile */
/*                               */

main()
{
    int a[DIMENSION];
    int i,j,index;

    for(i=0; i< DIMENSION; i++) /* Array fortlaufend mit Zahlen 0..DIMENSION einfüllen */
        a[i] = i;
    index=0;
    for(i=0; i< DIMENSION/ANZ_PRO_ZEILE; i++) /* Jedes Element des Array ausgeben */
    {
        printf("Zeile:%-4d",i);
        for (j=0;j < ANZ_PRO_ZEILE; j++)
            printf("\t%d",a[index++]);
        printf("\n");
    }
    /* Ev. Rest ausgeben falls Anzahl nicht vollstaendig teilbar ist*/
    if (DIMENSION % ANZ_PRO_ZEILE)

```

```

    { printf("Zeile:%-4d",i);          /* Sehen Sie hier eine Unsauberheit? */
      for (j=0;j < (DIMENSION % ANZ_PRO_ZEILE); j++)
        printf("\t%d",a[index++]);
    }
}
return 0;
}

```

Merkwürdig aber auf den zweiten Blick verständlich ist das folgende Beispiel:

```

int a[10]=[1,2,3,4,5,6,7,8,9,10};
int i;

for (i=0;i<10;i++)
    printf("%d",i[a]);

```

Hier wird Index und Array vertauscht. Da aber das Array mit dem Namen die Basisadresse als Zeiger verkörpert und der Index als int das gleiche Zahlenformat wie ein Zeiger hat, funktioniert dies. Die Arrayklammer stellt im Wesentlichen eine simple Addition von Basisadresse und Index dar.

Merke:

Indexüberläufe (Überschreitungen) werden weder zur Programmlaufzeit überprüft noch erkannt. Es wird einfach das im Speicher nächste Element gelesen oder schlimmer: beschrieben.

Ebenso werden negative Indizes nicht kontrolliert. Auch wird einfach die entsprechende Speicherzelle angesprochen, was natürlich falsch ist.

## 8.5 Strings

Strings sind spezielle Arrays. Sie sind vom Grundtyp char und können bis maximal 64k-1 gross sein. Strings haben am Ende der Zeichenkette immer einen sog. String Terminator, das Nullzeichen '\0'. Aufgrund dieses Nullzeichen weiss der C-Compiler wann ein String zu Ende ist.

Im Speicher wird also der String folgendermassen abgelegt:

```
char msg []="message";
```

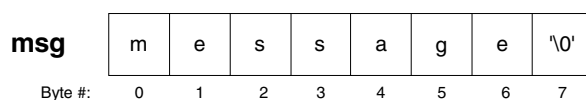


Bild 8-2: Darstellung eines C-Strings im Speicher.

Für Strings kennt C eine ganze Menge von Bearbeitungsfunktionen wie Vergleichen, Verketteten, Absuchen, etc.

Als Beispiel sei gezeigt wie zwei Strings verglichen werden können:

In C kann man nicht direkt den Inhalt zweier Arrays, also auch Strings miteinander vergleichen. Dies muss durch Vergleichen aller Einzelelemente geschehen. Die Funktion `strcmp()` macht genau dies.

```
#include <stdio.h> /* Fuer printf() */
#include <string.h> /* Fuer strcmp() */

main()
{ char eingabe[50]; /* Platz fuer einen String mit max 50 Zeichen (inkl. Terminator) */
  int ende;

  printf("Das Programm wird genau dann beendet wenn Sie 'beenden' eingeben:\n");
  do {
    scanf(" %s",eingabe);
    if (strcmp(eingabe,"beenden") == 0) /* Ist der eingegebene String == beenden ? */
      ende = 1;
    else
      { printf("\nKeine korrekte Eingabe. Versuchen Sie es erneut!\n");
        ende = 0;
      }
  } while (!ende);

  return 0;
}
```

## 8.6 Allgemeine Bemerkungen zu Arrays

Die Elemente eines Arrays werden zeilenweise gespeichert (Row Major). Der am weitesten rechts stehende Index variiert am schnellsten.

So hat ein Array `int a[2][3]` die Form:

a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]

Bild 8-3: Format und Indizierung eines 3x2 Arrays.

Im Arbeitsspeicher wird das Array in fortlaufender Reihenfolge der Zeilenelemente abgelegt:

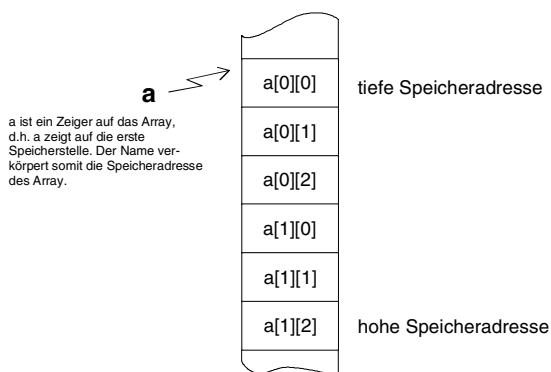


Bild 8-4: Darstellung eines zweidimensionalen Arrays im Speicher.

Die Kenntnis, wie Arrays im Speicher abgelegt werden, ist in einer Testphase von Nutzen. So kann man überprüfen ob die Elemente richtig wunschgemäß im Speicher abgelegt werden.

Man beachte, dass der Name des Arrays selbst eine Konstante ist. Er ist genaugesehen ein Zeiger, der die Speicheradresse des Arrays verkörpert.

### Technische Anmerkung:

Im PC verstehen wir unter Speicheradresse die Adresse relativ zum Datensegment für globale Variablen und relativ zum Stacksegment für lokale Variablen. Die Adresse wird für Zeigertypen im Debugger ausgewiesen und kann bei Interesse kontrolliert werden.

Daraus folgt auch, dass Arrays nicht mit der Wertzuweisung '=' kopiert werden können. Sollen Arrays kopiert werden, so muss dies Elementweise geschehen. Dies kann mit Schleifen erfolgen (klas-

sich) oder mittels Speicherkopierbefehlen (Profi-Like). Speicherkopierbefehle kopieren ganze Speicherbereiche auf Maschinenebene (oft mit DMA) unabhängig vom Inhalt. Dazu werden die jeweiligen Ziel- und Quelladressen gegeben und die Anzahl der zu kopierenden Bytes.

```
/* Beispiel zum Kopieren von Arraydaten.

Gerhard Krucker
16.10.2003
Intel C++ V7.1
*/

#include <stdio.h>    /* Fuer printf() */
#include <memory.h>  /* Fuer memcpy() */

void show(int a[][3]) /* Anzeigen eines 3x3-Arrays, das ueber a uebergeben wurde */
{ int i,j;           /* Anmerkung: Man haette a auch als **a definieren koennen */

  for (i=0; i < 3; i++)
  { for (j=0; j < 3; j++)
    printf("%4d",a[i][j]);
    printf("\n");
  }
}

main()
{ int a[3][3]={1,2,3},{4,5,6},{7,8,9}};
  int b[3][3];
  int c[3][3];
  int i,j;           /* Indexvariablen zum Kopieren */

  printf("Urspruengliches initialisiertes Array a;\n");
  show(a);

  /* Kopieren des Array a nach b, Element um Element */
  for (i = 0; i < 3; i++)
    for (j = 0; j < 3; j++)
      b[i][j]=a[i][j];

  /* Kopieren mit memcpy() Library-Funktion: a nach c */
  memcpy(c,a,sizeof(a));

  /* und beide Arrays anzeigen */
  printf("\nArray b, elementweise kopiert:\n"); show(b);
  printf("\nArray c, mit memcpy() kopiert:\n"); show(c);

  return 0;
}
```

## 8.7 Structs

Structs sind ebenfalls benutzerdefinierte Datentypen, jedoch mit dem Unterschied, dass sie verschiedenartige Typen aufnehmen können. Vom Konzept her gesehen, sind sie mit den RECORDS in PASCAL zu vergleichen. Sie dienen wie Arrays zur problemorientierten Programmierung mit der Idee, dass logisch zusammengehörende Daten als gemeinsame Einheit gespeichert werden sollen. Aus abstrakter Sicht verkörpern Struct's Datensätze, wobei jeweils ein Satz verschiedene Teilinformationen beinhaltet: (sog. *heterogene Datenstruktur*)

Beispiel:

```
typedef struct {
    float wert;
    TIME zeit;
    unsigned int messgeraet;
} MESSUNG;
```

Die Einzelelemente (Komponenten) werden über Selektion angesprochen. Die Selektion erfolgt mit dem *Selektorpunkt*:

```
#define HP3401 0
#define FLUKE 1
#define SCHLUMBERGER 2
....
MESSUNG mw;          /* Definition einer struct-Variablen */
....
mw.wert=1234.789f;
getTime(&mw.zeit);   /* Element (resp. Adresse) eines struct ist Parameter */
mw.messgeraet=HP_3401;
.....
```

Technisch gesehen, werden Struct's auch als ganzer, zusammenhängender Block im Speicher gehalten, ähnlich einem Array.

Nun zur Definition: Wie bei den Enums sind verschiedene Arten zur Definition von Structs möglich:

### 8.7.1 Klassische Art der Definition nach Kernighan & Ritchie

Definition erfolgt mit dem reservierten Wort `struct`, gefolgt von einem Tag mit anschließender Definition der Komponenten.

Diese noch häufig anzutreffende Art wurde aus der ursprünglichen Sprachdefinition von **Kernighan & Ritchie** [KER83] aus den 70er Jahren übernommen. Zu diesem Zeitpunkt gab es noch keine explizite Anweisung für eine Typendefinition. Die Referenz zum Typ wurde über das **Tag** (dt.: Etikett, Schildchen) vorgenommen. Der Tag verkörpert also eine implizite Typendefinition

Allgemeines Format für eine `struct`-Definition:

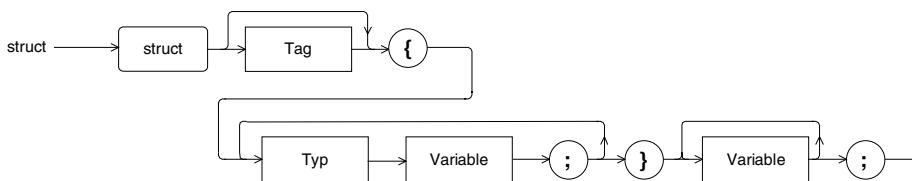


Bild 8-5: Definition von Structs nach klassischer Art

Die Variablen innerhalb des Blockes sind die Komponenten des `struct`. Die Variablen am Schluss (ausserhalb des Blockes) sind die instanziierten Variablen dieses Typs.

Beispiel:

```
struct datum { int tag;
               char monat[10];
               int jahr;
             };

struct person { char name[20];
               char vorname[20];
               struct datum gebdatum;
               char strasse[30];
               ..int hausnummer;
               int plz;
               char wohnort[15];
             };
```

Diese Deklaration mit Tag liefert also eine Art 'Konstruktionsmuster', (als Datentyp) für eine Variablendefinition:

```
struct person ich, du;           /* Definiert die struct-Variablen ich, du */
```

Der Zugriff erfolgt über **Selektion** mit dem Selektorpunkt:

```
ich.name[0]
du.plz
ich.gebdatum.jahr
```

ANSI-C erlaubt bei der Definition von Struct's das Weglassen des Wortes `struct`, wenn die Struktur bereits deklariert worden ist. Somit ist folgende Variablendefinition mit obiger identisch:

```
person ich, du;                 /* Definiert die struct-Variablen ich, du */
```

### 8.7.2 Neue Art nach ANSI

Definition über die Datentypen mit der `typedef`-Anweisung. Diese Art ist der klassischen Definition vorzuziehen. `typedef`-Anweisung ermöglicht eine einheitliche Art der Typendefinition.

Die `typedef`-Anweisung erzeugt jedoch selbst keine Variablen. Sie ist rein *deklarativen* Charakters. Deklarativ heisst vereinbarend an den Compiler im Sinne „ich werden dann so verwenden..“. Die eigentliche Definition von Variablen erfolgt nachher unter Bezugnahme der definierten Datentypen.

Für das vorherige Beispiel wird dies:

```
typedef struct { int tag;
                char monat[10];
                int jahr;
            } DATUM;

typedef struct { char name[20];
                char vorname[20];
                DATUM gebdatum;
                char strasse[30];
                ...int hausnummer;
                int plz;
                char wohnort[15];
            } PERSON;
```

`typedef` definiert nun die benutzerdefinierten Datentypen `person` und `datum`. Das Struktur-Tag fehlt hier, da es nicht benutzt wird. Ein Tag kann in solchen Definitionen gleichwohl verwendet werden. Dies wird dann notwendig, wenn man im Struct einen Bezug auf sich selbst machen muss:

```
typedef struct node { ITEM info;
                    node *next;    /* Zeiger auf den naechsten Knoten */
                } NODE;
```

Solche Typendefinitionen sind üblich um Elemente (sog. Knoten) für Baumstrukturen zu definieren. Dabei wird dann mittels geeigneten Verknüpfungen mit den Zeigern die jeweilige Struktur zur Programmlaufzeit erzeugt. Solche Strukturen werden praktisch immer in der Datenverarbeitung in Form von linearen Listen, binären Bäumen und anderen baumartigen Anordnungen gebraucht.

Structs können bei der Definition einer Variablen auch gerade einen Initialwert erhalten. Dies erfolgt in üblicher Manier. Soll ein Element des Structs nicht initialisiert werden, jedoch Nachfolgende wieder, so kann mit Kommata das entsprechende Feld übersprungen werden. Für unser Beispiel lautet eine konkrete Definition mit Teilinitialisierung:

```
PERSON ich={"Krucker", "Gerhard"};
```

Hier wurden nur die Komponenten `name` und `vorname` initialisiert. Der Rest hat im Falle einer auto-Variablen zufällige Werte, die vor einem Zugriff mit geeigneten Werten beschrieben werden müssen.

Ein Gesamtbeispiel:

```
/* Benutzerdefinierte Funktion zum Anzeigen der Daten eines Struct's auf
   dem Bildschirm. Die Daten werden dabei als Parameter an die Funktion
   uebergeben. Sie sind nachher in 'person' als Lokalvariable zugreifbar
*/

void zeigeDaten(PERSON person)
{ printf("Name: %s\n", person.name);
  printf("Vorname: %s\n", person.vorname);
  printf("Geburstdatum: %d. %s\n",
         person.gebdatum.tag, person.gebdatum.monat, person.gebdatum.jahr);
  printf("Strasse: %s %d\n", person.strasse, person.hausnummer);
  printf("Plz: %d\tWohnort: %s\n", person.plz, person.wohnort);
}

int main()
{ PERSON ich;
  static PERSON du={"Krucker", "Gerhard"};

  zeigeDaten(du); /* Daten des initialisierten Struct anzeigen */

  /* Daten erfassen */
  printf("\nEingabe des Namens: "); scanf("%s", ich.name);
  printf("Eingabe des Vornamens: "); scanf("%s", ich.vorname);
  printf("Eingabe des Geburstdatum (Bsp: 11 Maerz 1944): ");
  scanf("%d %s %d", &ich.gebdatum.tag, ich.gebdatum.monat, &ich.gebdatum.jahr);
  printf("Eingabe der Strasse: "); scanf("%s", ich.strasse);
  printf("Eingabe der Hausnummer: "); scanf("%d", &ich.hausnummer);
  printf("Eingabe der Postleitzahl: "); scanf("%s", ich.plz);
  printf("Eingabe des Wohnortes: "); scanf("%s", ich.wohnort);

  zeigeDaten(ich); /* Erfasste und eingespeicherte Daten anzeigen */

  return 0;
}
```

## 8.8 Ergänzungen

Die bis hierhin gezeigten Datentypen sind die wesentlichen Typen, die wir später zur Arbeit verwenden werden. C bietet jedoch noch Spezialitäten auf die wir jedoch nicht weiter eingehen werden:

- Unions
- Bitfelder

Sie werden hier nur vom Prinzip erklärt, so dass man etwa weiss, um was es sich grundsätzlich handelt. Meist ist der Gebrauch von solchen Typen durch das Umfeld vorgegeben. So werden Unions häufig bei Betriebssystemaufrufen benutzt.

### 8.8.1 Unions

Unions sind Varianten. Das heisst, mehrere Variablen benutzen denselben Speicherplatz. So kann beispielsweise ein Prozessorregister einmal als 16-Bit Register (Bsp.: AX) und ein anderes mal als zwei 8-Bit Register (Bsp.: AH, AL) beschrieben werden. In der Tat wird aber in beiden Fällen derselbe Speicherplatz, hier das Prozessorregister, beschrieben. Nur eben in verschiedenen Varianten.

Die Idee war in früherer Zeit, dass sich gegenseitig ausschliessende Datentypen durchaus aus Gründen der Speicherplatzersparnis physikalisch den gleichen Speicherplatz belegen können. Der Platzbedarf wird dann auf die grösste Variante ausgerichtet. Dieses Konstrukt ist nicht etwa C-typisch, andere Sprachen kennen solche Konzepte auch. Z.B. PASCAL mit den Varianten Records.

Von der Definition her sehen die Unions den Structs sehr ähnlich. Auch der Zugriff erfolgt genau gleich.

Beispiel:

```
union { char c;  
        int i;  
        } z;
```

Hier kann die Union-Variable z also entweder eine char oder eine int aufnehmen. Der Benutzer muss im Programmablauf sich selber im Klaren sein in welcher Variante er zugreifen will.

### 8.8.2 Bitfelder

**Bitfelder** sind eine Spezialität aus der maschinennahen Programmierung. Man kann hierbei für eine einzuspeichernde Grösse die genaue Anzahl zu verwendenden Bits vorgeben. So ist dieses Verfahren prädestiniert um eine logische Grösse (Bsp. ein Flag mit TRUE / FALSE), die nur zwei Zustände annehmen kann, in einem einzelnen Bit zu speichern.

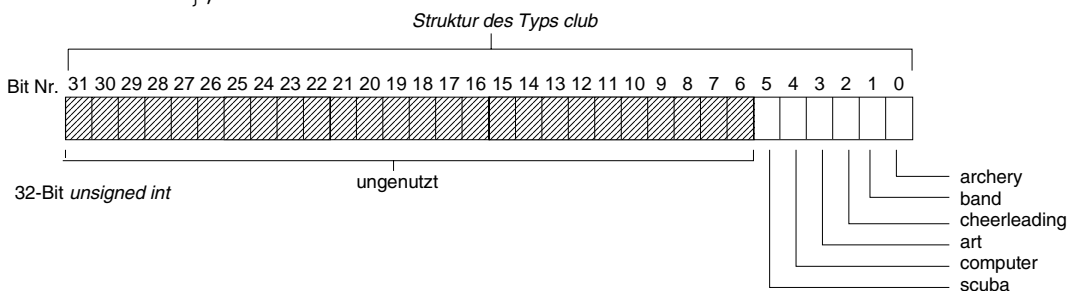
Dadurch kann (im Prinzip) eine Menge Speicherplatz gespart werden. Da aber die Prozessoren Wortweise auf den Speicher zugreifen, sind bitweise Operationen in der Regel recht ineffizient, so dass dieses Verfahren nur in Ausnahmen angewandt wird.

Nachfolgend ein Beispiel für ein Bitfeld. Die Anzahl zu verwendeten Bits wird in Doppelpunkten hinter der jeweiligen Komponente spezifiziert:

Beispiel:

```
struct club { unsigned int archery:1;  
             unsigned int band:1;  
             unsigned int cheerleading:1;  
             unsigned int art:1;  
             unsigned int computer:1;  
             unsigned int scuba:1;  
             };
```

Bild 8-6: Speicherformat eines Bitfeldes.



Die gesamte Anzahl Bits darf pro Komponente Wortbreite eines Prozessorregisters nicht überschreiten.

## 9 Funktionen

Funktionen dienen zum Zusammenfassen von Programmcode zu einer logischen Einheit. Sie sind in der Sprache C das wichtigste Element zur **Abstraktion** von Funktionalität. Funktionen sind in diesem Sinn eine generalisierte Lösung für eine Klasse von speziellen Problemen.

Die Definition einer Funktion bietet sich also dort an, wo etwa das gleich mehrfach im Programm gemacht werden soll. Die Feinauswahl was genau gemacht wird, erfolgt über **Parametrisierung**.

Mittels Funktionen kann ein komplexes Programm strukturiert werden. In der Sprache C ist dies eigentlich die einzige Möglichkeit, abgesehen von Modulen, Ablaufcode logisch zusammenzufassen.

In C/C++ wird eine Funktion über ihren Namen aufgerufen und führt als Subroutine die Instruktionen aus. Über Parameter können der Funktion Argumente mitgegeben und so das Verhalten der Funktion in Ihrem Ablauf gesteuert werden. Die Rückgabe eines Resultates ist in Form eines Funktionswertes an den Aufrufer möglich.

Eine Funktion kann nur global im aktuellen Modul definiert werden. Die Definition in der Idee „lokaler Funktionen“ ist in C/C++ nicht möglich. Dies ist aber kein Nachteil. Der Gültigkeitsbereich einer Funktion ist immer von der Definition/Deklaration an bis zu Ende des aktuellen Moduls.

### 9.1 Definition von Funktionen

Funktionen werden mit Resultattyp, Namen, Parameter und Funktionskörper definiert:

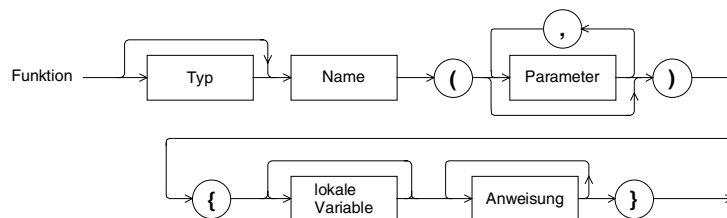


Bild 9-1: Syntaxdiagramm für die Definition von Funktionen in C/C++.

Der Name für die Funktion ist frei wählbar im Rahmen der Namensgebung für C-Symbole. Zu beachten ist aber:

1. Das Hauptprogramm von wo das Programm startet heisst immer `main()`. (bei WINDOWS-EXE: `WinMain()`.)
2. Wird ein Funktionsname definiert, der bereits als Library-Funktion existiert, so geht die ursprüngliche Library-Funktion verloren. Die eigene Definition hat also Vorrang.
3. Werden in einem gleichen Projekt zwei Funktionen mit dem gleichen Namen definiert, die aber in unabhängigen Modulen sind, kommt es zur Linkzeit zu einer entsprechenden Fehlermeldung.

### 9.1.1 Typ der Funktion

Unter *Typ* bei der Funktionsdefinition verkörpert den Datentyp des Resultates, den die Funktion mit `return` zurückliefert.

Beispiel:

```
int abs(int x)
{ if (x < 0)
  return -x;
  else
  return x;
}
```

Diese Funktion ist vom Typ `int` und liefert ein ganzzahliges Resultat als Funktionswert der aufrufenden Einheit zurück.

C kennt für den Typ zwei Eigenheiten:

1. Wird kein Resultattyp angegeben, so wird standardmässig `int` als Resultattyp angenommen.
2. Soll die Funktion **kein Resultat** zurückliefern können, wird eine Funktion vom Typ `void` definiert. Das Wort `void` besagt, dass diese Funktion kein `return` mit einem Wert machen wird. (Anm: Ein `return` ohne Wert wäre auch erlaubt.)

Beispiel:

```
void ausgabe()
{ printf("Hallo!\n");
}

main()
{
  ausgabe();
  return 0;
}
```

`void`-Funktionen entsprechen *Prozeduren* im Sinne von PASCAL oder Delphi.

Typenbezogene Funktionen liefern immer einen Wert zurück der verwertet werden sollte. Dies erfolgt entweder in einem Ausdruck oder in einer Zuweisung. Prozeduren stehen als alleinige Instruktion.

Als Typ für eine Funktion sind alle Systemdatentypen, Zeiger, Enums und Structs erlaubt.

### 9.1.2 Parameter

Der Funktion können Argumente übergeben werden. Diese Argumente werden *C Parameter* genannt und sind innerhalb der Funktion zugreifbar. Sie sind eine Art spezielle Lokalvariablen, die mit den entsprechenden Argumenten des Aufrufers initialisiert werden.

Der Informatiker spricht hier auch von formalen und aktuellen Parametern. Die in der Funktion selbst definierten Parameter heißen *formale Parameter*. Die beim Aufruf mitgegebenen Parameter heißen *aktuelle Parameter*. Beim Aufruf erhält dann der formale Parameter eine Kopie des aktuellen Parameters:

```
int addiere(int a, int b)
{ int summe;

  summe = a + b;
  return summe;
}

....
c = addiere(y,2);
```

Formale Parameter

Aktuelle Parameter

Bild 9-2: Aktuelle und formale Parameter beim Funktionsaufruf in C/C++.

Die Anzahl der Parameter für eine Funktion ist beliebig. Wir können auch eine Funktion ohne Parameter definieren. Aus syntaktischen Gründen muss trotzdem eine leere Klammer stehen. Man kann in die leere Klammer auch `void` schreiben, wenn man damit unterstreichen will, dass keine Parameter übergeben werden.

Beispiel:

```
void ohneParameter()
{ printf("Dies ist eine Funktion ohne Parameter");
}

...
ohneParameter();
....
```

Bemerkung:

Der Name einer Funktion ohne Klammern verkörpert einen Zeiger auf die Funktion, ähnlich bei einem Array. Dieser Zeiger zeigt dann im Speicher auf den Beginn des Maschinencodes der Funktion.

Der formale Parameter in der Funktion erhält immer eine Kopie des aktuellen Parameters. Eine Veränderung innerhalb der Funktion hat also keine Auswirkung auf den aktuellen Parameter. Dieses Prinzip, wo eine Kopie des Parameters übergeben wird, nennt man **Call by Value**.

Als Parameter sind alle Systemdatentypen, Aufzähltypen, Structs und Zeiger erlaubt. Ganze Arrays können nicht direkt übergeben werden. Dies erfolgt über einen Zeiger auf das Array.

Wird der Funktion ein einzelner Parameter übergeben, so wird dieser in der Parameterklammer notiert. Dazu ein Beispiel:

```
void show_digit(int digit)
{
    printf("%d\n", digit);
}

main()
{
    show_digit(6);
    return 0;
}
```

Mehrere Parameter werden mit Kommata getrennt. Bei einer grossen Anzahl kann auch auf die nächste Zeile übertragen werden.

Beispiel:

```
void display_sum(int a, int b)
{
    printf("%d + %d = %d\n", a, b, a+b);
}

main()
{
    display_sum(200, 37);
    return 0;
}
```

Die Funktion `display_sum` erhält zwei Argumente `a` und `b`. Diese sind in der Funktion Lokalvariablen und werden in derselben Reihenfolge mit den aktuellen Parametern initialisiert:

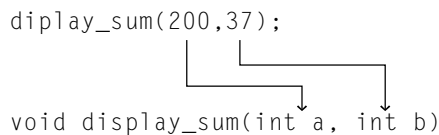


Bild 9-3: Übergabe der Werte der aktuellen an die formalen Parameter beim Funktionsaufruf in C/C++.

**Die Datentypen für formale und aktuelle Parameter müssen übereinstimmen**, d.h. sie müssen gleich oder mindestens zuweisungskompatibel sein. Zuweisungskompatibel heisst, dass die verschiedenen Typen ohne Cast einander zugewiesen werden können. Sonst gibt es beim Kompilieren (meist) eine Fehlermeldung. In manchen Fällen kann der Compiler Typenkonflikte in Parameter nicht erkennen. Dies führt dann zu Fehlern im Programm, wo die Funktion auf "unerklärliche Weise" die Parameter verliert.

### 9.1.3 Parameterübergabe über Zeiger

Oft möchte man in einer Funktion mehr als einen Wert zurückgeben oder den aktuellen Parameter in der Funktion persistent verändern können. Eine Funktion kann aber nur einen Wert als Resultat über den Funktionswert retournieren. Eine Beeinflussung des aktuellen Parameters in einer Funktion ist ebenfalls nicht möglich. Aus der Sicht der strukturierten Programmierung sind diese Einschränkungen durchaus sinnvoll.

Trotzdem wäre es in einigen Fällen wünschenswert, wenn man über Parameter auch Werte retournieren könnte. Dies ist in C/C++ wenn anstatt der aktuelle Parameter mit Kopie des Wertes eine Zeigerreferenz übergeben wird. In der Funktion wird dann über diese Zeigerreferenz indirekt auf den aktuellen Parameter zugegriffen. So kann indirekt auch schreibend auf den aktuellen Parameter zugegriffen werden. Mit diesem Prinzip können über Parameter Resultate zurückgeben werden. Dieses Prinzip wird *Call by Reference* genannt.

C++ kennt auch *Referenzparameter*. Diese erlauben direkt eine zeigerfreie Notation eines Call by Reference.

Eine andere Notwendigkeit für Zeiger ist die Übergabe von Arrays an eine Funktion. Hier wird ebenfalls nur der Zeiger übergeben. In der Funktion erfolgt dann der Zugriff auf die einzelnen Elemente in bekannter Form.

Hierzu ein Beispiel, dass aus einem Array mit Datenelementen den Mittelwert und Varianz bestimmt. Diese beiden Größen werden über Parameter zurückgegeben:

```

/* Beispiel für Parameteruebergabe ueber Zeiger.                                     File:MW_VAR.C
   Bestimmen des Mittelwertes und der Varianz der Datenelemente
   in einem Array mit n Elementen Laenge.

   Das Array wird mit Zeigerreferenz (Arrayname selbst) an die Funktion uebergeben.
   mw und var sind Parameter, die ueber Zeiger uebergeben werden. Ueber die Zeiger-
   referenz werden dann die Resultate zurueckgegeben.
   n definiert die Anzahl Elemente im Array und wird normal mit einem Call by Value
   uebergeben
*/
#include <stdio.h>

void mw_var(float *daten, float *mw, float *var, int n)
{ int i;
  float sum;

  /* Mittelwert bestimmen */
  sum = 0;
  for (i = 0; i < n; i++)
    sum += daten[i];
  *mw = sum / n;          /* Mittelwert ueber Zeigerreferenz der Variablen mw zuweisen */

  /* Varianz bestimmen */
  sum = 0;
  for (i = 0; i < n; i++)
    sum += (daten[i] - *mw) * (daten[i] - *mw);
  *var = sum / (n-1);    /* Varianz ueber Zeigerreferenz der Variablen var zuweisen
*/
}

main()
{ float daten[]={1.0f,2.0f,3.0f,4.0f};
  int anzahl=4;
  float mw;
  float var;

  mw_var(daten,&mw, &var, anzahl);
  printf("Mittelwert: %f\nVarianz: %f\n",mw, var);

  return 0;
}

```

### 9.1.4 Der Funktionskörper

Nach dem Funktionskopf mit Typ, Name und Parameter folgt der eigentliche Funktionskörper. Er beinhaltet in Blockklammern den eigentlichen Code der Funktion.

Im Funktionskörper können Lokalvariablen definiert werden. Diese Lokalvariablen sind dann nur in der entsprechenden Funktion unter diesem Namen bekannt gültig. Hat eine Lokalvariable denselben Namen wie eine global Variable, hat die lokale Variable Vorrang.

Im Funktionskörper können beliebige C-Statements ausgeführt werden. Eine Einschränkung gilt aber: Funktionsdefinitionen können nicht verschachtelt werden. Das heißt innerhalb einer Funktion können wir keine neue Funktion definieren (im Sinne einer lokalen Funktion wie in PASCAL). Dies ist aber von der Praxis her gesehen überhaupt keine Einschränkung.

### 9.1.5 Funktionsprototyp

Grundsätzlich gilt in C, dass nur das verwendet werden kann was vorgängig definiert wurde. Nun stellt sich die Frage: Wie verwende ich etwas was erst später definiert wird?

C bietet hierzu eine Lösung an: Den Funktionsprototypen. Er besteht aus dem Funktionskopf, also Typ, Namen und Parameter der Funktion. Durch Angabe des Funktionsprototypen teilt man dem Compiler mit, wie die Funktion und vor allem ihre Schnittstellen, zum Aufrufer beschaffen sind. Dadurch kann der Compiler wichtige Prüfungen bezüglich Typen der Parameter und Resultate vornehmen.

Funktionsprototypen bestehen wie gesagt nur aus dem Funktionskopf. Dazu muss Typ, Name und Typ der Parameter angegeben werden:

Beispiel: Die Funktion addiere

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

hat den Prototyp:

```
int addiere(int, int);
```

Die Namen der formalen Parameter werden im Prototyp nicht genannt. (Lässt man die Namen stehen, führt dies zu keinem Fehler). Wichtig ist, dass der Prototyp mit ; abgeschlossen wird.

Funktionsprototypen werden zwingend dort gebraucht wo man eine Funktion aufruft, die bis anhin noch nicht definiert worden ist.

Beispiel:

```
void display_sum(int, int);

main()
{
    display_sum(200, 37);
    return 0;
}

void display_sum(int a, int b)
{
    printf("%d + %d = %d\n", a, b, a+b);
}
```

Hier brauchen wir den Prototypen von `display_sum`, weil der Funktionsaufruf vor der Definition erfolgt.

Wir können den Funktionsprototyp auch lokal angeben. Das hätte dann zur Folge, dass `display_sum` nur in `main()` bekannt wäre:

```
main()
{ void display_sum(int, int);

  display_sum(200, 37);
  return 0;
}

void display_sum(int a, int b)
{
  printf("%d + %d = %d\n", a, b, a+b);
}
```

Eine Frage könnte sich jetzt stellen: Wozu brauchen wir Funktionsprototypen? Wir können doch einfach alles vorher definieren!

Zwei Begründungen, dass dies nicht reicht:

1. Alle Bibliotheksfunktionen sind normale Funktionen. Uns ist bekannt, dass wir vor der Benutzung einer solchen Funktion das entsprechende Header-File (Bsp. `stdio.h`) einbinden müssen. Die Headerfiles beinhalten nicht den gesamten Quellcode der Funktion. Dies wäre unwirtschaftlich, sondern nur die Prototypen.

2. Betrachten Sie folgenden Konstrukt:

```
int functionA(int a)
{ if (a > 0)
  functionB(a+1);
  return a;
}

int functionB(int b)
{ if (b > 0)
  functionA(b-2);
  return b;
}
```

Hier rufen wir in `functionA` eine Funktion auf, die erst nachfolgend definiert ist. Dies ist nicht ohne Funktionsprototypen möglich. In diesem Fall muss also zwingend vor dem Aufruf in `functionA` ein Funktionsprototyp für `functionB` definiert werden.

Konkret sähe das für unser Beispiel so aus:

```
/* Einfache indirekte Rekursion                               File: REKURS2.C
   Autor: Gerhard Krucker
   Datum: 10.2.1995
*/

#include <stdio.h>

int functionB(int b);

int functionA(int a)
{ if (a > 0)
  { printf("FunktionA mit a=%d\n", a);
    functionB(a+1);
  }
  return a;
}
```

```
int functionB(int b)
{ if (b > 0)
  { printf("FunktionB mit b=%d\n",b);
    functionA(b-2);
  }
  return b;
}

main()
{ int k=3;

  functionA(k);

  return 0;
}
```

Einige mögen den Ablauf dieses Programmes mit Befremden ansehen: Hier rufen sich gegenseitig zwei Funktionen auf. Dieses Prinzip, wo sich eine Funktion selbst aufruft oder mehrere Funktionen untereinander wechselseitig, nennt man Rekursion.

## 9.2 Rekursion

Die Rekursion ist eines der wesentlichen Konzepte der Informatik. Bei der rekursiven Lösung einer Aufgabe wird eine komplexe Aufgabe in eine strukturell gleiche, aber etwas einfacher zu lösende Aufgabe zerlegt. Dies wird sooft wiederholt, bis die Lösung trivial ist. Dies ist dann das Abbruchkriterium. Dann werden alle Teillösungen zu einer Gesamtlösung zusammengesetzt.

Dieses im Prinzip bestehend einfache Verfahren eignet sich jedoch nur für ganz bestimmte Lösungsansätze. Nämlich solche, wo die Lösung bereits rekursiv definiert ist. Das Standardbeispiel hierzu ist die Berechnung der Fakultät:

$$n! := \begin{cases} 1 & n=0, n=1 \\ n(n-1) & n > 1 \end{cases}$$

Diese Funktion kann man in C formulieren:

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

Die rekursive Lösung ist meist, von der Formulierung her, sehr elegant. Rekursive Funktionen sind jedoch in der Ausführung nicht besonders effizient, da alle Zwischenresultate inklusive Rückkehradressen für die Funktionsaufrufe im Speicher gehalten werden müssen, was bei grösseren Rekursionstiefen schnell viel Stackspeicher verbraucht.

Die meisten effizienten Standardverfahren (Algorithmen) für Suchen und Sortieren sind rekursive Methoden. Hier bringt der Einsatz der Rekursion eine wesentliche Verbesserung. Die ist aber in der Methode selbst begründet.

Ein etwas anderes Beispiel für Rekursion ist die Ausgabe eines Strings. Hier wird beginnend beim ersten Zeichen rekursiv ein Zeichen ausgegeben, bis das Ende des Strings ('\0'-Zeichen) erreicht wird.

```

/* Zeichenweise Ausgabe eines Strings mit Rekursion                               File: PRINT_RE.C
   Autor: Gerhard Krucker
   Datum: 15.2.1995
   Sprache: MS Visual-C++ V1.5
*/

#include <stdio.h>

void print(char *);          /* Funktionsprototyp fuer print, da erst spaeter definiert */

main()
{ char msg[]="Rekursion\n";    /* Auszubender String */

  print(msg);                 /* Rekursive Ausgabefunktion */
  return 0;
}

void print(char *a)
{ if (*a == '\0')             /* Ende des Strings erreicht? (Abbruchkriterium) */
  return;
  else
  { putchar(*a);              /* Aktuelles Zeichen ausgeben */
    print(++a);              /* Verfahren für naechstes Zeichen wiederholen */
  }
}

```

### 9.3 Arrays an Funktionen übergeben

Ganze Arrays werden grundsätzlich über Zeiger an Funktionen übergeben. Es wäre unwirtschaftlich ein ganzes Array in Form einer Kopie aller Elemente an die Funktion zu übergeben.

Über die Zeigerreferenz kann dann in der aufgerufenen Einheit (der Funktion) in normaler Weise auf die einzelnen Elemente zugegriffen werden. Dabei wird das Array im formalen Parameter als "offenes Array", also mit einer unbekannt Anzahl Elementen definiert.

Beispiel:

Definition eines initialisierten Arrays a mit den Werten 1..10. Aufruf der Funktion print mit dem aktuellen Parameter des Zeigers auf a. In der Funktion werden dann die einzelnen Elemente des Arrays ausgegeben:

```

/* Uebergabe eines Arrays an eine Funktion.                                     File: ARR_PAR.C
   Autor: Gerhard Krucker
   Datum: 15.2.1995
   Sprache: MS Visual-C++ V1.5
*/

#include <stdio.h>

/* Ausgeben eines Arrays mit int-Daten auf den Bildschirm.
   Parameter: a = Zeiger auf ein (offenes) Array mit int-Daten
              n = Anzahl der auszugebenden Elemente */
void print(int a[], int n)
{ int i;

  for (i = 0; i < n; i++) printf("%d ",a[i]);
}

main()
{ int daten[]={1,3,4,7,10,0,1,3,2,-2,8};    /* Array mit Daten, welche auszugeben sind
*/
  int anzahl;

  anzahl = sizeof(daten) / sizeof(int);    /* Anzahl Elemente im Array bestimmen */
  print(daten,anzahl);
  return 0;
}

```

In diesem Beispiel wird ein eindimensionales Array (`daten`) an die Funktion `print` übergeben. Die Parameterübergabe erfolgt über den Zeiger auf das Array. Der formale Parameter muss also vom Typ `int []` sein. Der aktuelle Parameter wird bei Arrayübergaben immer mit dem Namen selbst spezifiziert. Wir wissen, dass der Arrayname selbst (ohne Klammern), ein Zeiger auf das Array verkörpert.

Bemerkung: Alternativ hätte man den formalen Parameter auch als Typ `int *` definieren können. Der Zugriff innerhalb der Funktion sowie den Aufruf hätte man genau gleich spezifiziert.

Wie sähe das Beispiel der Zahlenausgabe nun aus, wenn wir ein n-dimensionales Array ausgeben möchten? Das Vorgehen ist analog, nur das wir hier bei einem n-dimensionalen Array n-1 Dimensionen fixieren müssen. Fixieren heisst, wir müssen n-1 Dimensionen ihren Grössen als Konstanten angeben, damit in der Funktion die Indizierung korrekt berechnet werden kann. Der äusserste Index darf offen bleiben (muss aber nicht).

Für ein 4x3 Array mit `int`'s heisst dies 3 Spalten und 4 Zeilen. Es wird definiert als `int daten[4][3]`. Demzufolge muss als formaler Parameter in der Funktion etwas in der Art `int arr[][3]` stehen. Wir haben hier die Anzahl Spalten fixiert (Vorschrift: n-1 Dimensionen sind zu fixieren). Die Anzahl Zeilen darf offen bleiben.

Konkret sieht dies so aus:

```
/* Uebergabe eines zweidimensionalen Arrays an eine Funktion.      File: ArrParameter2.C
   Autor: Gerhard Krucker
   Datum: 15.2.1995, 18.10.2003
   Sprache: Intel C++ V7.1 (Win32 Console Application)
*/

#include <stdio.h>

/* Ausgeben eines Arrays mit int-Daten auf den Bildschirm.
   Parameter: a = Zeiger auf ein zweidimensionales Array mit int-Daten
              n = Anzahl der Elemente Spalten
              m = Anzahl der Elemente Zeilen
*/
void print(int a[][3],int n,int m)
{ int i,j;

  for (i = 0; i < n; i++)
    { for (j = 0; j < m; j++)
      printf("%d\t",a[i][j]);
      putchar('\n');
    }
}

main()
{ int daten[4][3]={1,3,4},{7,10,0},{1,3,2},{-2,8,9}}; /* Array mit Daten, welche
                                                         auszugeben sind */
  print(daten,3,4);

  return 0;
}
```

## 9.4 Strings an Funktionen übergeben

Genau gleich wie bei den Arrays, werden Strings an Funktionen übergeben. Strings sind ja nur spezielle eindimensionale Arrays. Sie sind vom Typ `char` und mit dem Nullzeichen terminiert.

Bei 'normalen Arrays' erfolgt die Definition des formalen Parameters meist mit Typ in eckigen Klammern. Bei String hingegen werden meist im formalen Parameter als `char *` definiert. Einerseits ist dies aus Konventionsgründen so. Andererseits orientiert sich die Definition an der Art des späteren Zugriffes auf die Daten:

Wird das Datenelement als Arrayelement mit Indizierung angesprochen, so wählt man als Parameter `char []`. Wird hingegen das Element über Dereferenzierung mit `*` angesprochen, so wählt man `char *`. Gleichwohl beide Arten sind absolut identisch wie auch die beiden nachfolgenden Funktionsprototypen:

```
void print(char *);           void print(char []);
```

Und als Gesamtbeispiel:

```
/* Uebergabe eines Strings mit char * als formaler Parameter.      File: StrPar1.C
   Autor: Gerhard Krucker
   Datum: 15.2.1995, 18.10.2003
   Sprache: Intel C++ V7.1 (Win32 Console Application)
*/

#include <stdio.h>

/* Ausgeben eines Strings der über Parameter uebergeben wird*/
void print_str(char *p)
{ while (*p != '\0')
  putchar(*p++);
}

main()
{ char message[]="Hallo\n";           /* Array mit der Meldung, welche auszugeben ist*/

  print_str(message);                 /* message ausgeben */
  return 0;
}
```

## 9.5 Wertrückgabe über Referenzparameter

Wir wissen, dass über Parameter selber keine Wertrückgaben aus einer Funktion möglich sind. Wird jedoch ein Zeiger auf eine Variable übergeben, so kann in der Funktion über diese Zeigerreferenz ein Zugriff erfolgen. Wir haben somit also ein Mittel in der Hand um in einer Funktion mehr als einen Wert zu retournieren.

Wollen wir, dass in einer Funktion auf einen "aktuellen" Parameter schreibend zugegriffen werden kann, so übergeben wir nicht den Parameter mit dem Wert selbst, sondern einen Zeiger auf den Parameter.

Beispiel:

```
void f(int *x)
{
    *x=3;
}

main()
{
    int y=1;

    printf("y vor dem Aufruf von f: %d\n",y);
    f(&y);
    printf("y nach dem Aufruf von f: %d\n",y);

    return 0;
}
```

Ergibt die Ausgabe:

```
y vor dem Aufruf von f: 1
y nach dem Aufruf von f: 3
```

## 9.6 Zusammenfassung über das Thema Funktionen

- **Grundsätzliches**

C-Programme umfassen eine bis mehrere Funktionen. Die Hauptfunktion bei der das Programm beginnt, heisst immer `main()`.

Jede Funktion hat einen Namen. Ihm folgen ein Klammernpaar für Parameter und nachher der Funktionskörper, der in `{}`-Klammern eingebettet wird. Gibt die Funktion einen anderen Typ als `int` zurück, muss der Typ vor dem Funktionsnamen spezifiziert werden.

Die Klammern für Parameter müssen immer angegeben werden, auch wenn die Funktion keine Parameter hat. Sowohl bei der Definition, wie beim Aufruf.

Wird der Name einer Funktion nicht richtig geschrieben, wird das Programm zwar fehlerfrei kompiliert, beim Linken wird jedoch eine Fehlermeldung erzeugt die sagt, dass die Funktion unbekannt sei.

Variablen innerhalb des Funktionskörpers sind lokale Variablen. Sie gelten nur innerhalb der Funktion.

Um globale Variablen zu definieren, müssen diese Variablen ausserhalb der Funktion definiert werden. Sie sind dann aus jeder Funktion zugreifbar. Globale Variablen sollten aus der Sicht der strukturierten Programmierung soweit möglich vermieden werden.

Wenn selbst definierte Funktionen denselben Namen haben wie die Bibliotheksfunktionen, so hat man keinen Zugriff mehr auf die ursprüngliche Standardfunktion. Die eigene Definition hat

also immer Vorrang.

- **Argumente**

Wenn eine Funktion Argumente hat, muss jedes einzelne Argument mit Typ und Namen in der Parameterklammer angegeben werden.

Die Datentypen der formalen und aktuellen Parameter müssen übereinstimmen. In manchen Fällen erkennt der Compiler Typenkonflikte nicht, was zur Programmlaufzeit zu merkwürdigen Fehlern führen kann.

- **return-Statement**

Return beendet die Funktion und gibt Wert nach `return` als Funktionswert retour. Alle nach `return` folgenden Statements werden ignoriert.

Wird der Funktionswert im Programm nicht übernommen, so führt dies zu keinem Fehler. Der Wert wird einfach ignoriert.

Der Standarddatentyp für den Funktionswert ist `int`.

Obwohl eine Funktion prinzipiell mehrere `return`-Statements haben kann, ist es eine gute Gewohnheit die Funktion nur an einer Stelle zu verlassen.

- **void-Statement**

Wird eine Funktion als vom Type `void` definiert heisst das, dass die Funktion keinen Wert zurückgibt. `void`-Funktionen brauchen demzufolge kein `return`-Statement.

- **Strings Funktionen übergeben**

Ein String-Argument wird als formaler Parameter als ein Zeiger auf den Grundtyp `char` definiert. Verbreitet ist auch die Notation eines Arrays offener Grösse (Bsp. `char string[]`).

- **Arrays Funktionen übergeben**

Ganze Arrays werden prinzipiell über Zeiger übergeben. Beim Aufruf (aktuellen Parameter) erfolgt dies durch einfache Angabe des Namens. Der formale Parameter ist dann ein Zeiger von der Dimension des Arrays (Bsp. 2x2 Array von `int`: `int [][]`). Der Zugriff in der Funktion erfolgt durch normale Indizierung.

- **Call by Value**

Dieses Prinzip sagt, dass das Argument als Kopie an die aufgerufene Funktion übergeben wird. Eine Veränderung des aktuellen Parameters ist somit in der Funktion nie möglich.

- **Rekursion**

Eine Funktion, die sich selber aufruft, bis ein Abbruchkriterium erfüllt ist, heisst rekursive Funktion. Rekursive Funktionen eignen sich vor allem für rekursiv definierte Verfahren. Rekursive Funktionen sind zwar elegant zu programmieren, aber meist nicht besonders lauffzeiteffizient. Im Fehlerfall sind sie schwierig auszutesten. Wenn möglich ist eine iterative Lösung vorzuziehen.

## 10 Typenkonversion

Wie bereits gesehen, gelten für alle Datentypen bestimmte Regeln bei Zuweisungen und anderen Operationen. Grundsätzlich darf nur Gleiches mit Gleichem verarbeitet werden. C kennt jedoch Konversionsfunktionen die dazu dienen verschiedene Systemdatentypen ineinander zu überführen.

Dabei wird unterschieden zwischen:

Implizite Konversionen: Die Konversion wird von C automatisch vorgenommen. Dies geschieht bei **zuweisungskompatiblen Datentypen**.

Explizite Konversion: Sie heissen im Fachausdruck **Cast** und sind eine erzwungene Typenkonversion.

### 10.1 Implizite Konversionen

Sie erfolgen automatisch ohne Zutun des Programmierers. Eine solche implizite Konversion geschieht beispielsweise beim Verrechnen von `int`'s und `float`'s:

```
int a=3;
float f=1.23f;

f = 3 * 3;
```

Hier wird also eine implizite Konversion vom Resultat (der `int` 9) nach `float` vorgenommen, ohne dass man speziell etwas formulieren müsste.

Anfangs erscheinen die Konversionsvorschriften etwas diffus, da syntaktisch korrekte Konstrukte in einer Rechnung überhaupt nicht das erwartete Resultat liefern. Oftmals ist ein Konversionsproblem daran Schuld.

Implizite Konversionen erfolgen automatisch für zuweisungskompatible Datentypen:

```
char -> short -> long -> float -> double -> long double
```

Eine Umwandlung in den Datentyp mit dem grösseren Wertebereich erfolgt ohne Meldung automatisch, wenn 'verlustfrei' konvertiert werden kann. Unter 'verlustfrei' ist die Genauigkeit der Mantisse der zu konvertierenden Zahl gemeint. So kann beispielsweise die Zahl 3 als Ganzzahl nicht verlustfrei in eine Gleitkommazahl gewandelt werden, weil die interne Zahlendarstellung (in einem codierten Binärformat mit Mantisse und Exponent) zur Zahlenbasis 2 keine abbrechende Mantisse aufweist. Bei der Wandlung wird sie nach einer bestimmten Anzahl Stellen abgeschnitten. Der Compiler wird in diesem Fall eine Warnung ausgeben, obwohl vom numerischen Wert her gesehen, kein falsches Resultat erzeugt wird.

Erfolgt umgekehrt eine Wandlung in einen Datentyp mit kleinerem Wertebereich, erfolgt auch eine ev. mit Verlusten behaftete Konversion (mit entsprechender Warnung des Compilers). Dabei ist aber zu berücksichtigen:

Wird einer `char` eine `int` zugewiesen so werden die 8 höherwertigen Bits abgeschnitten. Dasselbe gilt für eine Zuweisung von `long` nach `short`.

**Warnings bei Zuweisungen deuten auf ein existentes Konversionsproblem hin**, das unbedingt behoben werden sollte, auch wenn das Programm scheinbar läuft.

Nachfolgend eine systematische Zusammenstellung der Regeln:

1. enum-Konstanten werden immer zu `int` konvertiert.
2. Wird eine Gleitkommazahl zu einer ganzen Zahl konvertiert, wird der Nachkommaanteil immer abgeschnitten (truncated). Falls der ganzzahlige Anteil in der Zielvariablen keinen Platz hat, ist das Ergebnis undefiniert. Gleitkommazahlen können in Typen mit kleinerem Wertebereich (z.B. `double` -> `float`) konvertiert werden sofern das Ergebnis in der Zielvariablen Platz hat sonst ist das Verhalten nicht definiert.
3. `char`, `short int`, egal ob `signed` oder `unsigned`, können zu `int` konvertiert werden. Kann eine `int` den Wert nicht aufnehmen, wird sie automatisch zu `unsigned int` konvertiert.
4. Ganze Zahlen können zu Gleitkommazahlen gewandelt werden. Es ist aber möglich, dass der Wert intern nicht exakt dargestellt werden kann.
5. Wird eine `int` einer `char` zugewiesen so werden die höherwertigen Bits abgeschnitten. Dasselbe für `long` nach `short`.

Generell gilt, dass automatische Konversionen nur dann erfolgen, wenn der Wert des Objektes erhalten bleibt. Unsinnige Konversionen werden unterdrückt und als Fehler gemeldet.

So ist z.B. unzulässig:

```
arr[2.0]=.....;
```

oder

```
double pi=3.14159;  
.....  
reihe[pi]=.....;
```

Hingegen ist erlaubt:

```
int n;  
float f;  
.....  
...= i+f;
```

Es ist zu empfehlen, dass verlustbehaftete Konversionen nur über explizite Konversionen vorzunehmen sind (Casts).

## 10.2 Arithmetische Konversionen:

Im Zusammenhang mit binären arithmetischen Operationen (+, -, \*, /, etc.) nimmt C automatisch die folgenden Konversionen vor. Entsteht aber aufgrund der Regeln ein Konflikt, so findet diejenige Regel Anwendung, die früher in der Liste steht:

1. Ist ein Operand `long double`, so wird der andere zu `long double` konvertiert.
2. Ist ein Operand `double`, so wird der andere zu `double` konvertiert.
3. Ist ein Operand `float`, so wird der andere zu `float` konvertiert.
4. Ist ein Operand `long int`, so wird der andere zu `long int` konvertiert.
5. Ist ein Operand `long int` und der andere `unsigned int`, so wird Letzterer zu `long int` konvertiert, falls dieser alle Werte des `unsigned int` halten kann. Sonst erfolgt eine Konversion zu `unsigned long int`.
6. Ist ein Operand `unsigned int`, so wird der andere zu `unsigned int` konvertiert.
7. Trifft keine der Regeln zu, so werden beide Operanden als `int` behandelt.

## 10.3 Explizite Konversionen (Casts)

Mit dem Cast-Operator können explizite Typenkonversionen erzwungen werden. Casts sind für alle Systemdatentypen und Zeiger anwendbar. Der Cast-Operator hat die Form:

(Datentyp) Ausdruck

Hier wird der Wert des Ausdrucks in den Datentyp der in Klammern angegeben wird, gewandelt. Bei der Wandlung werden niemals irgendwelche Variablenwerte im Ausdruck selbst verändert. Es ist in diesem Sinne eine Anweisung an den Compiler wie er den Wert aufzufassen hat.

So ergeben folgende Cast die Werte:

```
(int) 3.14159           ergibt 3
(float)2                ergibt 2f
(char)1000              ergibt -24
(unsigned char)1000     ergibt 232
(unsigned short)-2      ergibt 65534
```

oder:

```
typedef {rot, gruen, blau, weiss, gelb} SPEKTRUM;
typedef {montag, dienstag, mittwoch, donnerstag} TAG;
SPEKTRUM farbe;
TAG heute;

farbe = (SPEKTRUM) mittwoch;
heute = (TAG) 2;
```

Hingegen sind folgende Ausdrücke nicht erlaubt:

```
farbe = mittwoch;
heute = 2;
```

Cast mit Zeiger werden ebenfalls häufig verwendet. Praktisch alle Bibliotheksfunktionen der Sprache C, die einen Zeiger als Resultat liefern, liefern einen generischen Zeiger vom Type `void *` retour. `void *` sind neu in ANSI-C eingeführt und sind spezielle Zeiger, die auf alle Typen zeigen dürfen. Da 'normale' Zeiger in C eine sehr starke Typenbindung zum Datentyp haben auf den sie zeigen, ist bei Zuweisungen von solchen `void *` Zeigervariablen ein Casting notwendig.

Beispiel:

```

/* Array mit 5000 int Elementen zu Programmlaufzeit mit dynamischer Speicherplatzan-
forderung malloc() erzeugen und alle Elemente mit zufälligen Werten zwischen 11.0 und
12.0 beschreiben. Danach den Mittelwert bestimmen und ausgeben.

Autor: Gerhard Krucker
Datum: 13.12.1994, 18.10.2003
Sprache: Intel C++ V7.1
File: ArrMalloc.c
*/
#include <stdio.h>
#include <stdlib.h>          /* Fuer malloc() und rand() */
#define ARR_SIZE 5000      /* Array soll 5'000 Elemente haben */

int main()
{ float *arr;              /* Zeiger auf den Beginn des Arrays */
  long double summe;      /* Summe fuer die Mittelwertrechnung */
  float mw;               /* Mittelwert */
  int i;

  /* Speicherplatz fuer das Array mit 5000 Elementen bestellen. Dabei wird angegeben
  wieviele Bytes Speicherplatz gebraucht wird. malloc() liefert dann einen zeiger
  (void *) auf den belegbaren Bereich. Falls kein Speicher erhaeltlich war
  (d.h. Fehler), wird NULL retourniert.
  */
  arr = (float *) malloc(ARR_SIZE * sizeof(float));

  if (arr == NULL)        /* malloc() konnte den Speicher nicht bereitstellen,
                          Programm beenden */
    { printf("Fehler: Zuwenig Speicher!\n");
      exit (1);
    }

  /* Ganzes Array mit Zufallszahlen fuellen */
  for (i=0; i < ARR_SIZE; i++)
    { float z; /* aktuelle Zufallszahl */
      z = (float)rand()/(float)RAND_MAX; /* liefert eine Zufallszahl in [0..1] */
      z = z + 11; /* nach [11.12] schieben */
      arr[i] = z; /* und Element einschreiben */
    }

  // Array anzeigen (Falls man Zeit hat, sonst auskommentieren)
  for (i=0; i < ARR_SIZE; i++)
    printf("Wert %d: %f\n",i,arr[i]);

  /* Mittelwert berechnen */
  summe =0L;
  for (i=0; i < ARR_SIZE; i++)
    summe = summe + arr[i];
  mw=(float)(summe / ARR_SIZE);
  printf("Mittelwert: %f\n",mw);

  /* Allozierten Speicher wieder zurueckgeben */
  free(arr);

  return 0;
}

```

# 11 Der Preprozessor

Der C-Preprozessor ist eine Art 'Precompiler', der vor dem eigentlichen Kompilervorgang abläuft. Er wertet dabei alle Anweisungen die mit # beginnen aus. Solche Anweisungen sind Preprozessoranweisungen. Sie sind ausschliesslich für den Preprozessor bestimmt und demnach auch nicht Bestandteil der Sprache (Syntax) C.

Preprozessoranweisungen erzeugen selber keinen direkten Maschinencode, sondern dienen zur Steuerung des Kompilationsvorganges und zur Definition von symbolischen Konstanten.

Präziser gesagt, erfüllt der Preprozessor folgende Aufgaben:

1. Eliminieren aller Kommentare.
2. Expandieren von Makros. (Makros sind Anweisungen, die eine bestimmte Menge Text erzeugen)
3. Einbinden von anderen Sourcefiles.
4. Bedingte Kompilation, d.h. bestimmte Programmteile werden aufgrund einer Bedingung kompiliert oder nicht.

Der Preprozessor wertet die #-Anweisungen aus und erzeugt intern ein Zwischenfile, das dann vom eigentlichen C-Compiler weiterverarbeitet wird.

Summarisch kennt der Preprozessor folgende Anweisungen:

## Bedingte Kompilation

```
#if  
#ifdef  
#ifndef  
#elif  
#else  
#endif
```

## Einbinden und Definieren

```
#include  
#define  
#undef
```

## Sonstiges

```
#pragma  
#line  
#error
```

Da der Preprozessor total vom eigentlichen Compiler unabhängig ist, gelten für den Preprozessor teilweise andere Syntaxregeln.

Nachfolgend werden die wichtigen Preprozessoranweisungen kurz vorgestellt und mit einigen Beispielen illustriert.

## 11.1 Definieren von Symbolen und Makros (#define)

Die Anweisung `#define` definiert ein Makro. Makros sind spezielle Symbole, die eine bestimmte Menge Text erzeugen. Beim Preprozessordurchlauf werden dann diese Makros aufgelöst (expandiert). D. h. anstelle der Makrosymbole wird der konkrete Wert eingesetzt.

Die allgemeine Syntax für ein einfaches `#define` lautet:

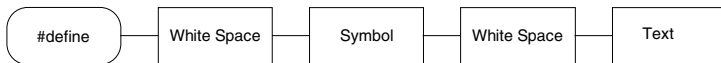


Bild 11-1: Syntaxdiagramm für `#define` Preprozessor Anweisung.

Im Syntaxdiagramm verkörpert `Symbol` ein beliebiger Name für ein Makrosymbol. `White Space` ist ein Separatorzeichen (Leerschlag, Tabulator). `Text` verkörpert dasjenige, das für `Symbol` eingesetzt werden soll.

**Wichtig** ist immer zu bedenken, dass hier eine **rein textuelle Substitution** erfolgt!

Ein einfaches Beispiel: Es soll ein Symbol `PI` definiert werden, das den Wert `3.14159` verkörpert:

Im Sourcecode schreiben wir:

```
#define PI 3.14159
```

Wird im Sourcecode vom Preprozessor beispielsweise die Anweisung

```
cos(2 * PI * k)
```

gelesen, so wird dies zu

```
cos (2 * 3.14159 * k)
```

expandiert.

Wird ein Symbol mit `#define` definiert, so ist dieses Symbol bis zum Ende des Sourcefiles definiert. Makros können nicht direkt umdefiniert werden, d. h. erhält ein Symbol einen Wert mit `#define`, so kann dasselbe Symbol später nicht ohne vorherige Massnahmen umdefiniert werden.

Wird eine solche Umdefinition gebraucht, so muss die vorgängige Definition gelöscht werden. Dies erfolgt mit der `#undef`-Anweisung.

Beispiel:

```
#define BLOCK_SIZE 512
...
...
buffer=BLOCK_SIZE * length;
...
#undef BLOCK_SIZE
#define BLOCK_SIZE 2048
...
...
buffer=BLOCK_SIZE * length;
...
```

Mit `#define` können auch Makros mit Argumenten definiert werden. Die Argumente werden in Klammern direkt nach Symbolnamen angegeben. Bei der Auflösung des Makros werden die Argumente dann in die Expansion eingesetzt.

Allgemeine Form:



Bild 11-2: Syntaxdiagramm für #define Preprozessor Makro-Definition.

Beispiel:

Definition eines Makros QUADRAT (x) das Produkt von x\*x einsetzt:

```
#define QUADRAT(x) ((x)*(x))
```

Wird in einem Programm vom Preprozessor

```
k = 3;  
i = QUADRAT(k);  
gelesen, so wird dies zu
```

```
k = 3;  
i = (k)*(k);
```

expandiert.

Beachten Sie, dass das Argument x bei der Makrodefinition in Klammern steht. Dies ist in der Regel notwendig. Wäre dies nicht so, so würde folgende Makroexpansion:

```
#define QUADRAT(x) (x*x)  
...  
k = 3;  
i = QUADRAT(k+2);
```

ein unkorrektes Resultat liefern.

Der Preprozessor bietet noch eine Reihe weiterer Optionen an: So können Argumente in String verwandelt werden und Symbole miteinander verschmolzen (verbunden) werden. Näheres hierzu in den Referenzhandbüchern zu den jeweiligen Compilern.

Hat eine Definition aus irgendeinem Grund nicht auf einer Zeile Platz, so kann mit dem Backslash die Definition auf der nächsten Zeile fortgesetzt werden. Das dabei entstehende Zeilenvorschubzeichen wird vom Preprozessor automatisch entfernt:

```
#define LANGESMAKRO "Dies ist ein langes \  
Makro"
```

## 11.2 Bedingte Kompilation

Hierbei wird der Kompilervorgang aufgrund einer Bedingung gesteuert, d.h. es können Programmteile aufgrund einer Bedingung kompiliert werden oder nicht.

Beispiel:

```
#ifdef MSDOS
    #define BLOCK_SIZE 512
#else
    #define BLOCKSIZE 2048
#endif
```

Hierbei wird je nachdem ob das Symbol MSDOS definiert ist (unabhängig ob es einen Wert hat oder nicht) das Symbol BLOCK\_SIZE mit 512 oder 2048 definiert.

Eine Anwendung ist das Einbinden von so genanntem Debug-Code in das Programm. Hierbei wird zusätzlicher Code in das Programm eingebracht, der während des Programmlaufes Zwischenresultate ausgibt. In der Testphase wird dazu ein Symbol DEBUG definiert und der Debug-Code mit bedingter Kompilation der Art:

```
#ifdef DEBUG
    printf("Zwischenresultat: %d ",x);
#endif
```

Ebenso kann mit #if auch aufgrund einer logischen Bedingung die Kompilation gesteuert werden:

```
#if VERSION == 1
    ...
#endif
#if VERSION == 2
    ...
#endif
```

## 11.3 Einbinden von Sourcefiles (#include)

#include bindet genau an der Stelle das entsprechende Sourcefile ein. Das heisst, der Compiler fährt an dieser Stelle mit dem Kompilieren des include-Files weiter und wenn dieses abgearbeitet wurde, wird normal weitergefahren.

Das include-File wird also in einer Art Subroutine kompiliert.

Include Files können ihrerseits wieder #include's beinhalten. Eine natürliche Grenze ist dabei in der Anzahl gleichzeitiger offener Files im jeweiligen System gesetzt.

## 11.4 Sonstiges

`#pragma` setzt Compileroptionen. Mit dieser Anweisung können Speichermodelle, FPU-Unterstützung und vieles andere beeinflusst werden. `#pragma`-Anweisungen sind durch die komfortablen Entwicklungsumgebungen mit den Einstellmöglichkeiten für Projektoptionen etwas in den Hintergrund geraten. Jedoch ist es so, dass mit `#pragma` praktisch alles compiler- und Linkerrelevante eingestellt werden kann.

Wichtig: `#pragma` Anweisungen sind für jeden Compiler und jede Version unterschiedlich!

`#line` bindet Zeilennummern in das Programm ein. Im Fehlerfall erscheint dann eine Fehlermeldung mit Angabe der Zeilennummer. Beim Kompilieren wird ein File erzeugt, das Informationen für diese Zeilennummern liefert. In der Regel wird diese `#line`-Anweisung eher selten verwendet. Zusatzprogramme, die mit dem Compiler zusammenarbeiten, nutzen jedoch oft diese Option.

`#error` produziert eine Meldung während des Kompilationsdurchlaufes. Die Meldung wird dabei auf den Standard Fehlerkanal (`stderr`, in der Regel der Bildschirm) ausgegeben. Diese Anweisung dient dazu, um spezielle Fehlermeldungen während des Kompilierens auszugeben, die nicht der Preprozessor selbst erzeugt oder erzeugen kann (Bsp. Makrosymbole ungültig o.ä.).

`#error` *Fehlermeldung*

erzeugt folgende Ausgabe:

```
Error: Filename Zeile#:Error Direktive: Fehlermeldung
```

Beispiel:

```
#define VERSION 3
#if VERSION != 1 && VERSION != 2
#error VERSION muss entweder 1 oder 2 sein
#endif
```

Erzeugt die Ausgabe:

```
Initializing...
Compiling...
d:\e2000\Informatik\error.c
d:\e2000\Informatik\error.c(3) : error C2189: #error : VERSION muss entweder 1 oder 2 sein

CL returned error code 2.
ERROR.C - 1 error(s), 0 warning(s)
```

## 12 File I/O

Datenein- und Ausgaben zu C-Programmen erfolgen über Kanäle, die durch Files verkörpert werden. In diesen Kanälen werden die Zeichen sequenziell ein- oder ausgegeben. Files stellen also sequenzielle Datenströme dar. Grundsätzlich erfolgen Datentransfers von und zu Programmen nur über Files. C++ verfolgt mit Stream-I/O ein gänzlich anderes I/O-Konzept.

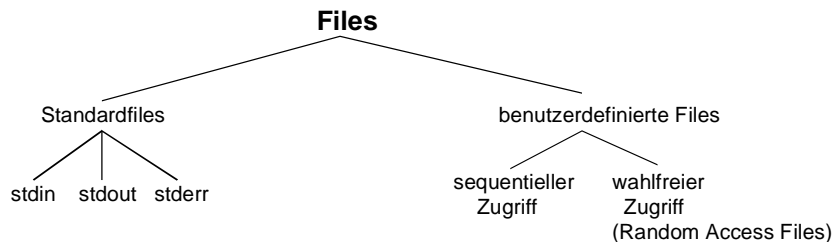


Bild 12-1: Filetypen in C.

Standardmässig kennt C folgende vordefinierte Files:

`stdin` (Standard Input) verkörpert in der Regel die Tastatur  
`stdout` (Standard Output) verkörpert in der Regel den Bildschirm  
`stderr` (Standard Error) geht auf den Bildschirm. (Kann nicht umgeleitet werden.)

Diese Files stehen beim Programmbeginn zur Verfügung ohne, dass irgendwelche zusätzliche Initialisierungen und Definitionen notwendig sind.

Benutzerdefinierte Files werden für den Zugriff auf externen Geräten benötigt. Darunter fallen alle Diskspeicher. Je nach Plattform können auch andere Periferiegeräte angesprochen werden

Der Zugriff auf Files erfolgt ausschliesslich über Funktionen. Sie werden in Funktionen zum Lesen, Schreiben und Verwalten gruppiert. Viele der Zugriffsfunktionen sind sehr speziell. So bietet `putch()` eine Zeichenausgabe an, die prinzipiell auch mit `printf()` möglich wäre, jedoch ist `putch()` wesentlich schneller.

### 12.1 Zugriffsfunktionen

C bietet ein umfangreiches Sortiment an Filefunktionen an. Von der internen Wirkungsweise wird in C zwischen fileorientierten Funktionen und blockorientierten Funktionen unterschieden. Fileorientierte Funktionen sind Standardfunktionen für Files. Sie ermöglichen bequemen Zugriff auf alle Files. Blockorientierte Funktionen arbeiten auf systemnaher Ebene und sind in ihrer Funktion maschinenorientiert zu sehen. Sie erlauben einen effizienteren Datendurchsatz, indem Daten Blockweise gelesen oder geschrieben werden.

Der Regelfall für unsere Anwendungen sind normale, fileorientierte Funktionen.

Zugriffsfunktionen auf `stdin`, `stdout`: (ohne Parameter)

<code>getchar</code>	Zeichen von <code>stdin</code> lesen
<code>gets</code>	String von <code>stdin</code> lesen
<code>putchar</code>	Zeichen auf <code>stdout</code> ausgeben
<code>puts</code>	String auf <code>stdout</code> ausgeben
<code>scanf</code>	Daten formatiert von <code>stdin</code> lesen
<code>printf</code>	Daten formatiert auf <code>stdout</code> ausgeben

Die wichtigsten Zugriffsfunktionen auf allgemeine Files: (ohne Parameter)

<code>getc</code>	Ein Zeichen vom File lesen (Makro)
<code>putw</code>	Ein Integerwort in das File schreiben (binär)
<code>fgetc</code>	Ein Zeichen vom File lesen (Funktion)
<code>fopen</code>	File zum Lesen/ Schreiben öffnen
<code>fgets</code>	String aus einem File lesen
<code>fputs</code>	String in File schreiben
<code>fprintf</code>	Daten formatiert in File schreiben
<code>fscanf</code>	Daten formatiert aus File lesen
<code>fseek</code>	Datenzeiger auf neue Position setzen
<code>ftell</code>	Position des Datenzeigers lesen
<code>feof</code>	Prüft ob Ende des Files erreicht ist
<code>Fclose</code>	File schliessen
<code>Fgetpos</code>	Position des Filezeigers ermitteln
<code>Fflush</code>	Ausgabepuffer leeren
<code>ferror</code>	Prüft Fehlerbedingungen
<code>fread</code>	Liest Datenblock aus dem File
<code>fwrite</code>	Schreibt Datenblock in das File

## 12.2 Benutzerdefinierte Files

Sie verkörpern Dateien auf peripheren Geräten (Harddisk, Floppy, CD-ROM, etc). Mit benutzerdefinierten Files können wir also in einem C-Programm selber auf eine Diskette schreiben oder ein bestehendes File belesen. Mit Files können also Daten dauerhaft gespeichert werden.

Der Zugriff erfolgt wie bei den Standardfiles über spezielle Bibliotheksfunktionen. Im Gegensatz zu anderen Programmiersprachen kennt C keinen Unterschied zwischen sequentiellen Files (Sequential Files ) und Files mit wahlfreiem Zugriff (Random Access Files). Es werden alle Files als Folge von Zeichen (Bytes) behandelt und die Verwaltung erfolgt intern über einen Zeiger auf eine Datenstruktur der alle filerelevanten Informationen enthält.

Der Vollständigkeit halber muss erwähnt werden, dass C ein ebenso grosses Sortiment an blockorientierten Filefunktionen kennt. Diese wirken auf einer systemnahen Ebene und transportieren die Daten mit einer grösseren Effizienz als die sog. Stream-Funktionen. Jedoch bieten sie keinen so grossen Komfort wie die Stream- Funktionen, bezüglich formatierte Ausgabe etc., da alle Transfers als Blöcke von Bytes erfolgen.

## 12.3 Praktische Arbeit mit benutzerdefinierten Files:

Alle benutzerdefinierten Files werden über eine spezielle Datenstruktur verwaltet. Jedes File hat einen Zeiger auf eine solche Datenstruktur. In ihr stehen Filegrösse, Datum, Zugriffsrechte, etc.. Umgangssprachlich werden diese Zeiger als **File Pointer** bezeichnet. (Technisch gesehen zeigt er auf den File Control Block)

### 12.3.1 File Pointer

Jedes File muss also seinen eigenen File Pointer besitzen. Über ihn werden alle Fileaktivitäten verwaltet. Der File Pointer wird als Zeiger auf Datentyp FILE definiert:

```
#include <stdio.h>

FILE *fp          /* fp ist ein File Pointer */
```

Die Definition (Konstruktionsmuster) für den Typ FILE ist in der Headerdatei `stdio.h` aufgeführt. Somit ist bei der Arbeit mit Files `stdio.h` zwingend einzubinden. Bevor mit einem File gearbeitet (gelesen oder geschrieben) werden kann, muss das File geöffnet werden.

### 12.3.2 Öffnen eines Files

Beim Öffnen eines Files wird `fp` mit dem entsprechenden Zeigerwert versehen. Das Öffnen erfolgt grundsätzlich mit der Funktion `fopen()`. Sie erhält als Parameter den Filenamen und den Zugriffsmodus. Sie liefert dann als Resultat einen Zeiger auf den zum File gehörenden Kontrollblock zurück.

Beispiel:

```
#include <stdio.h>

....
FILE *fp;
....
fp = fopen("MYFILE.C", "r");
if (fp == NULL)
    fprintf(stderr, "Fehler: File konnte nicht geoeffnet werden\n");
else
    { /* Oeffnen war erfolgreich */
```

`fopen()` öffnet hier das File mit dem Namen `MYFILE.C`. Der Bezug ist das aktuelle Inhaltsverzeichnis. Im String für den Filenamen kann auch ein ganzer Pfad spezifiziert werden. Jedoch ist die C-Syntax zu berücksichtigen für die `\`-Symbole. So wird ein `fopen()` für ein File `README.TXT` im Verzeichnis `C:\DOS` konkret:

```
fopen("C:\\DOS\\README.TXT", "r");
```

Der Zugriffsmodus auf das File wird mit dem zweiten Argument, dem Modusstring, spezifiziert:

r	File wird für lesenden Zugriff geöffnet
w	Ein neues File wird für schreibenden Zugriff eröffnet. Ein bestehendes File mit gleichem Namen wird ohne Meldung überschrieben.
r+	Ein bestehendes File wird für lesenden und schreibenden Zugriff geöffnet.
w+	Ein neues File wird für Lesen und Schreiben eröffnet.
a	An ein bestehendes File werden die Daten angehängt (Append).
t	File wird im Textmodus geöffnet (standard)
b	File wird im Binärmodus geöffnet

Der Textmodus besagt, dass beim Einlesen eines `\n`-Zeichens dies zu 'einer Zweizeichensequenz `\r\n` expandiert wird und beim Ausschreiben umgekehrt. Diese 'Kompression' von Textfiles ist in UNIX-System üblich.

Der Binärmodus liest und schreibt die Daten 1:1, ohne irgendwelche Modifikationen.

Soll nun angegeben werden, dass ein File im Binärmodus geöffnet werden soll, so wird dies als Zusatz-

angabe im Modusstring spezifiziert:

```
fopen("C:\\DOS\\ATTRIB.EXE", "rb"); /* File ATTRIB:EXE binaer oeffnen */
```

Folgendes Minimalprogramm erzeugt ein (leeres) File mit dem Namen DEMO.DAT:

```
/* Leeres File mit dem Namen DEMO.DAT erzeugen */  
  
#include <stdio.h>  
  
main()  
{ FILE *fp;  
  
  fp = fopen("DEMO.DAT", "w");  
  if (fp == NULL)  
    printf("Fehler: File konnte nicht angelegt werden\n");  
  else  
    { printf("File erfolgreich angelegt\n");  
      fclose(fp);  
    }  
  
  return 0;  
}
```

Konnte das File nicht erfolgreich geöffnet werden, so liefert `fopen()` den Zeigerwert `NULL` zurück. `NULL` als Resultat einer Funktion besagt immer, dass irgendwas fehlgeschlagen ist. So wird mit `NULL` angezeigt, dass das File nicht geöffnet werden konnte. (`NULL` ist eine von C vordefinierte Konstante.)

Hier eine kleine Auswahl von Möglichkeiten, die ein Fehlschlagen von `fopen()` bewirken:

1. File mit diesem Namen existiert nicht in diesem Verzeichnis. (Auch Pfad kontrollieren). Regelfall.
2. Ein schreibgeschütztes File wurde zum Schreiben geöffnet.
3. Anzahl maximal offener Files wurde überschritten (systemabhängig).

Wird ein File geöffnet, so muss es nach Gebrauch wieder geschlossen werden. Dies erfolgt mit der Funktion `fclose()`.

Eine Begründung für die Notwendigkeit des Schliessens:

Files stellen Datenkanäle dar, die über das Betriebssystem zur Peripherie laufen. Wird nun ein File geöffnet, so wird ein solcher Kanal belegt. Diese Kanäle sind auf vielen Plattformen nur in begrenzter Zahl vorhanden. Nach Beendigung der Transfers soll das Programm den Kanal wieder freigeben, damit andere Programme und Prozesse diese Kanäle nutzen können.

C ist jedoch so freundlich, dass beim Beenden des Programmes implizit alle Files geschlossen werden. Es ist jedoch eine gute Gewohnheit im Sinne einer sauberen Programmentwicklung, immer alle benutzten Files nach Gebrauch zu schliessen.

**Wichtig:**

Beim schreibenden Zugriff stellt das Schliessen des Files ein Muss dar. `fclose()` bewirkt, dass das Inhaltsverzeichnis aktualisiert wird.

### 12.3.3 Daten auf Files schreiben

Wie bereits gezeigt, haben wir ein umfangreiches Sortiment von Funktionen um Daten auf ein File zu schreiben. Bevor Daten auf ein File geschrieben werden können, muss das File eröffnet worden sein. Dies erfolgt immer mit `fopen()`. Danach können mit den Funktionen

```
fprintf  
fputs  
fputc  
putw
```

Daten in das File geschrieben werden. Wir zeigen das Vorgehen an dem Beispiel:

```
/* Daten in File das File ALPHABET.DAT schreiben          (File: FILE1.C) */  
  
#include <stdio.h>  
  
main()  
{ FILE *fp;  
  int i;  
  
  fp = fopen("ALPHABET.DAT", "w");          /* File zum Schreiben eroeffnen */  
  if (fp == NULL)  
    printf("Fehler: File kann nicht angelegt werden\n");  
  else  
    { fprintf(fp, "Alphabetische Ausgabe aller Grossbuchstaben:\n");  
  
      /* Jedes Zeichen des Grossbuchstabensatzes in einer Schleife ausgeben */  
      for (i='A'; i <= 'Z'; i++)  
        putc(i, fp);  
  
      fclose(fp);          /* File nach Einschreiben aller Zeichen schliessen */  
    }  
  
  return 0;  
}
```

Wir erhalten nach dem Programmlauf im aktuellen Inhaltsverzeichnis ein neues File mit dem Inhalt:

```
Alphabetische Ausgabe aller Grossbuchstaben:  
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

### 12.3.4 Lesen von Daten aus Files

Bevor ein Lesezugriff erfolgen kann, muss auch hier das File zum Lesen geöffnet worden sein. Das Öffnen erfolgt mit `fopen()`, das einen Zeiger auf den File Control Block liefert. Ist dieser Zeiger ungleich `NULL`, so können Daten bis zum Ende des Files gelesen werden.

Das Ende des Files kann man sich als unsichtbare Marke EOF (End Of File) im Datenstrom vorstellen. Vom Beginn des Files werden die Daten streng sequenziell bis zu EOF gelesen, ähnlich einer Datenlesung von einem Band:

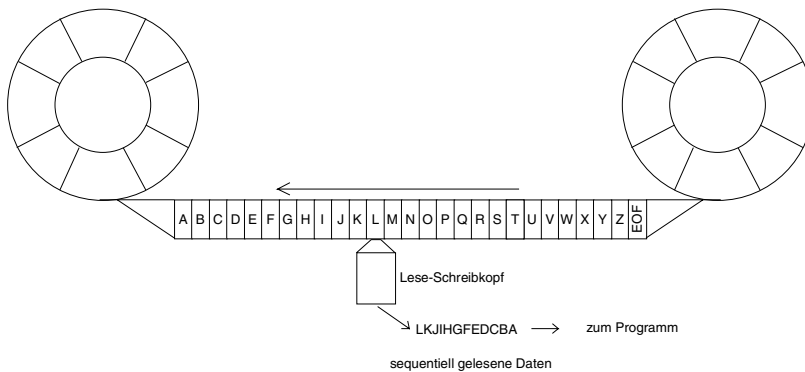


Bild 12-2: Prinzip des sequenziellen Zugriffs bei einem File, ähnlich dem Lesen vom Magnetband.

EOF wird bei Textfiles durch das Zeichen `\x1a` (Control-Z) dargestellt und beim Einlesen kann auf dieses Zeichen geprüft werden. EOF wird in C als `EOF` in Form einer vordefinierten Konstanten zur Verfügung gestellt (in `stdio.h`).

Als Beispiel lesen wir zeichenweise das vorher erzeugte Datenfile `ALPHABET.DAT` ein und geben die Zeichen auf dem Bildschirm aus:

```
/* Daten aus dem File ALPHABET.DAT lesen und anzeigen */
#include <stdio.h>

main()
{ FILE *fp;
  char c;

  fp = fopen("ALPHABET.DAT","r");          /* File zum Lesen eroeffnen */
  if (fp == NULL)
    printf("Fehler: File existiert nicht\n");
  else
    { while ((c=getc(fp)) != EOF)          /* Ein Zeichen in c einlesen und prüfen auf EOF
    */
      putchar(c);

      fclose(fp);                          /* File nach Einschreiben aller Zeichen schliessen */
    }

  return 0;
}
```

Ein Programmlauf erzeugt folgende Ausgabe:

```
Alphabetische Ausgabe aller Grossbuchstaben:
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

### 12.3.5 Lesen aus Binärfiles

Binärfiles haben kein EOF-Zeichen. Hier erfolgt die Prüfung durch Zählung der gelesenen Zeichen und Vergleich mit der Filegröße. Dazu verwendet man die Funktion `feof()`. Sie gibt als Resultat, ob das Fileende erreicht worden ist oder nicht.

Als Beispiel für das Belesen eines Files im Binärmodus ist ein Hexdump-Programm gezeigt. Es gibt den Inhalt eines Files als Hexbytes auf den Bildschirm aus.

```
/* Ein File binaer belesen und die Daten als Hexdump
   mit Adressen zeilenweise a 16 Bytes ausgeben. */

#include <stdio.h>

main()
{ FILE *fp;
  char fname[80];      /* String fuer den Filenamen */
  unsigned char b;     /* gelesenes Byte */
  unsigned long offset; /* Aktueller Offset im File */

  printf("Filename für Hexdump: ");
  scanf("%s", fname);

  fp = fopen(fname, "rb");      /* File zum Lesen eroeffnen */
  if (fp == NULL)
    printf("Fehler: File existiert nicht\n");
  else
    { /* File existiert: Nun Belesen und Daten ausgeben */
      offset = 0lu;
      while (!feof(fp)) /* Solange nicht EOF Daten lesen und ausgeben */
        { if (offset % 16 == 0) /* Offset am Beginn der Zeile ausgeben */
          printf("\n%06lu ", offset);
          b = getc(fp); /* 1 Byte lesen */
          printf("%02x ", b);
          offset++;
        }
      fclose(fp);
    }
  return 0;
}
```

Wird das Programm auf sich selbst angewandt, erhalten wir die Ausgabe:

```
Filename für Hexdump: file3.c

000000 2f 2a 20 45 69 6e 20 46 69 6c 65 20 62 69 6e 61
000016 65 72 20 62 65 6c 65 73 65 6e 20 75 6e 64 20 64
000032 69 65 20 44 61 74 65 6e 20 61 6c 73 20 48 65 78
000048 64 75 6d 70 0d 0a 20 20 20 6d 69 74 20 41 64 72
000064 65 73 73 65 6e 20 5a 65 69 6c 65 6e 77 65 69 73
000080 65 20 61 20 31 36 20 42 79 74 65 73 20 61 75 73
000096 67 65 62 65 6e 20 09 09 28 46 69 6c 65 3a 20 46
000112 49 4c 45 33 2e 43 29 20 2a 2f 0d 0a 0d 0a 23 69
000128 6e 63 6c 75 64 65 20 3c 73 74 64 69 6f 2e 68 3e
000144 0d 0a 0d 0a 6d 61 69 6e 28 29 0d 0a 7b 20 20 46
000160 49 4c 45 20 2a 66 70 3b 0d 0a 20 20 20 63 68 61
000176 72 20 66 6e 61 6d 65 5b 38 30 5d 3b 20 20 20 20
000192 20 20 20 2f 2a 20 53 74 72 69 6e 67 20 66 75 65
000208 72 20 64 65 6e 20 46 69 6c 65 6e 61 6d 65 6e 20
000224 2a 2f 0d 0a 20 20 20 75 6e 73 69 67 6e 65 64 20
000240 63 68 61 72 20 62 3b 20 20 20 20 20 2f 2a 20
000256 67 65 6c 65 73 65 6e 65 73 20 42 79 74 65 20 2a
000272 2f 0d 0a 20 20 20 75 6e 73 69 67 6e 65 64 20 6c
000288 6f 6e 67 20 6f 66 66 73 65 74 3b 20 20 2f 2a 20
000304 41 6b 74 75 65 6c 6c 65 72 20 4f 66 66 73 65 74
000320 20 69 6d 20 46 69 6c 65 20 2a 2f 0d 0a 20 20 20
000336 0d 0a 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
000352 20 20 20 20 0d 0a 20 20 20 70 72 69 6e 74 66 28
000368 22 46 69 6c 65 6e 61 6d 65 20 66 fc 72 20 48 65
000384 78 64 75 6d 70 3a 20 22 29 3b 0d 0a 20 20 20 73
```

## 13 Literaturverzeichnis

Folgende Literatur wurde referenziert:

- [KER83] Programmieren in C, Brian W. Kernighan/ Dennis M. Ritchie, C. Hanser Verlag 1983, ISBN 3-446-13878-1  
(Übersetzung von „The C-Programming Language“, Prentice-Hall 1977)
- [KER90] Programmieren in C, Brian W. Kernighan/ Dennis M. Ritchie, C. Hanser Verlag 1983 2. Ausgabe ANSI-C, ISBN 3-446-15497-3
- [KÜV99] Informatik für Ingenieure, Gerd Kuvler/ Dietrich Schwoch, Vieweg Verlag , 2. Auflage 1999, ISBN 3-528-14952
- [PRA98] Programmiersprachen – Design und Implementierung, Terrence Pratt/ Marvin Zelkowitz, Prentice-Hall 1998, ISBN 3-8272-9547-5
- [STR98] Die C++ Programmiersprache, Bjarne Stroustrup, Addison Wesley, 3. Auflage 1998, ISBN 3-8273-1296-5
- [THI85] Die 8087/80287 numerischen Coprozessor-Erweiterungen für 8086/80286 Systeme, Klaus-Dieter Thies, te-wi Verlag 1985, ISBN 3-921803-53-5