Datenverarbeitung für Physiker I

Vortragende:

M. Gröschl, E. Jericha, M. Mantler, H. Müller

J. Redinger, F. Sauerzopf

Skriptum:

M. Mantler, H. Müller, F. Sauerzopf

Redaktion: H. Müller

11. Oktober 2005

Inhaltsverzeichnis

Abbildungsverzeichnis

Vorwort

Dieses Skriptum wendet sich an die Hörer der Vorlesung mit integrierter Übung:

Datenverarbeitung für Physiker I

für die Studienrichtung "Technische Physik" an der TU-Wien.
Es ist nicht empfehlenswert den Stoff für diese Vorlesung allein anhand dieses Skriptums zu erlernen, da im praktischen Teil dieser Lehrveranstaltung Beispiele vorgeführt werden, die dieses Skriptum ergänzen und einen wesentlichen Anteil am Inhalt dieser LVA haben. Nur einige dieser Beispiele sind in diesem Skriptum vorhanden. Sie sind nie ganz ausprogrammiert und enthalten oft nur fragmentarische Programmabschnitte, die den Lehrstoff erklären, vielleicht als Vorlage, aber keinesfalls als fertige ausgereifte Programme anzusehen sind.

Bei einem Skriptum über Datenverarbeitung läßt es sich auf Grund der vielen englischen Fachaudrücke leider nicht vermeiden, daß viele Anglismen im deutschen Text verwendet werden. Wir haben hoffentlich nirgends vergessen die entsprechenden Übesetzungen anzugeben.

Dieses Skriptum und auch die zusätzlichen kommentierten Beispiele finden Sie in maschinenlesbarer Form unter der URL:

`http://www.ifp.tuwien.ac.at/institut/skripten/`

...

1 Grundlagen

1.1 Erstellen von Programmen

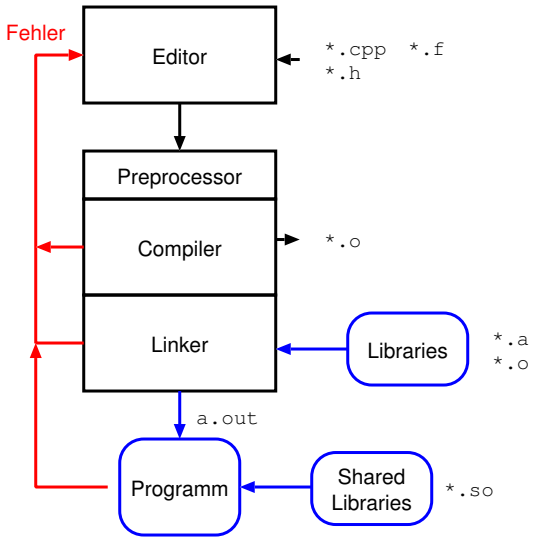

Abbildung 1.1: Compilieren & linken

Der **Editor** (z.B. `emacs, gvim, vim, vi, nedit`) erlaubt die Eingabe und die Bearbeitung der Quelltexte (`*.c, *.C, *.h; *.cpp, *.hpp oder *.f`).

Der **Compiler** (z.B. `gcc, f77`) dient zum Erzeugen eines lauffähigen Programms aus einem Quelltext. Viele moderne Produkte enthalten neben dem eigentlichen Compiler auch einen Preprocessor und den Linker.

Der **Preprocessor** bindet externe Quelltextpassagen ein, schließt Quelltextpassagen nach Maßgabe vordefinierter Bedingungen ein oder aus und ersetzt symbolische Konstanten durch wirkliche Werte. Das Ergebnis ist ein Quellprogramm, das dem eigentlichen Compiler zugeführt wird.

Der eigentliche **Compiler** bildet aus dem Quelltext den Maschinencode (Object-Code, `*.o`). Dieser enthält jedoch erst einstweilige Verweise auf externe Funktionen, deren Code im gegenständlichen Modul nicht enthalten war, z.B. Bibliotheksfunktionen oder eigenen Code in anderen Dateien.
Der **Linker** verbindet die einzelnen Object-Codes und fügt den Code der Bibliotheksfunktionen (`lib*.a`) ein. Das Ergebnis ist ein ausführbarer Code (wenn kein anderer Name angegeben wird, `a.out`), der die Informationen zum richtigen Laden und Initialisieren (Aufgabe des Betriebssystems) enthält.

Bibliotheken der C- (und C++) Standardfunktionen werden vom Compiler-Hersteller bereitgestellt. In Fortran sind Standardfunktionen in die Sprache integriert. Häufig verwendete Funktionen können auch vom Anwender zu eigenen Bibliotheken zusammengefaßt werden.
Optional steuert das `Makefile` (über das make-Programm) das Compilieren und Linken der einzelnen Teile des Programms. Es ist insbesondere bei größeren Projekten von Vorteil, weil automatisch nur jene Module neu übersetzt werden, die einer Änderung unterzogen wurden.

1.1.1 Ein einfaches C-Programm

Ein einfaches Programm ist:

```
# include <stdio.h>              /* Fuer Preprocessor  */

int main()                       /* Header von main() */
{                                /* Beginn von main() */
   printf("Hello, world\n");     /* Funktionsaufruf   */
}                                /* Ende von main()   */
```

- Es beginnt mit der (praktisch in allen Programmen enthaltenen) Anweisung an den Preprocessor: `#include <stdio.h>`
  mit der Wirkung, daß an dieser Stelle die Datei `stdio.h` eingefügt wird.
- Gefolgt von einem Kommentar: `/* Fuer Preprocessor*/`
- Mit dem Funktionskopf: `int main()`
  wird eine Funktion `main()` eingeleitet.
- Deren Code in einem Block `{...}` folgt: `{printf("Hello world\n");}`

Nach Belieben können überall, wo Trennzeichen erlaubt sind, zusätzliche Leerzeichen, Kommentare (s.u.) und Zeilenvorschübe eingefügt werden.

Die kleinste Einheit eines Programms ist der **Befehl (statement)**

- Wird durch Strichpunkt ; beendet
- Leere Befehle enthalten nur einen Strichpunkt

Befehlsgruppen werden zu **Blöcken** (*blocks*) zusammengefaßt. Blöcke sind durch geschwungene Klammern *(braces)* eingeschlossen und legen den Gültigkeitsbereich (scope) lokaler Variablen fest. Blöcke werden auch als **zusammengefaßte Befehle** (*compound statements*) bezeichnet und in Schleifen, logischen Abfragen, usw. wie ein einziger Befehl behandelt.

Achtung: Jeder Einzelbefehl wird mit einem Strichpunkt abgeschlossen, nicht jedoch der Block selbst, das wäre ein leerer Befehl nach einem Block. Nicht: "`{...};`"

Aber: Ein Strichpunkt folgt als Abschluß von Initialisierungen und Struktur-Deklarationen.

Blöcke können beliebig geschachtelt sein:

```
{ ...Befehle... {... weitere Befehle... {... weitere Befehle... } }
 ... weitere Befehle...
}
```

Jeder äußerste Block ist eine **Funktion** (*function*) und kann durch einen Namen angesprochen werden und mit Hilfe von symbolischen Parametern Daten austauschen. Er wird mit einem **Kopf** (*header*) versehen, der den Namen enthält und in dem die übergebenen Parameter und der Typ des Rückgabewerts deklariert werden. Eine Funktion kann einen Wert zurückliefern, muß aber nicht.

C ist sehr funktionsorientiert. So ist etwa `printf()` eine Funktion (die auch einen Wert zurückliefert, nämlich die Anzahl der ausgegebenen Zeichen).

`printf()` gehört zu den Standardfunktionen, das sind elementare Funktionen wie auch `fread()`, `fwrite()`, `sin()`, `tan()`, usw. Sie sind nicht in der Sprachdefinition von C, sondern in eigenen Bibliotheken enthalten. Der minimale Umfang der Standardbibliothek ist durch Normen festgelegt. Nachdem über viele Jahre die Beschreibung aus dem Standardwerk von Kernighan und Ritschie (K&R) die de-facto Norm war, wird jetzt die neuere Normierung durch ANSI allgemein befolgt.

Auch `main()` ist wie jede andere Funktion aufgebaut. Sie kann Parameter (vom Betriebssystem) übernehmen (i.a. die Kommandozeilen-Parameter), manchmal auch zurückgeben (z.B. Fehlernachrichten, Statusmeldungen).

Ein **Modul** (*module*) ist eine Datei, die eine oder mehrere Funktionen (sowie eventuell zusätzliche, externe Deklarationen und Preprocessor-Anweisungen) enthält. Jedes Modul wird vom Compiler als Einheit (compilation unit) verarbeitet.

- Ein **Programm** besteht aus einer Aneinanderreihung von Funktionen.
- Genau eine davon muß den Namen **main** haben.
- Mit dieser Funktion beginnt das Programm beim Start.
- Die Reihenfolge der Funktionen im Quelltext ist egal.
- Die Funktionen können auf mehrere Dateien (Module) aufgeteilt sein, doch ist jede Funktion für sich unteilbar.

Kommentare (*comments*)

1. Kommentare werden zwischen `/*` und `*/` geschrieben:

   ```
   /* GAUSS FIT written by N. Nostren *//* Version 1.0a */
   ```

   Schachtelung ist nicht erlaubt:

   ```
   /* Die äußere Schachtel /* die innere Schachtel */ GEHT NICHT */
   ```

   Ein Kommentar kann jedoch auch innerhalb eines Befehls stehen:

   ```
   printf(/*Parameterliste*/"Ausgabetext");
   ```

2. Alternativ in C++:
   Der Kommentar beginnt (an beliebiger Stelle einer Zeile) mit doppelten Schrägstrichen und endet mit dem Zeilenende.

   ```
   printf("Program halted"); // Fatal Error
   ```

Kombinationen von beiden Typen sind in C++ möglich, doch (standardmäßig) nicht geschachtelt. Viele C++ Compiler erlauben auch in C-Programmen die `//`-Notation (manchmal einstellbar).

2 Einfache Sprachelemente

2.1 Einfache Zuweisungen und Objekte

Speicherplätze, die Zahlenwerte (also ein Bitmuster) aufnehmen können, werden durch Namen (Variablen) angesprochen. Sie sind damit einfache Objekte. Um das Bitmuster korrekt interpretieren zu können, muß der Objekt-Typ angegeben werden, das heißt, die Variable wird ”deklariert”. Einfache Datentypen sind beispielsweise ganze Zahlen (integer) und Gleitkommazahlen (real/float).

```
int main()
{
   int myint;     /*integer*/
   float myfloat; /*floating point*/

   myint=100;
   myfloat=3.14;

   printf("%i %f\n",myint,myfloat);
}
```

Groß/Kleinschreibung wird in C und C++ beachtet (`Banane` und `bAnane` sind zwei verschiedene Variablen).
Die maximale Länge des Variablennamens ist compilerabhängig und muß zur Erfüllung des Standards in C/C++ mindestens 31 Zeichen sein.

2.2 Einfache Felder = Vektoren = Arrays

2.2.1 Syntax

Vektoren enthalten mehrere (im Speicher aneinandergereihte) Objekte eines Typs. Mit Hilfe des Index' kann auf die einzelnen Elemente zugegriffen werden.

Fünf `int`-Variablen: `iA[5]`
Hier sind die Inhalte der Speicherstellen, die symbolisch mit `iA[0],iA[1],...` bezeichnet wurden, dargestellt.

57	7	34	1234	-678
`iA[0]`	`iA[1]`	`iA[2]`	`iA[3]`	`iA[4]`

Statische Feldvereinbarungen erfolgen nach dem Vorbild normaler Typvereinbarungen, wobei die Feldgröße durch eine Konstante festgelegt wird (der Compiler kann

sie auch aus der Anzahl der Initialisierungselemente selbst bestimmen). Jedenfalls muß die Größe des Felds zum Zeitpunkt des Compilierens festgelegt sein. Der Compiler reserviert den Speicherplatz (= Größe eines Elementes (in bytes) * Anzahl der Elemente).
Dynamische Feldvereinbarungen sind ebenfalls möglich.
Genauere Erklärungen dazu folgen im Kapitel über **Zeiger** (*pointer*).

Der Feldindex wird in eckigen Klammern angegeben und ist vom Typ int

Man kann beliebige berechenbare Ausdrücke angeben, wenn diese durch automatische Standardkonversion in den Typ int umgewandelt werden können.

Der Index [0] bezeichnet das erste Feldelement

Beachten Sie, daß daher bei einer Dimensionierung [N] der Feldindex maximal den Wert [N-1] annehmen darf. Die Dimensionierung beschreibt die Anzahl der Feldelemente, nicht den maximalen Indexwert.

```
/* Vereinbarung */
/* 100 int-Feldelemente */
int iArray[100];
int i;

/* Durchlaufen des Arrays */
/* Zuweisung von Werten */
for(i=0;i<100;i++)iArray[i]=i*2;

/* Ausgabe */
for(i=0;i<100;i++)printf("iA[%d]:%d\n",i,iArray[i]);
```

Erzeugt int-Array mit 100 Elementen. Belegt das Array mit Werten (i*2). Druckt die Werte samt Index wieder aus.

Die Einhaltung der Feldgrenzen wird weder vom Compiler noch zur Laufzeit geprüft!
Das Überschreiten des vereinbarten Speicherbereichs ist eine der tückischesten Fehlerquellen in der Fortran und C-Programmierung. Zur Fehlervermeidung gibt es jedoch Compileroptionen mit denen eine Feldüberschreitung zur Laufzeit laufend überprüft werden kann.

2.2.2 Initialisieren von Arrays

Ein Array kann durch Definition initialisiert werden:

```
int iDown[10]={10,9,8,7,6,5,4,3,2,1};
float v[3]={3.33,4.44,5.55};
```

in solchen Fällen kann der Compiler selbst die Felddimensionierung vornehmen:

```
int iDown[]={10,9,8,7,6,5,4,3,2,1};
float v[]={3.33,4.44,5.55};
```

2.2.3 Felder von Zeichen

Einen eigenständigen Datentyp ”String” gibt es in C nicht.

Ein String ist ein Array von Zeichen

Da es keine Funktion gibt, die das Ende eines Felds erkennen kann, wird das Ende einer Zeichenkette mit dem Zeichen `\0` (=ASCII-Zeichen Null) symbolisiert. Viele Funktionen bearbeiten Zeichenketten nur bis zum ersten auftretenden `\0` oder schließen ein Ergebnis (neuen String) damit ab. Bei der Größenvereinbarung ist der erforderliche Platz für das Zeichen `\0` zu berücksichtigen.

2.2.4 Mehrdimensionale Arrays

Beispiel für ein initialisiertes, zweidimensionales Array:

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

Die Indexfolge entspricht hier [Zeile][Spalte] ([row][column]). Die inneren Klammern unterstreichen die Struktur, sind aber nicht erforderlich:

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

Bei der Dimensionierung kann der erste Index weggelassen werden

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

Felder von Zeichen:

```
char cSchirm[25][80];
```

ist ein Array von Arrays, das 25 Arrays vom Typ char mit je 80 Elementen (= 2000 Zeichen = Bytes) speichern kann.

Deklaration und Initialisierung von Strings

`'\0'` wird automatisch angehängt

```
char szString[9]="Beispiel"; /* [9] für 8 Zeichen plus '\0' */
/* oder */
char szString[]="Beispiel";  /* Compiler berechnet Länge */
```

Auf ein einzelnes Zeichen wird einfach zugegriffen:

```
szString[0]='B'; szString[4]='p'; szString[8]=0;
```

Beachten Sie die unterschiedliche Verwendung von einfachen und doppelten Anführungszeichen:

`'A'` ... Ein Zeichen, ASCII-Code von A also eine Integerzahl.
`"A"` oder `"ABCd"` ... Eine Zeichenkette, ein Feld mit abschließendem `\0`-Zeichen.

Es ist **nicht** möglich, einem String außerhalb der Deklaration durch Zuweisung einen Wert zu geben:

```
char szS[9];              /* nicht initialisierter String */
char szT[9]="Beispiel";   /* OK in der Deklaration */
szS="Beispiel";           /* Fehler, Zuweisung außerhalb der Deklar. */
szS=szT;                  /* Geht auch nicht (siehe "pointer") */
```

Für das Kopieren von Strings gibt es einige Bibliotheksfunktionen. Einhaltung der vereinbarten Feldgrenzen beachten!

2.3 Aufbau einer Funktion

Kopf + Block *(header + body)*

Bevor eine Funktion aufgerufen (verwendet) wird, muß sie deklariert oder definiert werden.

Deklaration: Vereinbarung des Variablentyps, bei Funktionen die gesamte Information des Kopfs über die Datentypen.

Definition: Deklaration + Wertzuweisung, bei Funktionen Kopf + Code:

```
KOPF
 {...Befehle...}
```

KOPF (HEADER):

Typ	Funktionsname	(Parameterliste)
Variablentyp des Rückgabewerts	Beliebiger Name	Zu übergebende Parameter, jeweils mit Angabe der Variablentyps (immer paarweise anzugeben) Trennzeichen: , (Beistrich) nach jeder Variablen (nicht zwischen Typenangabe und Variablen!).

Beispiel: Definition einer Funktion

```
float Grad_to_Rad (double WinkelGrad)     /*KOPF  (header)*/
{return WinkelGrad*3.14/180.;             /*BLOCK (body)*/
}
```

Übergabe-Parameter: `WinkelGrad` vom Typ `double`

Rückgabewert: Zahlenwert vom Typ `float`

```
float Volumen (float A, float B, float C)
{return A*B*C;
}
```

Beispiel: Deklaration einer Funktion

Angabe von Kopf + Strichpunkt

Die Namen der Variablen in der Parameterliste können weggelassen werden:

```
float Grad_to_Rad (double WinkelGrad);
```
oder
```
float Grad_to_Rad (double);
```

```
float Volumen (float A, float B, float C);
```

oder

```
float Volumen (float, float, float);
float Volumen(float, float, float); /* auch OK */
```

Die Funktionsdeklaration oder Definition muß vor dem ersten Aufruf und außerhalb jeder anderen Funktion erfolgen. Jede Funktion muß deklariert werden, auch die Funktionen der Standardbibliothek. Wird eine Funktion zunächst nur deklariert und erfolgt die Definition (mit Code) später im Programm, so ist dort der vollständige Funktionskopf (ohne Strichpunkt) nochmals vor dem Code (im Klammernpaar { }) einzugeben.
Die Deklaration der Funktionen der Standardbibliothek erfolgt praktisch immer in ”Header-Dateien”(siehe unten). Der (bereits compilierte) Code ist in den entsprechenden Bibliotheken (libraries) enthalten und wird durch den Binder (linker) eingebunden.
Beispiel:
Aufteilen eines ”Projekts” in 3 Dateien; eine Header-Datei und zwei Module

Header-Datei `myfuncs.h`:	Modul `myfuncs.c`:
Enthält Funktions-Deklarationen	Enthält Funktions-Implementationen

`int SumInts(int from, int to);`	`#include "myfunc.h"`
	`int SumInts(int from, int to)`
	`{int i,s;`
	`for(i=from,s=0; i<=to; i++)s+=i;`
	`return s;`
	`}`

Modul: `main.c`:
Verwendung der Funktionen

`#include <stdio.h>`
`#include "myfunc.h"`
`main()`
`{printf("Summe der Zahlen von 1-100: %d\n",SumInts(1,100));`
`}`

Compilieren erfolgt mit:
`gcc -c myfunc.c main.c` #ergibt die Dateien `myfunc.o, main.o`

Linken erfolgt mit:
`gcc -o sum myfunc.o main.o` #ergibt lauffähiges Programm `sum`

2.4 #include-Anweisungen und Header-Dateien

Programme enthalten meist zu Beginn Zeilen wie:

```
#include <stdio.h>

#include "mydefs.h"
```

Mit dem Zeichen # werden directives und macros eingeleitet, die dann vom Preprocessor umgewandelt werden. Als Folge der `#include`-Anweisung *(directive)* wird die angegebene Datei an dieser Stelle eingefügt; die Wirkung ist die gleiche wie durch Kopieren oder Eintippen. Die Dateien können beliebigen Code (oder Code-Teile) enthalten, der syntaktisch an die angegebene Stelle paßt.

Variante 1: ”Pfad+Dateiname”
Die Datei befindet sich an der durch den Pfad angegebenen Stelle (die Interpretation ist von der Implementierung und vom Betriebssystem abhängig). Wird sie dort nicht gefunden, wird Variante 2 probiert.

Variante 2: <Dateiname>
Die in spitzen Klammern < > angegebene Datei befindet sich in einem voreingestellten Suchpfad des Compilers (Compiler-Systemverzeichnis). Diese Suchpfade werden bei der Installation des Compilers definiert, können aber vom Anwender durch Compiler-Optionen (switches) erweitert werden. Die Interpretation der Zeichen ’ " \ / oder * in Dateinamen innerhalb von < > ist nicht definiert.

Als (System-)Header-Dateien bezeichnet man Dateien, die Deklarationen von Systemfunktionen enthalten, daneben auch Deklarationen und/oder Definitionen von (symbolischen) Konstanten und/oder Variablen. Der Dateityp ist *.h (in C) oder *.hpp (in C++). Man kann und **soll** natürlich auch eigene Header-Dateien für häufig wiederkehrenden Code (Definitionen oder Deklarationen) verwenden und beliebig bezeichnen.
Vorteile:

- Spart Aufwand und Platz. System-Header werden vom Compilerhersteller bereitgestellt.
- Änderungen in der (eigenen) Header-Datei wirken sich auf alle Programme aus, in die sie eingebunden ist (natürlich nur durch [Neu-] Compilieren).

Achtung: Die Angabe von Header-Dateien bewirkt nicht die Einbindung der Bibliotheksroutinen im Link-Vorgang. Wird zum Beispiel eine Sinus-Funktion im Programm benötigt, muß die Header-Datei `<math.h>` zuvor angegeben werden.

Damit wird beispielsweise (neben anderen mathematischen Funktionen)

```
double sin(double);
```

deklariert, aber kein Code für den Sinus definiert. Die Mathematik-Bibliothek wird standardmäßig nicht miteingebunden und muß mit `-lm` (gcc in Linux oder Unix) extra verlangt werden:

```
gcc programm.c -o programm -lm
```

Die korrekte Behandlung von Header-Dateien und mehreren Quelldateien in größeren Programmen verlangt somit eine gewisse Aufmerksamkeit beim Kompilieren und Linken. Dieser Aufwand kann durch einmaliges Erstellen einer make-Datei (Makefile) minimiert werden. Die kurze Einarbeitung in einfache Makefiles (siehe Anhang B, Make) ist daher jedenfalls zu empfehlen.

2.5 Beenden eines Programms

Normalerweise endet ein Programm nach der Anweisung `return()` in `main()`, danach übernimmt wieder das Betriebssystem die Kontrolle. Wie bei allen Funktionen muss dabei der zurückgegebene Wert dem Typ der Funktion entsprechen. Standardmäßig sollte der Rückgabewert von `main()` eine ganze Zahl sein, mit deren Hilfe man verschiedene Ergebnisse des Programmes zur Weiterverarbeitung charakterisieren kann (z.B. Fehler oder Erfolg). Diese Auswertung wird sehr oft in BATCH-Dateien zur weiteren Steuerung des Ablaufs verwendet. Man kann das Programm auch mit der Funktion `exit()` anhalten (z.B. in tief geschachtelten Funktionen), die wiederum einen (wählbaren) Wert an das Betriebssystem übergibt.

3 Datentypen und Wertebereiche

3.1 Bezeichnung von Objekten

Objekte (=Variablen) *objects (=variables)* werden durch Namen bezeichnet.

Name, Bezeichner *(identifiers)*:
Eine Folge von Buchstaben, Ziffern und _ *(underscore)*, die mit einem Buchstaben beginnt. Darf nicht gleich einem **Schlüsselwort** (*keyword*) sein. Interne Namen können eine Länge von bis zu 31 Zeichen haben (ANSI-C Mindestanforderung), und es wird zwischen Groß- und Kleinschreibung unterschieden.

Die Länge externer Namen wird manchmal durch das Betriebssystem, (fremde) Linker oder durch Eigenschaften anderer Compiler stärker eingeschränkt, und es wird dann oft auch nicht nach Groß/Kleinschreibung unterschieden (z.B. mixed-code Problem mit FORTRAN).

Namen tragen:

- **Funktionen** (*functions*)
- **Struktur-tags** (*tags of structures*)
- **Aufzähler** (*enumeration constants*)
- **Typendefinitionen** (*typedef-names*)
- **Objekte** (*objects*) = **Variablen** (*variables*)

Empfehlung K&R:

- Kleinbuchstaben für lokale Variablen
- Großbuchstaben für symbolische Konstanten (z.B. PI)
- *ungarische Notation*: Bei umfangreichen Programmen (mehrere Autoren)

Zum Beispiel: `iLoop, fDeltaX, szFileName, iMax_FieldSize`

Unzulässig: `1Zahl, 12xy, a&b, %Satz`

Die **Ungarische Notation** ist hilfreich, um Bezeichner im Programmfluß leicht verstehen und identifizieren zu können.
Der Name einer Variablen (Funktion) besteht aus 2 Teilen:

1. Einem Präfix (Kleinbuchstabe), das den Datentyp der Variablen angibt.

c	Zeichen (char)
by	BYTE (unsigned char)
i(n)	Ganzzahl (int)
s	String
sz	String mit Null-Abschluß
x,y	(Bildschirm) Koordinaten (short)
cx,cy	x-, y-Länge (short)
l	LONG (long)
w	UINT oder WORD (unsigned)
dw	DWORD (unsinged long)
b	BOOL (int)
p	Zeiger (Pointer) (*)
fn	function

   Es können mehrere Präfixe sinnvoll kombiniert werden.

2. Einem kennzeichnenden frei wählbaren Namen, der mit einem Großbuchstaben beginnt:

```
iCount       // Integer; Zählvariable
szFilename   // String mit Null am Ende
dwByteCount  // unsigned long
bDrawn       // Boolsche Variable (0/1)
```

Einige Namen sind in C und C++ mit festen Bedeutungen vorbelegt → **Schlüsselwörter** (*keywords*). Sie können nicht für Namen verwendet werden. Alle Schlüsselwörter werden in Kleinbuchstaben geschrieben. Daher erlaubte, jedoch nicht zweckmäßige Variablennamen:

```
ELSE, NEW,...
```

3.2 Konstanten

Konstanten (*constants*) können entweder als (ausgeschriebene) **Zahlenwerte** (*literal constants*) oder **Zeichenketten** (*string literals*) oder durch symbolische Konstanten angegeben werden, die durch `#define`-Macros definiert sind (siehe unten).
Symbolische Konstanten sind nur im Quelltext bekannt und werden vom Preprocessor durch *"suchen und ersetzen"* substituiert. Der eigentliche Compiler sieht ihre Namen nicht mehr. Sie belegen daher auch keine (z.B. durch den debugger) ansprechbaren Speicherstellen.

Neben den genannten Konstanten gibt es auch Variablen, die als `const`(-ant) vereinbart werden, also ihren Wert nicht ändern dürfen (siehe: Variablentypen). Sie sind aber innerhalb ihres Definitionsbereichs echte Variablen.

3.2.1 Ganze Zahlen *(integer constants)*:

Binäre, oktale, dezimale und hexadezimale Schreibweise sind möglich.

Beispiele:

oktal:	`0376` (beginnt mit Null)
dezimal:	`2549`
hexadezimal:	`0xFE` (Buchstaben: X ABCDEF; Groß- oder Kleinschreibung erlaubt)

Der Compiler bestimmt, wieviele Bytes zur Speicherung günstig und notwendig sind (ist daher nicht einheitlich für alle Compiler und Betriebssysteme). Will man ein Format selbst wählen, so kann man ein geeignetes Suffix anhängen.

`l` ("kleines L") oder `L`:	`long`	
`u` oder `U`:	`unsigned`	(vorzeichenlos, nur positiv)

Kombinationen sind möglich.
`500L` wird als long int gespeichert.

Damit legt man zumindest den Typ fest. Die Byte-Zahl für einen Typ ist im allgemeinen nicht veränderbar oder wählbar.

3.2.2 Aufzählungs-Konstanten *(enumeration constants)*:

sind vom Typ `int` (siehe Variablenvereinbarungen, 3.4.7)

3.2.3 Gleitkomma-Zahlen *(floating constants)*:

Normale oder wissenschaftliche Darstellung:

z.B.: `2.345` `oder` `1.33E-13` `oder` `567e-1`

Die interne Darstellung von Gleitkommazahlen erfolgt als Exponent und Mantisse und belegt den Speicherplatz des Typs double. Ist der Typ float ausdrücklich erwünscht, so ist das Suffix F zu verwenden:
z.B.: `9.876F`

3.2.4 Zeichen-Konstanten *(character constants)*:

In westlichen Sprachen ist jede char-Konstanten ein 8-bit ANSI-Zeichen (es gibt insgesamt 256 Zeichen, davon sind die ersten 128 durch den ASCII-Standard definiert, die zweite Hälfte von der gewählten Codepage abhängig). Für fernöstliche Sprachen werden 16-bit Zeichen verwendet.
Allgemeines Konzept: 16-bit UNICODE

Ein einzelnes druckbares Zeichen wird zwischen einfachen Anführungszeichen '...' angegeben.
z.B.: `'A', '/', '*', ...`
Nicht druckbare Zeichen werden mit Hilfe von `\` *(backslash)* dargestellt (siehe Tabelle).
In C können Ausdrücke wie: `'A' + 32` (gibt 'a')
durchaus compiliert werden und einen Sinn ergeben.

Escape Code	Bedeutung	ASCII-Wert		
		Dec	Hex	Symbol
`'\a'`	Alarmton Audible alert	7	07	BEL
`'\b'`	backspace	8	08	BS
`'\f'`	Seitenvorschub formfeed	12	0C	FF
`'\n'`	Zeilenvorschub newline	10	0A	LF
`'\r'`	Wagenrücklauf carriage return	13	0D	CR
`'\t'`	Tabulator horizontal horizontal tab	9	09	HT
`'\v'`	Tabulator vertikal vertical tab	11	0b	VT
`'\\'`	Backslash	92	5C	
`'\''`	Einfaches Anführungszeichen single quote	39	27	'
`'\"'`	Doppeltes Anführungszeichen double quote	34	22	"
`'\?'`	Fragezeichen question mark	63	3F	?
`'\000'`	Zeichen im Octal Code			
`'\x00'`	Zeichen im Hex Code			

3.2.5 Symbolische Konstanten

Bezeichner für symbolische Konstanten, die üblicherweise in Großbuchstaben geschrieben werden (leicht zu finden). Sie werden mit einem `#define`-Makro definiert:

```
#define IMAX_INDEX 100
#define YES        'Y'
#define DROPOUT    0.7
#define SZYES_NO   "<Ja/Nein>"
/*Solche Namen sind nahezu selbsterklärend*/
```

Achtung: Kein Strichpunkt als Abschluß (würde Teil der Definition werden).

Das Zeilenende beendet normalerweise die Definition, allerdings kann eine Verlängerungszeile durch das Zeichen `\` am Zeilenende (d.h. unmittelbar vor dem

<RETURN>) bewirkt werden.
Die Wirkung des #define- Makros ist ähnlich wie *"suchen und ersetzen"*. Es können auch intelligente (funktionsähnliche) Vorschriften definiert werden, die im Kapitel Preprocessor erklärt werden.
Die Verwendung von symbolischen Konstanten ist **unbedingt zu empfehlen!** Reine Zahlenwerte sind kaum selbsterklärend.

Was ist 8.8705e10 ? Besser z.B.:

```
#define PI 3.14159265358979323846  /* meist in math.h enthalten */
#define EPSNULL 8.854e-12          /* Dielektrizitätskonstante */
#define R_EPS4PI (1./(4.*PI*EPSNULL))
```

Man kann davon ausgehen, daß moderne optimierende Compiler keine Probleme mit der Eliminierung der eventuell wiederkehrenden Multiplikationen im Code haben (macht manchmal schon der Preprocessor).
Da der definierte Ausdruck genau wie in der Definition angegeben in möglicherweise komplexe Ausdrücken eingefügt wird, sollte man zur Sicherheit ausreichend Klammern setzen.
Vorteile:
Code ist besser lesbar, Tippfehlergefahr wird verringert. Eventuelle Änderungen des Werts brauchen nur einmal im gesamten Programm vorgenommen werden. Häufig verwendete Konstanten können in einer INCLUDE-Datei gesammelt und durch eine Programmzeile in jedes Programm aufgenommen werden. Viele Konstanten sind in den System-Header-Dateien vordefiniert.

3.2.6 Zeichenketten *(string literals)*:

(Konstante) Zeichenketten werden in der Form:

"Folge von Zeichen und ESC-Sequenzen"

angegeben. Der Doppelapostroph ist nicht Teil der Zeichenkette, jedoch wird (automatisch) das Zeichen '\0' (ASCII-Null) zur Kennzeichnung des Endes der Zeichenkette angefügt. Aneinandergereihte Zeichenketten werden zu einer **verknüpft** (*string concatenation*) und automatisch mit '\0' abgeschlossen:
"ABC" "DEF" "GH" ist gleich "ABCDEFGH"

Achtung: 'A' und "A" sind völlig unterschiedliche Datentypen!
'A' ist vom Typ char, d.h. eine einfache Variable,
dagegen ist "A" vom Typ char*, das ist eine **Zeigervariable** (*pointer*).

Ausschnitt aus math.h

```
/* Some useful constants.  */
# define M_E            2.7182818284590452354   /* e */
# define M_LOG2E        1.4426950408889634074   /* log_2 e */
# define M_LOG10E       0.43429448190325182765  /* log_10 e */
# define M_LN2          0.69314718055994530942  /* log_e 2 */
# define M_LN10         2.30258509299404568402  /* log_e 10 */
# define M_PI           3.14159265358979323846  /* pi */
# define M_PI_2         1.57079632679489661923  /* pi/2 */
# define M_PI_4         0.78539816339744830962  /* pi/4 */
# define M_1_PI         0.31830988618379067154  /* 1/pi */
# define M_2_PI         0.63661977236758134308  /* 2/pi */
# define M_2_SQRTPI     1.12837916709551257390  /* 2/sqrt(pi) */
# define M_SQRT2        1.41421356237309504880  /* sqrt(2) */
# define M_SQRT1_2      0.70710678118654752440  /* 1/sqrt(2) */
```

Ausschnitt aus limits.h

```
#  define CHAR_BIT      8

#  define SCHAR_MIN     (-128)
#  define SCHAR_MAX     127

#  define UCHAR_MAX     255

#  ifdef __CHAR_UNSIGNED__
#   define CHAR_MIN     0
#   define CHAR_MAX     UCHAR_MAX
#  else
#   define CHAR_MIN     SCHAR_MIN
#   define CHAR_MAX     SCHAR_MAX
#  endif

#  define SHRT_MIN      (-32768)
#  define SHRT_MAX      32767

#  define USHRT_MAX     65535

#  define INT_MIN       (-INT_MAX - 1)
#  define INT_MAX       2147483647
#  define UINT_MAX      4294967295U
```

3.3 Objekte (Variablen)

Objekt: Benannte Speicheradresse, die Daten enthält. Der Name kann frei gewählt werden (siehe Bezeichner).

In C gibt es folgende Basisdatentypen:

Typ	Bits/Bytes*)	Wertebereich**)
char	$8/1$	$-128 \ldots +127$
unsigned char	$8/1$	$0 \ldots 255$
short int	$16/2$	$-2^{15} \ldots 2^{15}-1$
unsigned short int	$16/2$	$0 \ldots 2^{16}-1$
int	$32/4$	$-2^{31} \ldots 2^{31}-1$
unsigned int	$32/4$	$0 \ldots 2^{32}-1$
long int	$32/4$	$-2^{31} \ldots 2^{31}-1$
unsigned long int	$32/4$	$0 \ldots 2^{32}-1$
long long int***)	$64/8$	$-2^{63} \ldots 3^{63}-1$
unsigned long long int ***)	$64/8$	$0 \ldots 2^{64}-1$
float	$32/4$	$\sim 10^{-38} \ldots 10^{38}$ (7)
double	$64/8$	$\sim 10^{-308} \ldots 10^{308}$ (17)
long double	$96/12$	

*) Die Anzahl der Bits/Bytes ist vom Compiler und Rechner abhängig. Die hier angegebenen Werte sind die Standardeinstellungen für Linux/gcc. Der Typ char belegt hier immer 1 byte.
**) In Klammern steht die ungefähre Anzahl der signifikanten Dezimalstellen.
***) nicht Standard (im gcc verfügbar)

Um eine Variable vollständig zu beschreiben, sind zusammen mit dem Wert **sechs** Angaben notwendig:

Name:	`DeltaX`
Typ:	`float`
Wert:	`0.123`
Adresse:	`0x0BFFFFB38`

Name (Bezeichner): Damit kann die Variable im Programm identifiziert werden.

Typangabe: Legt den Wertebereich, den beanspruchten Speicherbereich, die Art der internen Darstellung und die Interpretation des Bitmusters fest.

Adresse: Speicheradresse, ab welcher der Wert der Variablen im Speicher steht. Ist das Objekt länger als 1 Byte, ist es die Adresse des ersten Bytes, die Länge des belegten Bereichs kennt der Compiler durch die Typangabe.

Definitionsbereich: Programmbereich, in dem die Variable unter ihrem Namen bekannt ist und angesprochen werden kann.

Lebensdauer: definiert, wann der Wert der Variablen undefiniert wird.

3.4 Variablenvereinbarung

Prinzip: Jede einfache Variable wird durch folgende Parameter deklariert (vereinbart) und eventuell definiert (wenn gleichzeitig eine Wertzuweisung durch den Initialisierer erfolgt). Die Variablenvereinbarung erfolgt immer am Beginn eines Blocks (anders als in C++, wo sie überall erlaubt ist).

Speicherklasse	**Typ**	Modifizierer	**Variable(n)**	Initialisierer;

Beispiel:

```
static int const MyInteger = 100;  /* initialisiert */
            int Myint2;            /* nicht initialisiert */
```

3.4.1 Speicherklasse (*storage class*)

auto*)	extern	static	typedef	register

*) default

`auto`**:** Default, muß nicht angegeben werden. Die Variable wird jedesmal bei Erreichen der Anweisung zur Typvereinbarung neu erzeugt und verliert ihre Definition (ihren Wert) beim Verlassen des umgebenden Blocks (lokale, temporäre Variable). Der zugewiesene Speicherplatz wird (möglicherweise) anderwärtig wiederverwendet und überschrieben. Auto-Variablen werden **nicht** automatisch mit Null initialisiert.

`extern`**:** Die Variable ist in einem anderen Modul (andere Datei = andere Übersetzungseinheit) global definiert (d.h. außerhalb jedes Blocks) und wird, wenn nicht anders angegeben, mit Null initialisiert. Sie ist im aktuellen Modul ab der Vereinbarung definiert. Solche Variablen sind gleichzeitig static, das heißt, der zugewiesene Speicherplatz bleibt während der gesamten Laufzeit des Programms erhalten.

`static`**:** Die Variable ist innerhalb oder außerhalb eines Blocks definiert und behält ihren zugewiesenen Speicherplatz über die Laufzeit des Programms. Sie wird, wenn nicht anders angegeben, automatisch mit Null initialisiert. Alle Variablen, die außerhalb jedes Blocks definiert werden, sind automatisch static und vom Ort der Definition bis zum Dateiende unter ihrem Namen ansprechbar.

`register`**:** Wie auto, jedoch wird nach Möglichkeit ein Prozessor-Register für die Speicherung verwendet. Meist ist die Angabe völlig sinnlos, weil der Optimierungsschritt des Compilers (nach oft massiven Umordnungen des Programms) solche Zuordnungen selbst trifft. Register-Variable haben keine ansprechbare Adresse (Adress-Operator `&` wäre hier illegal). Durch auto oder register vereinbarte Variablen werden am Stack gespeichert und nicht mit einem Anfangswert initialisiert.

`typedef`**:** Dient zur Erzeugung neuer Datentypen(namen) auf der Grundlage der Basistypen. Erzeugt keinen neuen Speicherplatz und wird hier nur zur K&R-Konformität angeführt.

3.4.2 Typ (*type specifier*)

	signed	unsigned	short	long
void	-	-	-	-
int	+	+	+	+
char	+	+	-	-
float	-	-	-	-
double	-	-	-	+
enum				
struct				
typdef				

Einige Datentypen entstehen durch zusammengesetzte Typvereinbarungen, z.B. `unsigned long int`. Die möglichen Kombinationen sind oben mit + angezeigt.
Dabei ist als Kurzschreibweise möglich:

```
        signed int  =  int
  (signed) long int  =  long
  (signed)short int  =  short
  unsigned long int  =  unsigned long
 unsigned short int  =  unsigned short
       unsigned int  =  unsigned
```

Die Länge der Datentypen (Bytes im Speicher) ist nicht durch die Sprachdefinition vorgegeben. Die Richtlinie ist ”zweckmäßiger, schneller Code”, also maschinen- und betriebssystemabhängig. Man kann mit Hilfe des Operators `sizeof()` die Anzahl der von einem Variablentyp belegten Bytes zur Laufzeit feststellen:

```
int LaengeInt, LaengeShort, LaengeLong;
   LaengeInt = sizeof(int);
 LaengeShort = sizeof(short);
  LaengeLong = sizeof(long);
```

`sizeof()` ist ein Operator, nicht eine Funktion. Er liefert den Speicherbedarf des Operanden als Integer-Zahl (in Bytes).

Allgemein gilt (es bedeutet <= ”kleiner oder gleich”):

```
sizeof(short) <= sizeof(int) <= sizeof(long)
sizeof(float) <= sizeof(double).
```

Das Vorzeichenverhalten von Char(acter)variablen ist in C (anders als in C++) nicht spezifiziert (daher sollte man immer `signed char` oder `unsigned char` angeben!).

3.4.3 Modifizierer (*type-qualifier*)

`const`: Die Variable kann nach der Initialisierung nicht mehr verändert werden (Fehlermeldung des Compilers). Da der Compiler keine Überprüfung z.B. von Unterprogrammen in fremden Modulen vornehmen kann, wird (meist) die Angabe `const` im function-header geprüft.

Beispiele:

```
int iCount,iEnd,iStart;
static float x,y,z;
float p;

const int IMAX_INDEX  = 100;
IMAX_INDEX=150; /* Fehler */
```

`volatile`: Kennzeichnet eine Variable, die möglicherweise durch Zugriff von außen (anderes Programm, DMA-Interface) verändert wird und verhindert verschiedene Optimierungsprozeduren (z.B. kann hier eine mehrmalige, gleiche Anweisung sinnvoll sein).

```
volatile int iV;
iV=0;
externe_Zugriffsroutine();
iV=0;   /* in diesem Fall sinnvoll, obwohl */
        /*keine für den Compiler ersichtliche Veränderung erfolgt ist */
```

3.4.4 Initialisierung (*initializer*)

Initialisierung = (Anfangs)Wertzuweisung. Sie erfolgt entweder zusammen mit der Deklaration:

```
int i={11};        /*Bei einfachen Datentypen kann {} entfallen*/
int i1=11;
char szName[10]={"Karli"};               /* Zeichenkette */
static int iArray[4]={1,2,3,4},iCount=0; /* Feld */
```

oder durch Zuweisung (Zuweisungsoperator) im Programmverlauf:

```
main()
{int iStart;   /* deklariert, aber noch nicht definiert*/
 iStart=0;     /* Initialisierung durch Zuweisung*/
}
```

3.4.5 Gültigkeitsbereich von Variablen

`auto` **Variablen:**

örtlich: Innerhalb ihres Blocks

zeitlich: Verlieren ihren Wert beim Verlassen des Blocks

Initialisierung: Nicht automatisch (!)

`static` **Variablen:**

örtlich: Ab der Definition bis zum Ende des Blocks. Bei Definition außerhalb jedes Blocks: bis zum Ende der Datei.

zeitlich: Bis zur Programmbeendigung

Initialisierung: Mit Null wenn nicht anders vorgegeben;

`extern` **Variablen:**

örtlich: Wie static Variablen

zeitlich: Wie static Variablen

Initialisierung: Im Modul der Definition, mit Null wenn nicht anders vorgegeben;

Beispiel:

```
#include <stdio.h>
int iGlobal=1;          /* Global ab hier */
main()
{int iMainLocal;
 static int iiGlobal;    /* mit Null initialisiert */
 int iGlobal=2;          /* Lokal in main */
   {int iG=iGlobal;         /* Wert=2 (lokal im Block) */
    int iL;                 /* keine automat. Initialisierung */
   }
 iMainLocal=iGlobal;   /* Wert=1 (globale Variable) */
}
```

Gültigkeitsbereich einer Variablen: bis zu den Grenzen des umgebenden Blocks.

Global: ab Definition bis Dateiende.

3.4.6 Logische Variablen (*boolean*):

In C gibt es keinen eigenen Datentyp für logische Variablen. Dafür wird zweckmäßig `int` oder `char` verwendet.
Es gilt allgemein in C (auch für `float/double`-Vaiablen):

Jeder Wert ungleich 0 gilt als wahr.
Jeder Wert gleich 0 gilt als falsch.

Beispiel:

```
int i=0, k=-10;
if (i) printf("\n Ich werde nicht gedruckt");
if (k) printf("\n Ich werde gedruckt");
```

3.4.7 Aufzählungstypenelemente (*enum*)

Beispiel: Farbenliste

1. Möglichkeit: Preprozessoranweisungen

```
#define BLACK 0
#define BLUE 1
...
#define WHITE 15
```

Nachteil: Lange, unflexible Liste (noch schlimmer wäre es, die Zahlenwerte direkt ins Programm einzubauen).

2.Möglichkeit: Aufzählungstypen

```
enum colors {BLACK,BLUE,GREEN,CYAN,RED,
             MAGENTA, BROWN, LIGHTGRAY,
             DARKGRAY,LIGHTBLUE,LIGHTGREEN,
             LIGHTCYAN,LIGHTRED,
             LIGHTMAGENTA,YELLOW,WHITE};
```

Die konstanten Elemente zwischen `{}` werden mit dem Bezeichner `colors` verbunden, welcher der Name eines neuen Datentyps ist. Das reservierte Wort `enum` weist den Compiler an, die Elemente zu numerieren. 0 für das erste (BLACK), 1 für das zweite (BLUE) usw. Das Numerieren besorgt der Compiler. Nach der Deklaration eines `enum`-Datentyps können neue Variablen erzeugt werden:

```
enum colors BackGround = BLUE;
```

Da `BackGround` eine Variable ist, kann sie geändert werden:

```
BackGround = RED;
```

Mit `const` kann eine ”konstante Farbe” definiert werden:

```
const enum colors Frame = YELLOW;
```

Weitere Beispiele:

```
enum WOCHENTAGE {MO,DI,MI,DO,FR,SA,SO};
enum {FALSE,TRUE}; //unbenanntes enum !!!
```

Normalerweise führt der Compiler die Nummernzuweisung automatisch durch, man kann aber auch Modifikationen angeben.
z.B.:

```
enum MONATE {JAN=1,FEB,MAR,APR,MAI,JUN,JUL,
             AUG,SEP,OKT,NOV,DEZ};           /* 1,2,3,...,12 */

enum errors {NO_ERR,MEM_ERR,INP_ERR,         /*   0,   1,   2, */
              OUT_ERR,DISK_ERR=100,READ_ERR, /*   3, 100, 101, */
              WRI_ERR,KEYB_ERR=200,BAD_ERR}; /* 102, 200, 201  */
/* Erleichtert das Einfügen neuer Werte */
```

3.4.8 typedef

Mit typedef können eigene Datentypnamen vereinbart werden.

```
typedef int Length;  /* Length wird zum Synonym für int */
Length Lx,Ly,Lz;     /* Deklaration von Lx,Ly,Lz als int */
```

Während diese Vereinbarung für die einfachen Datentypen entbehrlich erscheint, ist sie für komplexere **Strukturen** (*structures*) eine bequeme und übersichtliche Alternative. Auch für die Portierbarkeit bietet sie Vorteile, da die Länge (Speicherplatzbedarf) der Datentypen systemabhängig sind.
Beispielsweise kann

```
typedef int INT_4
```

in einem Headerfile vereinbart werden und dort für ein anderes System leicht z.B. in `typedef long INT_4` geändert werden, während alle betroffenen Variablen unverändert mit `INT_4` deklariert bleiben.

3.4.9 const-Deklarationen

`const` vor dem Objekt eines bestimmten Datentyps bedeutet, daß dieses Objekt nicht verändert werden darf.

```
const int nMaxIndex=5;
```

Bewirkt daß bei einer Zuweisung im weiteren Programmablauf, wie z.B. `nMaxIndex=10;` die Compilierung durch eine Fehlermeldung abgebrochen wird.

`const`-Parameter bei Funktionsaufruf verhindern unnötiges Kopieren.

```
double area(const double a, const double b)
```

Bedeutet, daß die Variablen `a, b` in der Funktion `area` **nicht** verändert werden und somit **keine** Kopie angelegt werden muß.

Man unterscheidet:

- Zeiger auf nicht veränderbaren Wert:

```
const char *name="abcdef";
const char *p;
p=name;                 // OK
*p ='x';                // Fehler!
```

- Nicht veränderbarer Zeiger auf Wert

```
char *name;
char * const p;
*p='x&';                  //OK
p = name;                 //Fehler!
```

Der Adresse eines `const`-Objektes kann nur einem Zeiger auf ein `const`-Objekt zugewiesen werden!

3.4.10 Zeiger (*pointer*)

Zeiger sind Variablen, deren Wert die (Speicher)-Adresse eines Objekts ist. Sie sind Voraussetzung für die dynamische Speicherverwaltung in C (und C++) und erlauben, zur Laufzeit des Programms Speicher vom Betriebssystem anzufordern und freizugeben (ähnlich wie der Heap in Pascal und anderen Sprachen).

Grundlagen:

- Ein Zeiger ist eine Variable wie jede andere.
- Eine Zeigervariable enthält eine Adresse, die auf eine Stelle im Speicher zeigt.
- An dieser Adresse steht ein Objekt, auf welches der Zeiger verweist. Dieses Objekt kann beispielsweise eine Variable sein, eine Datenstruktur, eine Funktion oder ein anderer Zeiger (es gibt keine Einschränkung).
- Ein Zeiger kennt den Typ der Daten auf die er zeigt oder ist vom Typ `void*`.

Vereinbarung von Zeigern:

Typangabe * Variablenname;

Variablenname bezeichnet eine Variable, die als Wert die Adresse eines Objekts der Art Typ hat.

z.B.:

```
  int* pi;    /* pi kann die Adresse einer
                 int-Variablen aufnehmen */
float* pWert; /*  pWert kann die Adresse einer
                  float-Variablen aufnehmen */
 char* pc;    /* pc kann die Adresse einer
                 char-Variablen aufnehmen */
```

Man kann `int*␣i` oder `int␣*i` schreiben, allerdings ist ist bei der Deklaration mehrerer Variablen der Stern entsprechend oft anzugeben:

```
int *i, *j, *k;     /* drei Pointer*/
int i,j,k,*l,m,*n;  /* zwei Pointer, 4 Variablen*/
int* i,j,k;         /*  ein Pointer, 2 Variablen */
```

Es ist daher oft klarer, wenn der Stern generell mit dem Variablennamen verknüpft wird.

Zur sinnvollen Verwendung von Pointern benötigt man noch zwei Operatoren:

1. indirection oder dereferencing operator `*`:

   `*` Zeigervariable

   Wirkung: Ergibt den Wert der Variablen, auf die die Zeigervariable weist.

2. Address-Operator `&`:
   `&` Variable

   Wirkung: Ergibt die Adresse (des ersten Bytes) von Variable.

Beispiel:

```
int   j;       /* Annahme: Adresse von j  = 0xbffff588 */
int   i = 23;  /* Annahme: Adresse von i  = 0xbffff584 */
int *pi;       /* Annahme: Adresse von pi = 0xbffff580 */

pi = &i;       /* pi erhält die Adresse von i (0xbffff584) */

j = *pi;       /* j = "Inhalt der Speicherstelle, auf die pi zeigt"
                  in diesem Beispiel dasselbe wie j=i */
```

Speicherauszug nach der Initialisierung		
Name	Adresse	Wert
pi	bffff580	-
i	bffff584	23
j	bffff588	-
*pi		-

Speicherauszug bei Programmende		
Name	Adresse	Wert
pi	bffff580	bffff584
i	bffff584	23
j	bffff588	23
*pi		23

Beispiel:
Das Format für die Ausgabe eines Pointers kann mit %p oder %x festgelegt werden. Im ersten Fall ist es maschinen- und compilerabhängig, im zweiten Fall einfach hexadezimal.

```
#include <stdio.h>

main()
{  int i, *pi;
 float f, *pf;
  char c, *pc;

 i=2;  f=3.14;  c='J';  pi=&i;  pf=&f;  pc=&c;

printf(" Adresse von Wert von\n");
printf("  i : %X  i: %d\n",&i,i);
printf(" pi : %X pi: %X\n",&pi,pi);
printf("  f : %X  f: %f\n",&f,f);
printf(" pf : %X pf: %X\n",&pf,pf);
printf("  c : %X  c: %c\n",&c,c);
printf(" pc : %X pc: %X\n",&pc,pc);

printf(" Verwendung von *:\n");
printf(" pi : %X *pi : %d\n",pi,*pi);
printf(" pf : %X *pf : %f\n",pf,*pf);
printf(" pc : %X *pc : %c\n",pc,*pc);
exit(0);
}
```

Ergebnis:

Name	Adresse	Wert
i	BFFFF5C8	2
pi	BFFFF5C4	BFFFF5C8
f	BFFFF5C0	3.14
pf	BFFFF5B0	BFFFF5C0
c	BFFFF5BB	'J'
pc	BFFFF5B4	BFFFF5BB

Verwendung von *:

Name	Wert
pi	BFFFF5C8
*pi	2
pf	BFFFF5C0
*pf	3.14
pc	BFFFF5BB
*pc	'J'

Der Operator * kann auch mehrfach angewendet werden (Pointer auf Pointer, Pointer auf Pointer auf Pointer, ...):

```
int i, *pi, **ppi;
i=5, pi=&i, ppi=&pi;
```

`i`: `int`-Variable
`pi`: Zeiger auf eine `int`-Variable
`ppi`: Zeiger auf einen Zeiger, der auf eine `int`-Variable verweist

Ergibt:

Variable	Adresse	Wert
i	FF00	5
pi	FF02	FF00
ppi	FF04	FF02
**ppi		5

Zeiger und Typprüfung:
Zeiger sind an Datentypen gebunden, da mit dem Zeiger auch die Information über die Länge (in Bytes) des zugehörigen Datentyps vorhanden ist und auch die jeweils richtigen Typ-Konversionen durchgeführt werden müssen.

```
float *pf;
 char c;
pf=&c       // Liefert Fehler beim Compilieren
```

Die Länge einer Adresse (in Bytes) ist von der Hardware und vom Betriebssystem abhängig.
Die Größe eines Zeigers kann mit dem Operator `sizeof` ermittelt werden:

```
   short *pi;
long int *pfi;

sizeof (pi);     // Ergibt 4 (Bytes)
sizeof (int *);  // Ergibt 4 !!
sizeof (pfi);    // Ergibt 4
```

NULL-Zeiger:
Ein NULL-Zeiger zeigt ”nirgendwo” hin. NULL kann jedem Zeiger zugewiesen werden:

```
pi=NULL;
```

Ausdrucksweise in C++:
Dieser Zeiger adressiert keine Daten (nicht gültige Daten).

NULL: Makro-Äquivalent von 0.

Definition in den HEADER-Dateien stdef.h, stdio.h, stdlib.h, string.h entsprechend der verschiedenen Speichermodelle.

Initialisierung von Zeigern:
Es gelten dieselben Regeln wie für das Initialisieren von Variablen.

```
void fn(void)
{ float *pf;
...
}
```

pf ist nicht initialisiert und zeigt auf eine nicht vorhersehbare Stelle im Speicher !!!
Nicht initialisierte Zeiger sind eine häufige und tückische Fehlerursache, die sich meist durch ”sonderbares” Programmverhalten oder Programmabstürze äußert!

void-Zeiger:
Ein `void`-Zeiger ist auf keinen bestimmten Datentyp gebunden. Er kann eine beliebige Speicherposition adressieren, z.B.: eine float-Variable, eine Position die zu DOS oder zum ROM BIOS gehört.
`void`-Zeiger gestatten es, Daten zu bearbeiten, ohne den Datentyp im voraus zu kennen.

z.B.:

```
#include <stdio.h>

char cBuffer[4]={'A','B','C','D'};
void *pb;

pb=&cBuffer[0];        // zeigt auf 'A'
printf("pb: %c",*pb);  // Fehler !!
```

Der Typ von pb ist nicht bekannt, der Compiler weiß nicht, wie viele Bytes er interpretieren soll und wie. Der Zeiger muß zuerst auf den gewünschten Typ konvertiert werden.

```
printf("pb: %c",*((char*) pb) ); // OK!
// Ergebnis : A
...
printf("pb: %x",*((short*) pb) ); // OK!
// Ergebnis: 0x4241
```

4 Lexikalische Struktur

Wird ein Programm compiliert, ist der erste Schritt die Bearbeitung durch den Preprocessor. Er erzeugt eine Sequenz von **tokens**.

Token sind:

- **Bezeichner** (*identifiers*), z.B.: `sin`
- **Schlüsselwörter** (*keywords*), z.B.: `goto`
- **Operatoren** (*operators*), z.B.: `+`
- **Separatoren** (*separators*), z.B.: `HT` (horiz. Tabulator)
- **Konstanten** (*constants*), z.B.: `3.1415`
- **Zeichenketten** (*string literals*), z.B.: `"Eine Banane\n"`

4.1 Schlüsselwörter (*keywords*)

Einige Namen sind in C mit festen Bedeutungen vorbelegt. Sie können nicht für Namen verwendet werden. Alle Schlüsselwörter werden in Kleinbuchstaben geschrieben!!

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

*)`asm, pascal, fortran`: nur manche Compiler (nicht Standard)
`far, near`: für DOS-memory models (auch `tiny, huge`, usw.)
**)`sizeof` ist ein Operator.
***) War nur in alten Sprachdefinitionen (vor-ANSI) ein reserviertes keyword
In Kursivschrift: zusätzliche Schlüsselwörter in C++

4.2 Operatoren (*operators*)

Operatoren beschreiben, wie Operanden zu einem neuen Wert verknüpft werden sollen.

Ausdruck:→ Operanden und Operatoren

Operanden: Konstante, Variable, Rückgabewerte von Funktionen

Kommen mehrere Operatoren in einem Ausdruck vor, so gilt:

Höhere Priorität (0) vor niederer Priorität (14).

Bei gleicher Priorität:
Auswertung je nach Definition von links nach rechts (lnr) oder von rechts nach links (rnl).

4.2.1 Klammern ( ) (0,lnr)

Operator mit der höchsten Priorität:
→ das Innere von ( ) wird vor dem Äußeren durchgeführt.

4.2.2 Zuweisung (*assignment*) = (13,rnl)

Der Wert rechts von = wird auf den Speicherplatz kopiert, auf den das Symbol links von = verweist.

`a`	=	`b + c + 1`
L-Wert	Zuweisungs Operator	R-Wert
Namen von Variablen oder Objekten, die sich auf Speicheradressen beziehen		Oft temporär; meist das Ergebnis von berechenbaren Ausdrücken

Der Typ des R-Wertes (R-value) wird, wenn möglich, automatisch auf den Typ des L-Werts (L-value) konvertiert.

Da = ein Operator ist , wird ein Ausdruck wie:
`a=b=c=0;` sinnvoll. Wegen der Abarbeitung von (r n l) ist das äquivalent zu :
`a=(b=(c=0));` alle 3 Variablen erhalten den Wert 0.

4.2.3 Arithmetische Operatoren

Sie dienen zur Verknüpfung von numerischen Werten. Ihre Bedeutung, Handhabung und Vorrangregeln sind wie in der Mathematik und anderen Programmiersprachen.

	Operator	Stufe, Richtung
+	Addition	3, lnr
-	Subtraktion	3, lnr
*	Multiplikation	2, lnr
/	Division	2, lnr
%	Modulo (Ganze Zahlen)	2, lnr
-	[unär] negatives Vorzeichen	1,rnl
+	[unär] positives Vorzeichen	1,rnl

Das Ergebnis ist vom selben Variablentyp wie die (einheitlichen) Operanden. Bei uneinheitlichen Operatoren muß eine Typkonversion erfolgen.
z.B.:
`9 / 4` → `2` einheitlich `int`
`11 / 4.0` → `2.75` uneinheitlich, Konversion nach `float`
`8 % 3` → `2` einheitlich `int`

4.2.4 Typumwandlungsoperator (*cast-Operator*) (1,rnl)

Viele Typenumwandlungen erfolgen automatisch. Will man eine bestimmte Umwandlung erzwingen, kann man dazu den cast-Operator verwenden. Er besteht aus der Bezeichnung des gewünschten Typs in runden Klammern und wird der umzuwandelnden Variablen oder dem umzuwandelnden Ausdruck vorangestellt:

(Typ) Ausdruck

Wirkung:
Der Wert des Ausdrucks wird in den angegebenen Typ umgewandelt.

```
int ivar=1;
float fvar;

fvar = ivar;          /* automatische Umwandlung */
fvar = (float) ivar; /* explizit mit dem cast-Operator */
```

Achtung auf Prioritäten:

```
fvar = (float) 3 / 2;  /* gibt 1.5 (= 3.0/2); cast wirkt nur auf 3 */
fvar = (float) (3/2);  /* gibt 1. zuerst Integerdivision, dann cast */
```

Grobe Regel der automatischen Typumwandlung:
Bei einem Ausdruck werden alle Operanden auf den des "kompliziertesten" Typs umgewandelt. Viele Compiler nehmen eine solche Expansion auch dann vor, wenn

sie nicht notwendig ist. Anders sind die Verhältnisse, wenn ein Ausdruck bewertet und dann einer Variablen zugewiesen wird (je nach Compiler werden mitunter Warnungen erzeugt).

Ein Beispiel:

```
int i,k;
float z;
char c;

i = 12;      // Variable i: 12
k = 2 * i;   // Variable k: 24

z=321.78;    // Variable z: 321.78
i=z;         // Variable i: 321 (nicht gerundet!)
c=z;         // Variable c: 65 (letzten (LSB) 8 bit von 321)

i=23;z=7;
z=i/k        // Variable z: 3.
```

4.2.5 Bitoperatoren

In C gibt es insgesamt 6 Operatoren zur bitweisen Verknüpfung.

	Operator	Stufe, Richtung
~	Bitkomplement	1, rnl
<<	Shift links	4, lnr
>>	Shift rechts	4, lnr
&	bitweise UND	7, lnr
^	bitweise exklusiv ODER	8, lnr
|	bitweise ODER	9,rnl

Gilt: int a,b,n;

~a	Negation von a: 0→1 und 1→0
a<<n	Logischer Linksshift von a um n Stellen
a>>n	Arithmetischer Rechtsshift von a um n Stellen
a&b	Bitweise AND-Verknüpfung von a mit b
a|b	Bitweise OR-Verknüpfung von a mit b
a^b	Bitweise XOR-Verknüpfung von a mit b

Häufige Anwendungen bestehen darin, aus einem Bitmuster einen Teil auf 0 oder 1 zu setzen oder einen Teil herauszuschneiden (maskieren).

Einen Teil auf 1 setzen:

```
        x 1010 1010 1010 1010
    Maske 0000 0111 1111 0000
x | Maske 1010 1111 1111 1010
```

Einen Teil auf 0 setzen:

```
         x 1010 1010 1010 1010
     Maske 0000 0111 1111 0000
x & ~Maske 1010 100000000 1010
```

Einen Teil ausschneiden:

```
        x 1010 1010 1010 1010
    Maske 0000 0111 1111 0000
x & Maske 0000 0010 1010 0000
```

Einen Teil ausschneiden und das Komplement bilden:

```
         x 1010 1010 1010 1010
     Maske 0000 0111 1111 0000
~x & Maske 0000 0101 0101 0000
```

4.2.6 Inkrement- und Dekrementoperator

Häufig muß der Wert einer Variablen um 1 erhöht (inkrementiert) oder erniedrigt (dekrementiert) werden. Dafür gibt es in C zwei Operatoren.
Inkrementoperator (1 rnl):

`++ Variable` (Prefix)

`Variable ++` (Postfix)

Dekrementoperator (1 rnl):

`-- Variable` (Prefix)

`Variable --` (Postfix)

Weitere Prioritätsregel: Postfix vor Prefix
Wirkung:

Der Wert der Variablen wird um 1 erhöht (erniedrigt).

Bei der Prefix-Schreibweise wird der Ausdruck nach der Erhöhung (Erniedrigung), bei der Postfix-Schreibweise vorher bewertet.
Diese Operatoren sind nicht auf den Typ int beschränkt!

```
int i,j,k;

i=1;j=1;k=1;

i++;  // i: 2

++i;  // i: 3

i=2;j=3;
k=i+j++;  // k: 5, j: 4, i: 2

i=2;j=3;
k=i+ ++j; // k: 6, j: 4, i: 2
```

`i+++j` ist erlaubt und wird als `(i++)+j` interpretiert.
Wollten Sie das? Ist das für den Programmierpartner auch klar?
Vermeiden Sie undurchsichtige Konstruktionen!

4.2.7 Logische Operatoren

Es gibt 6 Vergleichsoperatoren.

Trifft ein Vergleich zu, hat der Vergleichsausdruck einen Wert ungleich Null (true) und sonst 0 (false). Welchen Wert true hat, ist undefiniert und compilerabhängig.
Ausprobieren: `printf("%d",2==2);` (oft 1).

	Operator	Stufe, Richtung
<	Kleiner als	5, lnr
>	Größer als	5, lnr
<=	Kleiner gleich	5, lnr
>=	Größer gleich	5, lnr
==	Gleich	6, lnr
!=	Ungleich	6, lnr
&&	Logisch UND	10, lnr
||	Logisch ODER	11, lnr
!	nicht	1, rnl

Die logischen Verknüpfungen `&&` bzw. `||` dürfen auf keinen Fall mit den Bitoperatoren `&` bzw. `|` verwechselt werden. Der logische Operator == darf nicht mit dem Zuweisungsoperator = verwechselt werden.
Es wird von (lnr) ausgewertet und nur solange, bis das Ergebnis feststeht.

```
(i!=0) && (n/i<20) // Keine Division /0
```

4.2.8 Zusammengesetzter Zuweisungsoperator: (13 lnr)

Oft werden Ausdrücke der Form: `Summe = Summe + YO;` verwendet.
C gestattet dabei eine kürzere und effizientere Schreibweise:

```
Summe += YO;
```

Folgende Formulierungen sind gleichbedeutend:

```
    x += y;
    x  = x + y;
// Bzw.
    x *= y+z;
    x  = x * (y + z);
```

Zusammengesetzter Zuweisungsoperator: **Variable op= Ausdruck**

Wirkung: **op=** ist gleichbedeutend mit **Variable = Variable op Ausdruck**

op kann jeder der folgenden Operatoren sein:

```
+ - * / % << >> ^ | &      /* auch bitweise Operatoren ! */
```

Beispiel:

```
 count &= ~1;
 count %= 16;
```

4.2.9 Bedingte Bewertung (12, rnl):

Der `?:` Operator wirkt ähnlich wie eine verkürzte `if-else` Konstruktion.

Bedingung ? **Ausdruck1** : **Ausdruck2**

Wirkung:
Ist *Bedingung* true, so ist der Wert des gesamten Ausdrucks jener von *Ausdruck1*, sonst der von *Ausdruck2*.

Beispiele:

```
(i == 7) ? printf("i ist 7):printf("i ist nicht 7");

(Index==0) ? Summe=0 : Summe += yWert;

y=(x==0 ? 1 : sin(x)/x);
```

Die beiden Anweisungen könnten auch durch eine if-Anweisung ersetzt werden:

```
if(i==7) printf("i ist 7");
else printf("i ist nicht 7");

if(Index==0)Summe=0;
else Summe += yWert;
```

4.2.10 Der sizeof-Operator (1 rnl)

sizeof-Operator: `sizeof(`Argument`)`

Wirkung: Es wird der Speicherplatz von Argument in Bytes ermittelt.

Argument: Typname, Variable oder Konstante.

Beispiele (compilerabhängig):

Ausdruck	Wert
`sizeof(3.21);`	8
`long i; sizeof (i);`	4
`sizeof (short);`	2

4.3 Anweisungen (*statements*)

4.3.1 Ausdrucksanweisung

Aus einem Ausdruck wird durch den abschließenden Strichpunkt ; eine Anweisung, d.h. der Ausdruck wird bewertet (berechnet), aber nirgendwohin zugewiesen. Dies ist beispielsweise bei alleinstehenden Funktionsaufrufen der Fall, wie `printf("Hello world");`. Hier wird ein Ausdruck bewertet, der Wert (die Funktion `printf()` liefert die Anzahl der ausgegebenen Zeichen zurück) wird aber oft nicht weiter verwendet. Der Funktionsaufruf (=Ausdrucksanweisung) ist natürlich trotzdem sinnvoll.
Die Ausdrucksanweisung sin(3.141/2.); ist ebenfalls legal, aber sinnlos.
Ebenso sind beispielsweise 1; oder 1/1; legale, sinnlose Anweisungen.

4.3.2 Verbundanweisung

Ermöglicht mehrere Anweisungen zu einer Einheit zusammenzufassen. Diese Anweisungen werden in `{ }` eingeschlossen. Die Verbundanweisung endet nicht mit `;`.

z.B.:

```
if(Bedingung) { i = 1; j = 2; k = 3; l = 5; }
```

Bedingung kann jeder berechenbare Ausdruck sein, der als falsch oder wahr interpretiert wird, je nachdem ob er Null ist oder nicht.

4.3.3 if-Anweisung

`if ( Bedingung ) Anweisung1;`

oder `if ( Bedingung ) Anweisung1; else Anweisung2;`

oder
```
if ( Bedingung1 ) Anweisung1;
     else if( Bedingung2 )Anweisung2;
          else Anweisung3;  /* else gehört zum naeheren if! */
```

oder
```
if ( Bedingung1 ) Anweisung1;
     else
     {if ( Bedingung2 )Anweisung2; else Anweisung3; /* Wie zuvor */
     }
```

Zur klareren Lesbarkeit Klammern setzen!

Beispiel:

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

/* Program yesno0.cpp */
int main()
{int cAnswer;

 printf(" Type Y for yes, N for no:");
 cAnswer=getchar();

 if(cAnswer=='Y' || cAnswer=='y')exit(1);
 else exit(0);
}
```

Das Programm gibt an das Betriebssystem entweder 1 oder 0 zurück, je nachdem ob Y(y) oder irgendeine andere Taste gedrückt wurde.

4.3.4 switch-Anweisung

Für die Auswahl aus mehreren Anweisungen gibt es die `switch`-Anweisung:

```
if(bFall1) { ... } else if(bFall2) { ... } else if(bFall3) { ... } else usw.
```

Solche Programmteile sind zwar zielführend, können jedoch sehr unübersichtlich werden.

Oft besser mit switch-Anweisung:

```
switch(Ausdruck1)
 {case konst_Ausdruck1:
   ...Anweisungen; break;

  case konst_Ausdruck2:
  ... Anweisungen; break;

  ...

  default: Anweisungen;
}
```

Wirkung:
Es wird `Ausdruck1` bewertet. Kommt dieser Wert unter den Konstanten nach `case` vor, werden die nach `:` folgenden Anweisungen ausgeführt. Kommt er nicht unter den Konstanten nach `case` vor, so werden die Anweisungen nach `default` ausgeführt.
Fehlt `default` so hat die `switch`-Anweisung für nicht vorkommende Ausdrücke keine Wirkung. Fehlt ein `break`, werden alle Anweisungen bis zum nächsten `break` oder bis zum Ende der `switch`-Anweisung durchgeführt.
Hinweis:
Die `switch`-Anweisung endet mit der abschließenden Klammer `}`! Es wird also ohne `break;` der Code aller nachfolgenden `case`-Möglichkeiten auch ausgeführt, auch wenn dort die `case`-Bedingung nicht mehr erfüllt ist. Eigentlich ist `switch` eine Art von `goto` zu einem variablen Sprungziel.

Die Konstanten nach `case` dürfen nur einfache `int` Datentypen (keine Arrays oder Strings) sein.

```
if(Bedingung) Anweisung1; else Anweisung2;
```

ist gleichbedeutend mit:

```
switch(Bedingung)
  { case 0: Anweisung2;
            break; /* Null entspricht false */
   default: Anweisung1;
  }
```

Hier ist die `switch`-Konstruktion aber keine empfehlenswerte Alternative, weil die Logik nicht einfach durchschaubar ist.

Beispiel:

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

int main()

{ int cAnswer;

  printf("\nMenu: Add Delete Quit: ");

  cAnswer=getchar();

  printf("\n");

  switch(toupper(cAnswer))
        {case 'A': printf("Add selected");
                   break;
         case 'D': printf("Delete selected");
                   break;
         case 'Q': printf("Quit selected");
                   exit(0);
          default: printf("Illegal selection");
        }/*switch end*/
}/*main end*/
```

4.3.5 Wiederholungsanweisungen (Schleifen)

C kennt folgende Schleifenkonstruktionen:

`while` (Bedingung) Schleifenkörper;

`do` Schleifenkörper;
`while` (Bedingung);

`for`(Anweisung(en); Bedingung; Anweisung(en)) Schleifenkörper;

Die `for`-Schleife ist eine sehr universelle Konstruktion, die der `do`-Schleife in Fortran ähnelt, aber viel flexibler ist.

Die beiden anderen unterscheiden sich durch die Anordnung der Bedingung:

- Vor der Schleife
  Soll der Schleifenkörper überhaupt einmal durchlaufen werden? (`while`-Schleife)
- Am Ende der Schleife
  Soll die Schleife nach dem jedenfalls erfolgten ersten Durchlauf weiter fortgesetzt werden? (`do-while`-Schleife)

while-Anweisung mit Abbruchbedingung am Anfang:

`while (` Bedingung `)` Schleifenkörper;

Wirkung: *Bedingung* wird bewertet:

true: Der Schleifenkörper wird ausgeführt und die Bedingung neu geprüft.

false: Die Schleife wird abgebrochen.

Ist Ausdruck bereits zu Beginn false so wird die Schleife gar nicht durchlaufen. Der Schleifenkörper kann eine einzelne Anweisung oder eine mit `{ }` zusammengefaßte Verbundanweisung sein.

Beispiel: Herunterzählen von 10 mit while-Schleifen:

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

main()
{ //-1---
 print("\n/*1*/");
 int iCount=10;

 while(iCount>0)
  {printf("%3d",iCount);
   --iCount;
  }
//-2---
 print("\n/*2*/");
 iCount=10;

 while(iCount>0)
  {printf("%3d",iCount);
   iCount--;
  }
// -3---
 print("\n/*3*/");
```

```
 iCount=10;

 while(iCount) printf("%3d",iCount--);
//-4---
 print("\n/*4*/");
 iCount=10;

 while(iCount) printf("%3d",--iCount);
//-5---
 print("\n/*5*/");
 iCount=10;

   while(iCount--) printf("%3d",iCount);
//-6---
 print("\n/*6*/");
 iCount=10;

 while(--iCount) printf("%3d",iCount);
}
```

Ergebnis:

```
/*1*/ 10 9 8 7 6 5 4 3 2 1
/*2*/ 10 9 8 7 6 5 4 3 2 1
/*3*/ 10 9 8 7 6 5 4 3 2 1
/*4*/ 9 8 7 6 5 4 3 2 1 0
/*5*/ 9 8 7 6 5 4 3 2 1 0
/*6*/ 9 8 7 6 5 4 3 2 1
```

do-while-Schleife mit Abbruch am Ende:

`do` Schleifenkörper `; while (` Ausdruck `);`

Wirkung: Anweisung(en) des Schleifenkörpers werden ausgeführt und danach wird Ausdruck bewertet:

true: Die Anweisung wird erneut ausgeführt und Ausdruck neu bewertet.

false: Die Schleife wird abgebrochen.

Die Schleife wird in jedem Fall einmal durchlaufen. Der Schleifenkörper kann eine einzelne Anweisung oder eine mit `{ }` zusammengefaßte Verbundanweisung sein.

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

main()
{int iEnd, iCount;

 printf("iEnd = ");
 scanf("%d",&iEnd);
 iCount=1;

 do
  {printf("%3d",iCount);
   ++iCount;
  }
 while(iCount<=iEnd);printf("\n");

 exit(0);
}
```

for-Anweisung:

`for` (Anweisung1; Ausdruck2; Anweisung3) Schleifenkörper;

Wirkung:

Anweisung1 : Beliebige Initialisierungsanweisung(en)

Ausdruck2 : Bedingung(en) für das Weiterlaufen der Schleife

Anweisung3 : Beliebige Anweisung(en) nach Beendigung jedes Schleifendurchlaufs oft Inkrement einer oder mehrerer Schleifenvariablen

Schleifenkörper : Einzelne Anweisung oder Verbundanweisung

Fehlen *Anweisung1* und/oder *Anweisung3*, so gelten sie als nicht vorhanden. Fehlt *Ausdruck2* so gilt er als true. In *Anweisung1* initialisierte Variablen können im *Schleifenkörper* beliebig verändert werden.
Die Ausdrücke im Schleifenkopf sind grundsätzlich beliebiger Natur. *Anweisung1* und *Anweisung3* können auch aus mehreren Anweisungen (ohne ;) bestehen, die durch Beistriche getrennt werden. Im Prinzip kann man auf diese Art einen ganzen *Schleifenkörper* in den Schleifenkopf einbauen, allerdings auf Kosten der Lesbarkeit.

```
for (;;) ANWEISUNG; /*Ist eine Endlos-Schleife! */

for(;;); /*Endlose Warteschleife */
```

Beispiel:

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

main()
{int iEnd, iCount;

 printf("iEnd = ");
 scanf("%d",&iEnd);

 for(iCount=1;iCount<=iEnd;iCount++)
    {printf("%3d",iCount);
    }

 printf("\n");
 exit(0);
}
```

Durch Komma getrennte Schleifenelemente können mehr als eine Variable in der Schleife steuern. Im Prinzip sind sie frei programmierbare Anweisungen.
z.B.:

```
int i,j;

for(i=0, j=9; i<=9 && j>=0; i++, j--)
   {printf("\ni=%d j=%d\n",i,j);
   }
```

oder

```
int i,j;

for(i=0, j=9; i<=9 && j>=0; printf("\ni=%d j=%d",i++,j--));
```

Hier steht `printf()` im Schleifenkopf. Beachten Sie, daß der `printf()` Befehl nicht mit einem Strichpunkt endet. Der Strichpunkt am Zeilenende ist der Schleifenkörper (leerer Befehl).

Welche Schleifenkonstruktion ist die beste?
Man wählt immer die Konstruktion, die der Problemstellung am anschaulichsten entspricht. Aus der Sicht der Ausführungsgeschwindigkeit gibt es keinen Unterschied. Kunstvolle Einbettungen komplexer Anweisungen in den for-Schleifenkopf, die nichts mit der Ablaufkontrolle zu tun haben, sind schlechter Stil.

4.3.6 Sprunganweisungen

goto-Anweisung:

```
goto Label;
```

Wirkung: Es wird die mit Label gekennzeichnete Anweisung ausgeführt. Die Reichweite ist auf eine Funktion beschränkt.

Label: Ist ein Name und wird durch : von der Anweisung getrennt.

```
if ( Bedingung ) goto weiter;
weiter: Anweisung;
```

Nach einem Label muß eine Anweisung folgen. Soll ein Label die Bearbeitung eines Blocks beenden (aus dem Block aussteigen), so muß eine Leeranweisung folgen:

```
{...
 goto ENDE;
 ...
ENDE:
;
}
```

Die `goto`-Anweisung gilt als unelegant, weil die Logik des Programmablaufs (Programmausdruck am Papier) zerrissen wird. Sie sind aber jedenfalls nützlich für den Ausstieg aus dem Inneren von geschachtelten Schleifen oder geschachtelten `if`-Anweisungen.

break-Anweisung:

```
break;
```

Wirkung: Eine Schleife (`for`, `while` oder `do-while`) oder eine `switch`-Anweisung wird bedingungslos abgebrochen.

Hinweis: Bei verschachtelten Schleifen bedeutet `break` nur den Abbruch der aktuellen Schleife. Herausspringen aus dem Inneren von verschachtelten Schleifen ist nur mit `goto` möglich.

continue-Anweisung:

```
continue;
```

Wirkung: Der Rest des aktuellen Schleifendurchlaufs wird übersprungen und die Fortsetzung der Schleife erneut geprüft.

```
for (..;...;...)

for(...;...;...)

{if (bedingung1) continue

if (bedingung2) break;

...
}
```

5 I/O-System in C

In C werden die Datenkanäle als Ströme (streams) bezeichnet. Jedes C-Programm ist standardmäßig durch 3 Streams mit seiner ”Umwelt” verbunden:

stdin Standardeingabe (Tastatur)

stdout Standardausgabe (Bildschirm)

stderr Standard-Fehlerausgabe (Bildschirm)

Ein elementares Bedürfnis besteht darin, ASCII-Zeichen am Bildschirm auszugeben. Für formatierte Ausgabe über stdout (Bildschirm) steht die Funktion `printf()` zur Verfügung:

5.1 Formatierte Ausgabe

Funktion: `int printf("Formatstring", Argumentliste)`

Prototyp (Deklaration): `<stdio.h>`

Parameter: Formatstring: Beliebige Zeichenketten und eingebettete Formatspezifikationen für die Ausgabe von Variablen. Die Formatspezifikationen werden mit % eingeleitet.

Argumentliste: Liste von Variablen, deren Wert(e) ausgegeben werden soll(en)

Trennzeichen: Beistriche. Jeder Variablen muß genau eine mit % eingeleitete Formatspezifikation entsprechen.

Ausnahmen:
1. Die %n-Spezifikation, bei der eine Variable einen Wert aufnimmt statt ausgibt.
2. Die variable Feldlängen-Vereinbarung, für die eine oder zwei zusätzliche Variablen benötigt werden.

Wirkung: Der Text des Formatstrings wird ausgegeben und die Werte der Variablen in der Argumentliste den Formatspezifikationen entsprechend eingebettet. Die Variablenliste kann leer sein. (Ausnahmen: wie oben angemerkt.)

Funktionswert (Rückgabewert): Anzahl der ausgegebenen Zeichen, vom Typ int.

Beispiel:

```
  int iWert = 2049, iL;
float     z = 456.78

iL=printf("iWert: %d %4x %4X %o \n", iWert,iWert,iWert,iWert);

printf("Länge: %d\n",iL);

printf("z: %f %e %7.4f\n",z,z,z);
```

Formatstring:	Besteht aus normalen Zeichen und Formatangaben.
Normale Zeichen:	Werden direkt ausgegeben
Formatangaben:	`%Breite.Präzision Typ`
Breite:	(Mindest-) Gesamtlänge des Felds für die Variable.
Präzision:	
bei Gleitkommazahlen:	Anzahl der Nachkommastellen
bei Zeichenketten:	Maximalzahl der Schreibstellen. Siehe auch die Beschreibung der Bibliotheksfunktionen im Anhang.

Vor der Breite/Präzision-Spezifikation kann noch eines der folgenden Zeichen eingefügt werden:

Leerzeichen:	Fügt 1 Leerstelle vor positiven Zahlen ein, Minuszeichen bei negativen Zahlen
+ (Pluszeichen):	Vorzeichenangabe vor jeder Zahl
- (Minuszeichen):	Linksbündige Anordnung
0 (Null):	Bei Zahlenwerten: füllt mit führenden Nullen
# (number):	je nach Format wird eingefügt: führende 0 bei Oktalzahlen, führendes 0x oder 0X bei Hexadezimalzahlen jedenfalls Dezimalpunkt bei e,E,f,g,G - Formaten Nullen am Ende bleiben bei g und G Formaten

5.2 Formatiertes Einlesen

Formatierten Einlesen (Tastatur) erfolgt mit der Funktion `scanf()`, die mit ähnlichen Parametern wie `printf`, jedoch mit Zeigern auf die Variablen statt der Variablennamen arbeitet. Es handelt sich dabei um die Angabe der Adresse einer Variablen. Der Adreßoperator ist ein dem Variablennamen vorangestelltes & .

Funktion: `int scanf()`

Prototyp: `<stdio.h>`

Parameter: Formatstring und Zeiger auf Variablen

Wirkung: Es werden die mit dem Formatstring korrespondierenden Werte auf die Speicherplätze der Variablen übertragen.

Rückgabewert: Die Anzahl der erfolgreich konvertierten Eingabe-Werte oder EOF, wenn ohne irgendeine erfolgreiche Konversion das Ende der Eingabe (<CTRL> Z in DOS oder <CTRL> D in UNIX) erreicht ist oder ein Fehler aufgetreten ist.

z.B.

```
  int iWert;
float z;

scanf("%d,%f", &iWert,&z);
```

Anmerkungen:

1. Die Eingabedaten werden bei allen Formatspezifikationen außer %c (character) folgendermaßen interpretiert (im folgenden Absatz symbolisiert WS eine Folge von white space-Zeichen, die eckigen Klammern dienen hier nur der klareren Feldabgrenzung):

   ```
   [WS][Eingabefeld 1][WS][Eingabefeld 2][WS] ...
   ```

   - WS begrenzt die Eingabefelder und wird ansonsten ignoriert, also ohne Interpretation überlesen. Die Leseroutine sucht jeweils nach dem nächsten Eingabefeld, das aus nicht-WS besteht und versucht, dieses entsprechend der zugeordneten Formatspezifikation zu interpretieren. Beachten Sie, daß ein Zeilenende als WS angesehen wird und die Eingabe nicht beendet (außer in der Funktion gets()).
   - Jedes Eingabefeld wird von links nach rechts in Segmenten der angegebenen Breite-Spezifikationen interpretiert, die Angabe der Präzision wird ignoriert und sollte unterbleiben.
     Beispielsweise liefert die Eingabe:
     [WS][1234567...] mit "%1i%2i%3i"-Format die Werte 1, 23, und 456.
   - Wird vor dem Typ das Zeichen * angegeben (z.B. %*s), wird das entsprechende Feld übersprungen und es erfolgt keine Zuweisung auf eine Variable.
   - Bei Verwendung des %s -Formats wird die eingegebene Zeichenkette automatisch durch Anhängen von '\0' abgeschlossen (ausreichend dimensionieren!).
   - Andere Zeichen als die mit % eingeleiteten Spezifikationen im Formatstring müssen mit den Eingabedaten korrespondieren. Textkomponenten im Formatstring müssen auch in den Eingabedaten an entsprechender Stelle vorhanden sein.
     Beispielsweise liefert die Eingabe
     [WS][abcdef] mit "abc%2s" den String [de\0].

- Zum Testen, ob alle Konversionen erfolgt sind, Rückgabewert abfragen!

2. %c Format: Liest jedes Zeichen (auch white-space). Es werden soviele Zeichen gelesen, wie die Breite -Spezifikation angibt. Default-Wert ist 1 Zeichen. Will man das nächste non-WS Zeichen einlesen, ist die Verwendung von `%1s` (also ein aus einem Zeichen bestehender String) zu empfehlen (`'\0'` wird angehängt!).

5.3 Einlesen eines einzelnen Zeichens

Einzelne Zeichen können mit der Funktion `getchar()` eingegeben werden. Die Funktion wartet normalerweise auf RETURN der Tastatur. Die genaue Wirkung hängt allerdings unter Unix (Linux) von verschiedenen Einstellungen des I/O Subsystems ab.

Funktion: `int getchar()`

Prototyp: `<stdio.h>`

Parameter: keine

Wirkung: Es wird ein Zeichen über stdin (Tastatur) gelesen. `getchar()` wartet auf `<RETURN>` der Tastatur

Funktionswert: ASCII-Wert des gelesenen Zeichens

Beispiel:

```
int cAny;

/* . . .*/

printf("Drücken Sie eine Taste und <CR>:");

cAny=getchar();

printf("\nIhre Taste war: %c ASCII-Wert:%x \n",(char)cAny,cAny);
```

Anmerkungen:

1. Die Konstruktion `(char) cAny` ist ein cast (Typenumwandlung) der `int`-Variablen `cAny` in den Typ `char`. Sollte bei den meisten Compilern nicht erforderlich sein.

2. Gelegentlich verbleibt im Buffer des Unix I/O-subsystems das Zeichen `'\n'` (ASCII 10) und erscheint dann unerwartet beim folgenden Aufruf von `getchar()`.
   Lösung: übertragenes Zeichen prüfen und gegebenenfalls ignorieren.

5.4 Formatierte Ein/Ausgabe auf Dateien

Für die formatierte Ein- und Ausgabe auf eine Datei werden in C folgende Funktionen verwendet:

```
//Ausgabe
int fprintf(FILE *stream, const char *format, ...);
//Eingabe
int fscanf(FILE *stream, const char *format, ...);
int fgetc(FILE *stream);
char *fgets(char *s, int size, FILE *stream);
```

Sie arbeiten analog, zu den vorher erklärten Funktionen. Nur weisen alle den zusätzlichen Parameter, `FILE * f` (*file descriptor*) auf, einen Zeiger auf eine Struktur mit dem Namen `FILE`, die den Ein-/Ausgabe-Strom *(stream)* definiert.
Dieser 'stream' muß erst durch die Funktion:

```
FILE *fopen(const char *path, const char *mode);
```

"geöffnet" werden.
Kann eine Datei erfolgreich geöffnet werden, liefert `fopen` einen gültigen Zeiger auf ein Objekt vom Typ `FILE`, dieser Zeiger wird einfach an die folgenden Ein-/Ausgabe-Funktionen (`fprintf, fscanf, fgets, ...`) weitergereicht, die dann die entsprechende Lese- oder Schreiboperation durchführen. Um den eigentlichen Mechanismus hinter dieser '`struct FILE`' braucht sich der "normale" C-Programmierer keine Gedanken machen, das Weiterreichen des Parameters an die jeweilige Funktion ist ausreichend.
Nach dem alle Ein- bzw. Ausgabe-Funktionen auf die entsprechende Datei fertiggestellte wurden, ist abschließend noch die Funktion:

```
int fclose(FILE *f);
```

aufzurufen, erst dann werden beim Schreiben alle Änderungen an der entsprechenden Datei aktiv.

Beispiel: Zeilenweises einlesen einer Datei mit Ausgabe auf den Bildschirm:

```
#include <stdio.h>
#define FileName "/etc/passwd"
#define MAX_LINELENGTH 1024

main()
{ char buffer[ MAX_LINELENGTH+1 ];
  FILE *f=fopen(FileName,"r");
 if(f==NULL)
   {printf("Cannot open %s\n",FileName);
    exit(1);
   }
 while(fgets(buffer, MAX_LINELENGTH, f))
      printf("%s",buffer);
 fclose(f);
 exit(0);
}
```

Beispiel: Kopieren einer Textdatei:

```
#include <stdio.h>
#define SrcFile "/etc/passwd"
#define DstFile "passwd.copy"
#define MAX_LINELENGTH 1024

main()
{ char buffer[ MAX_LINELENGTH+1 ];
  FILE *s,*d;

 s=fopen(SrcFile,"r");
 if(!s)
   {printf("Cannot open %s\n",SrcFile);
    perror("Reason");
    exit(1);
   }
 d=fopen(DstFile,"w");
 if(!d)
   {printf("Cannot open %s\n",DstFile);
    perror("Reason");
    exit(1);
   }
 while(fgets(buffer, MAX_LINELENGTH, f))
      fprintf(d,"%s",buffer);
 fclose(s);
 fclose(d);
 exit(0);}
```

Funktion: `FILE *fopen(const char *path, const char *mode);`

Prototyp: `<stdio.h>`

Parameter: Dateiname und Modus
Dateiname: String, der einen gültigen Dateinamen angibt. 'Gültig' heißt, daß der Benutzername unter dem das Programm abläuft, die angegebene Datei, je nach Modus, lesen oder schreiben darf.
Modus: Ein String mit folgender Bedeutung:

`"r"`	Die Datei wird zum Lesen geöffnet. Beginn am Anfang der Datei.
`"w"`	Die Datei wird zum Schreiben geöffnet. Beginn am Anfang der Datei. Falls die Datei bereits existiert, geht der bestehende Inhalt verloren
`"a"`	Die Datei wird zum Anhängen geöffnet. Die Datei wird erstellt falls es sie nicht gibt, ansonsten wird an die bestehende Datei angefügt.

Es gibt noch weitere Möglichkeiten für den 'Mode-String'; siehe `man fopen`.

Wirkung: Öffnet die Datei mit dem angegebenen Namen und Modus und weist den entsprechenden Datenstrom zu.

Rückgabewert: Beim erfolgreichen Öffnen wird ein Zeiger auf eine gültiges FILE-Objekt zurückgegeben. Anderenfalls wird der NULL-Zeiger zurückgegeben und die globale Variable `errno` wird entsprechend gesetzt. Mit der Funktion `perror`, kann im Fehlerfall ein zu `errno` entsprechender Text ausgegeben werden.

Funktion: `int fclose(FILE * stream);`

Prototyp: `<stdio.h>`

Parameter: Zeiger auf ein zuvor geöffnetes FILE-Objekt.

Wirkung: Beendet den Datenstrom zur zugewiesenen Datei. Bei 'gebufferten' Schreibvorgängen wird der entsprechende Datenbuffer gelehrt.

Rückgabewert: Beim erfolgreichen Beenden wird 0 zurückgegeben. Anderenfalls EOF und die globale Variable `errno` wird entsprechend gesetzt. Wird auf einen bereits geschlossen 'stream' zugegriffen, so ist das Ergebnis nicht definiert (auch mit `fclose`).

Funktion: `int fprintf(FILE *stream, const char *format, ...);` und
`int fscanf(FILE *stream, const char *format, ...);`

Prototyp: `<stdio.h>`

Parameter, Rückgabewert, Wirkung Wie bei den entsprechenden, bereits besprochenen Funktionen `printf()` und `scanf()`, nur daß die entsprechende Aktion auf die, durch den zusätzlichen Parameter spezifizierte Datei, erfolgt.

Funktion: `int fgetc(FILE *stream);`

Prototyp: `<stdio.h>`

Wirkung: Liest das nächste Zeichen von `stream` und gibt es als 'cast' von einem '`unsigned char`' zu '`int`' zurück. Im Fehlerfall oder wenn das ende der Datei erreicht ist wird EOF zurückgegeben.

Funktion: `char *fgets(char *s, int size, FILE *stream);`

Prototyp: `<stdio.h>`

Wirkung: Liest maximal `size -1` Zeichen von `stream` und speichert sie im Buffer auf den `s` zeigt. Der Einlesevorgang endet nach EOF (Dateiende) oder nach einem Zeilenvorschub (`\n`). Der Zeilenvorschub wird ebenfalls im Buffer gespeichert, ebenso wird ein `'\0'` als letztes Zeichen im Buffer abgelegt.

Details siehe auf den entsprechenden `man`-Seiten.

In jedem C-Programm stehen die 3 'file descriptoren' `stdin`, `stdout` und `stderr` für Schreib- bzw. Leseoperationen zur Verfügung. Diese Datenströme brauchen weder geöffnet noch geschlossen werden.
Ein `printf("...",...)` entspricht `fprintf(stdout,"...",...)`, sowie `scanf("...",...)` entspricht `fscanf(stdin,"...",...)`.

Fehlermeldungen sollten immer auf `stderr` ausgegeben werden:
`fprintf(stderr,"Die Datei %s kann nicht geoeffnet werden\n",fname);`
sie erscheint standardmäßig, d.h. wenn keine Umleitung erfolgt ist, am Bildschirm. Der Benutzer des Programms kann dann aber, wie unter Linux üblich die Fehlermeldungen in eine Datei umleiten:

```
programname 2> fehler.txt
```

oder wenn sie nicht benötigt und angezeigt werden sollen:

```
programname 2> /dev/null
```

5.5 Nichtformatierte Ein/Ausgabe, Ein/Ausgabe auf Dateien

Siehe Standard-Bibliotheksfunktionen.

6 Datenstrukturen und Zeiger

6.1 Datenstrukturen (*structures and unions*)

Eine Gruppe von Variablen kann mit dem reservierten Wort struct zu einer Struktur zusammengefaßt werden. Dies erzeugt einen neuen Datentyp, der in einem zusammenhängenden Block gespeichert werden kann.

Beispiel: Periodensystem der chemischen Elemente
Es sollen folgende Daten gespeichert werden:

- Elementname `char Ele[2]`
- Atomgewicht `float Atwt`
- Ordnungszahl `int Z`

```
struct Periodensystem
{ char Ele[3];
 float Atwt;
   int Z;
} Element;
```

Diese Struktur hat den Namen (*structure tag*) `Periodensystem`. Die drei Strukturelemente (*structure members*) sind `Ele`, `Atwt` und `Z`. Damit ist die Struktur definiert. `Element` ist nun ein Objekt im Speicher, das (vermutlich) 12 aufeinanderfolgende Bytes belegt, die als `char`, `float` und `int` interpretiert werden (und möglicherweise Füllbytes enthalten).
Man kann den Namen (*tag*) weglassen, wenn das Objekt (hier: `Element`) angegeben ist:

```
struct
{ char Ele[3];
 float Atwt;
   int Z;
}Element;
```

Andererseits kann man die Struktur auch unabhängig von einem Objekt deklarieren, natürlich nur mit einem Namen:

```
struct Periodensystem
{ char Ele[3];
 float Atwt;
   int Z;
};
```

und dann ein oder mehrere Objekt(e) durch Hinweis auf den Namen deklarieren:

```
struct Periodensystem Element1, Element2;
```

oder ein ganzes Feld von Objekten:

```
struct Periodensystem Elemente[92];
```

Die einzelnen Elemente spricht man folgendermaßen an:

```
  int iz;
float wt;
 char c1,c2;

iz = Element1.Z;
wt = Element1.Atwt;

Elemente[29].Z = iz;

c1 = Elemente[29].Ele[1];
c2 = Element1.Ele[0];
```

Ein Struktur-Objekt kann gleichzeitig mit der Deklaration definiert und initialisiert werden:

```
struct Periodensystem
{ char Ele[3];
 float Atwt;
   int Z;
} Cu = {"Cu", 63.55, 29}; /* Deklaration und Defininition */
```

Das funktioniert auch für Felder von Objekten, wird allerdings leicht unübersichtlich. Oft helfen Klammerstrukturen zur Verdeutlichung.

Anmerkung:
Der sizeof-Operator kann auch auf Strukturen oder deren Variablen angewendet werden.

```
i = sizeof(Cu); /* liefert unter Linux/cc den Wert 12 */
```

6.1.1 Verschachtelte Strukturen

Nun sollen im Periodensystem die Bindungsenergien der Elektronen für jedes Element eingebaut werden. Man kann die Daten direkt eingeben:

```
struct Periodensystem
{ char Ele[3];
 float Atwt;
   int Z;

 float K_Energie;
 float L1_Energie;
 float L2_Energie;
 --- usw ---
}Element;
```

oder eine neue Struktur aufbauen:

```
struct Bindungsenergien
{float K_Energie;
 float L1_Energie;
 float L2_Energie;
--- usw ---
};
```

und in der ersten Struktur verwenden:

```
struct Periodensystem
{  char Ele[3];
  float Atwt;
    int Z;

 struct Bindungsenergien EBindung;

}Element;

float K_eng = Element.EBindung.K_Energie;
```

6.1.2 typedef in Strukturen

6.1.3 `typedef` in Strukturen

```
typedef struct Periodensystem
{ char Ele[3];
 float Atwt;
   int Z;
}Ele;

Ele pElement;
```

6.1.4 Unions

`union` erlaubt es, Speicherplatz unter verschiedenen Namen anzusprechen und dort Daten unterschiedlichen Typs abzuspeichern (ähnlich wie `equivalence()` in Fortran)

z.B.

```
union example { float aFloat; long li; } ex;
```

Die Syntax ist analog zu der in Strukturen.
Im Beispiel der `union example` verweisen `ex.aFloat` und `ex.li` auf dieselbe Speicheradresse. Wenn `ex.aFloat` ein Wert zugewiesen wird, wird der Speicherplatz im float-Format beschrieben und sollte im gleichen Format wieder gelesen werden. Verwendung von `ex.li` würde den Speicherplatz als `int` interpretieren, was vermutlich unerwünscht ist. Es erfolgt keine automatische Konversion. (Strenge Regeln in C++).

6.2 Zeiger (*pointer*) und Felder (*arrays*)

6.2.1 Pointer und eindimensionale Arrays

Wird ein Array vereinbart, z.B.

```
int iA[10];
```

so ist per definitionem `iA` ein Pointer auf das erste Feldelement `iA` $\equiv$ `&iA[0]`.

Das folgende Feldelement erhält man durch Fortschreiten um soviele Bytes, wie die Datenlänge der Feldelemente angibt. Das geschieht automatisch durch inkrementieren der Zeigervariablen:

`iA`	zeigt auf das Feldelement	`iA[0]`
`iA+1`	zeigt auf das Feldelement	`iA[1]`
`iA+2`	zeigt auf das Feldelement	`iA[2]`, usw.

Es sei int *p = iA;

`iA`	`&iA[0]`	`p`		Adresse Array-Anfang
`iA+i`	`&iA[i]`	`p+i`		Adresse von `iA[i]`
`*(iA+i)`	`iA[i]`	`*(p+i)`	`p[i]`	Wert von `iA[i]`

Da `iA+i` dasselbe bedeutet wie `&iA[i]` und `iA+i` dasselbe ist wie `i+iA`, gilt immer auch `iA[i]` $\equiv$ `i[iA]` (ausprobieren!)

Nochmals: In C werden Indexgrenzen grundsätzlich nicht überwacht!

Vorteil: kürzere Rechenzeit

Nachteil: geringere Programmsicherheit

Was geschieht, wenn der Index außerhalb des vereinbarten Bereichs zu liegen kommt?

z.B.:

```
int iA[10],i;

for(i=0;i<20;i++)iA[i]=2*i;
```

Die obige Schleife wird 20 mal durchlaufen. Für `i>9` wird der hinter `iA[9]` liegende Speicherplatz benutzt; was auch immer dort steht wird überschrieben.

6.2.2 Pointer und zweidimensionale Arrays

N ehmen wir an, das folgende zweidimensionale Feld ist vereinbart:

```
float m[4][5];
```

Damit kann beispielsweise folgende Datenmatrix aufgebaut werden:
1.Index = Zeile , 2.Index = Spalte

	`m[0][0]`				`m[0][4]`
`m[0]`	2.34	3.14	1.59	6.28	1.
`m[1]`	-1.23	1.9E23	44.12	6.7	0.815
`m[2]`	9.0	0	0.0123	1.2e-7	9.11
`m[3]`	66.11	47.11	0.11	3.19	13.1
			`m[3][2]`		

Die Interpretation der Feldvariablen als Zeiger ist hier:

Man hat ein Feld von 4 Zeigern (`m[0]`, `m[1]`, `m[2]`, `m[3]`). Jeder zeigt auf ein Feld, das die Werte von 5 float-Variablen aufnehmen kann. Jeder dieser Zeiger weist auf das jeweils erste Feldelement:

`m[0]` auf `m[0][0]`
`m[1]` auf `m[1][0]`
`m[2]` auf `m[2][0]`
`m[3]` auf `m[3][0]`

`m[i]` sind also Zeiger auf Felder von `float`-Variablen. `m` ist ein Zeiger auf das erste Feldelement dieses Felds, also auf `m[0]` und ist damit ein Zeiger auf ein Feld von Zeigern auf `float`-Variablen, also vom Typ `float **m`; Gleichzeitig zeigt `m` auch auf `m[0][0]`, weil ja `m[0]` auch dorthin zeigt.

`m`	→`m[0]`	`= & m[0][0]`♣
`m[i]`	`= & m[i][0]`	
`*m[i]`	`= m[i][0]`	`=**(m + i)`
`**m`	`= m[0][0]`	
`m + i`	`= & m[i][0]`	`= *(m + i)`
`(m + i) + 1`	`= & m[i+1][0]`	
`*(m + i) + k`	`= & m[i][k]`	
`*(*(m + i) + k)`	`= m[i][k]`	

♣ Die Adressen sind gleich, die Zeiger sind jedoch verschiedenen Typs

6.3 Parameterübergabe an Funktionen

Die Typen der Übergabeparameter an eine Funktion müssen ebenso wie der Typ des zurückgegebenen Werts in der Funktionsdeklaration festgelegt werden. Dabei ist nun von Interesse, ob die Übergabewerte in der Funktion verändert werden können bzw. ob sich eine solche Veränderung auch auf die Variablen der rufenden Funktion (*parent function*) auswirkt.
Grundsätzlich ist dabei zu unterscheiden, ob eine Variable durch ihren Wert oder durch ihre Adresse (also durch einen Pointer) übergeben wird. Wählt man nicht ausdrücklich einen Pointer (oder setzt den Adreßoperator ein), so wird der Wert der Variablen als Kopie des Originalwerts übergeben.
Die Übergabe als Wert hat zur Folge, daß eine Veränderung nur lokal (an der Kopie) vorgenommen wird und die Original-Variable in der rufenden Funktion davon nicht betroffen ist ("Übergabe als Wert").
Als Alternative können pointer als Variablen übergeben werden ("Übergabe als Adressen"). Die aufgerufene Funktion kann dadurch das Objekt, auf das der Pointer verweist, ansprechen und damit eine Veränderung der Originaldaten vornehmen. Zwar wird auch hier der Mechanismus "Übergabe als Wert" im Prinzip eingehalten, doch betrifft das nur den Pointer, den man meist nicht verändern wird.

In Fortran werden Variablen grundsätzlich durch "Übergabe der Adresse" in Subroutines und Functions übergeben.

6.3.1 Übergabe von eindimensionalen Feldern

Da der Name eines Felds immer einen Pointer auf das erste Feldelement darstellt, erfolgt die Übergabe automatisch als Adresse. Dabei kann die Deklaration entweder die gleiche Feldvereinbarung enthalten wie in der Originaldeklaration (z.B. `int iA[10]`), oder auch vereinfacht `int iA[]` oder int `*iA`. Die Feldgrenzen werden **nicht** geprüft. Die Dimension des Felds sollte daher unbedingt mitübergeben werden (falls sie der gerufenen Funktion nicht z.B. durch eine globale Variable bekannt ist).

Beispiel:

Ein float-Array soll

- Von der Tastatur eingegeben
- Sortiert
- Am Bildschirm ausgegeben

werden.

Dazu können folgende Funktionen brauchbar sein:

`int iReadArray(float *fA);`
Liest die Daten von der Tastatur und übergibt die Länge des Array an den Aufrufer.

`void ShowArray(float *fA, int iLength);`
Druckt das Array am Bildschirm

`void SortArray(float *fA, int iLength);`
Sortiert das Array.
Die Länge wird durch den `int`-Parameter `iLength` übergeben.

Das Programm könnte folgendermaßen aussehen:

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

#define MAXINDEX 20
#define SWAP(TYP,X,Y) {TYP T=X;X=Y;Y=T;}

int iReadArray(float *fA);
void ShowArray(float *fA, int iLength);
void SortArray(float *fA, int iLength);

main()
{float fArray[MAXINDEX];
 int iCount;

 if( !(iCount=iReadArray(fArray) ))exit(0);
 printf("Unsorted array %d values:\n",iCount);

 ShowArray(fArray,iCount);
 SortArray(fArray,iCount);

 printf("Sorted array %d values:\n",iCount);

 ShowArray(fArray,iCount);
 exit(0);
}
// ------------------------------
int iReadArray(float *fA)
{int i=0;

 printf("Enter Array (^D:End) :\n");
 do
  {printf(" %d: ",i);
  } while(scanf("%f",&fA[i++])!=EOF && i<MAXINDEX);

 return(i-1);
 }
// ------------------------------
void ShowArray(float *fA, int iLength)
{int i;
 for(i=0;i<iLength && i<MAXINDEX;i++)printf("%d : %f\n",i,fA[i]);
}
// ------------------------------
void SortArray(float *fA, int iLength)
{...}
```

6.3.2 Übergabe von mehrdimensionalen Feldern

Auch hier wird mit dem Namen des Felds ein Pointer übergeben, der auf das erste Feldelement zeigt. Allerdings muß nun beachtet werden, daß die interne Adressierung von mehrdimensionalen Feldern in linearisierter Weise erfolgt. Das Feldelement `iA[i][j]` ergibt sich beispielsweise als `iA + i*Nj+j`, wenn `Nj` die Anzahl der Spalten ist. Daher muß auch die Dimensionierung in der Funktionsdeklaration vorgenommen werden. Die Übergabe als Pointer `int **iA` ist zwar möglich, doch können die Feldelemente nicht mehr sicher mit `iA[i][j]` angesprochen werden und es folgt eine diesbezügliche Compiler-Warnung oder Fehlernachricht.
Die Ursache liegt darin, daß die bisher besprochene Feldvereinbarung einen kontinuierlichen Speicherbereich reserviert, in dem die lineare Adressierung möglich ist. Im Kapitel "Pointer und zweidimensionale Arrays" wurde dazu ein Beispiel mit einem Feld der Dimension `float m[4][5]` gezeigt. Dort zeigt `m` auf `m[0][0]` und gleichzeitig auf `m[0]`und weiters `m[1]` auf `m[1][0]`, usw.
Die Felder, auf die `m[0]`, `m[1]`, `m[2]` und `m[3]` zeigen, müssen aber keineswegs immer gleich lang sein und zusammenhängen, wenn sie dynamisch vereinbart wurden (siehe unten). Dann funktioniert natürlich auch die Linearisierung der Indizierung nicht und statt mit der `iA[i][j]`-Schreibweise muß das Feldelement `i,j` mit `iA[i]+j` angegeben werden.

6.3.3 Zeiger zur Übergabe von Datenstrukturen

Die Übergabe von Zeigern auf Strukturen erfolgt genauso wie die Übergabe anderer Zeiger und hat ebenfalls zur Folge, daß das Objekt in der aufgerufenen Funktion verändert werden kann. Die Dereferenzierung erfolgt mit dem Operator `->`.

Ist eine Struktur beispielsweise definiert als

```
struct extremes
{int iMin, iMax;
} Ext={-1000,1000}; /Definition mit Initialisierung */
```

so kann ein Funktionsaufruf mit Angabe des Strukturnamens erfolgen, wobei der Inhalt der Struktur kopiert wird, oder mit Angabe der Adresse.

```
void fn1(struct extremes, int);   /*Dekl. mit Wertübergabe*/
void fn2(struct extremes *, int); /*Dekl. mit Adreßübergabe*/

void fn2(struct extremes *Exxx, int iVal) /* Pointer in fn2 */
{if(iVal < Exxx->iMin) Exxx->iMin = iVal; /* Dereference-Op -> */
 if(iVal > Exxx->iMax) Exxx->iMax = iVal;
return;  /* auch rufendes Programm sieht die geänderten Werte */
}
```

Die Schreibweise `Exxx->iMin` ist äquivalent zu `(*Exxx).iMin`

6.3.4 Parameterübergabe an main

Das Betriebssystem übergibt dem Programm die Anzahl der Commandline-Parameter und Zeiger darauf. Will man sie verwenden, muß `main()` entsprechend deklariert werden:

```
void main(int argc, char** argv)
```

Dabei bedeutet die Variable `argc` (argument-count) die Anzahl der Parameter, wobei der aufgerufene Programmname als erster Parameter gilt. `argv` (argument-values) ist ein zweidimensionales `char`-Feld:
`argv[0]` Pointer auf einen String, der den Programmnamen (samt Pfad, wenn angegeben) enthält `argv[1]` Pointer auf einen String, der den ersten Parameter enthält `argv[2]` Pointer auf einen String, der den zweiten Parameter enthält.
Die Variablennamen `argc` und `argv` sind prinzipiell frei wählbar, werden aber selten anders benannt.

Beispiel:
Aufruf des Programms im Betriebssystem:

```
myProg input.dat output.dat /copy
```

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

main(int argc, char** argv)

{int i;
 for(i=0;i<argc;++i)printf("\n%i %s",i,argv[i]);
}
```

```
Ausgabe:

0 myProg
1 input.dat
2 output.dat
3 /copy
```

Achtung:

1. Wenn Zahlenwerte als Parameter angegeben werden, werden sie als Strings übergeben und müssen erst in einen Zahlenwert konvertiert werden (dafür gibt es Bibliotheksfunktionen)

2. Beachten Sie, daß Beistriche und Strichpunkte keine Trennzeichen sind, sondern als Teil des Strings angesehen werden.

6.4 Dynamische Speicherplatzverwaltung

Globale Variable:

- Adresse steht beim Programmstart fest
- Belegen feste Adressen im Datensegment

Lokale Variable:

- Stehen am Stack
- Existieren nur solange die Funktion aktiv ist

Beide Variablentypen werden beim Compilieren des Programms definiert.

Dynamische Variable:

- Definition zögert sich bis zur Laufzeit des Programms hinaus
- Sie werden am Heap (zur Laufzeit vom Betriebssystem zugewiesener Speicherplatz) gespeichert
- Sie werden immer mit einem Zeiger adressiert
- Adresse steht erst zur Laufzeit fest
- Wird der Speicherplatz nicht mehr benötigt, so kann er freigegeben werden

Dadurch wird der Speicherplatz besser genutzt, da nicht bereits beim Programmstart für alle möglichen Variablen der Platz reserviert werden muß. Das Programm kann nicht benötigten Platz freigeben, der von anderen Variablen (oder anderen Programmen, die parallel bearbeitet werden) genutzt werden kann.

Reservieren von Speicher mit malloc:
In C müssen zum Reservieren von Speicherplatz die Bibliotheksfunktionen `malloc` oder `calloc` und zum Freigeben die Funktion `free` verwendet werden:

Funktion: `void *malloc(unsigned Size)`

Wirkung: Im Heap werden Size Bytes Speicherplatz reserviert.

Funktionswert: Die Adresse des ersten Bytes. Gibt es nicht genug Platz: NULL.

z.B.:

```
int *pi;

pi=(int *) malloc(10*sizeof(int));  /*Array von 10 int-Werten*/
```

Funktion: `void *calloc(unsigned Nobj, unsigned Size)`
`Nobj` ... Anzahl der Objekte
`Size` ... Speicherplatz, den jedes Objekt benötigt

Wirkung: Im Heap werden `Nobj*Size` Bytes Speicherplatz reserviert und mit 0 initialisiert.

Funktionswert: Die Adresse des ersten Bytes. Gibt es nicht genug Platz: NULL.

z.B.:

```
float *pf;

pf=(float *) calloc(100,sizeof(float));  /*Array von 100 float-Werten*/
```

Funktion: `void free(void *p)`

Wirkung: Zeigt p auf einen zuvor belegten Speicherblock, wird dieser freigegeben.

6.5 String-Funktionen

Ein String ist ein Array von Zeichen. An der letzten Stelle steht das ASCII-Zeichen 0x00 (nicht mit dem NULL-Zeiger zu verwechseln). Das Kopieren und die Wertzuweisung mit Strings ist nicht so einfach wie in anderen Programmiersprachen.

Eine Zuweisung wie `char s[10]="A?string",t[10],*s1;` funktioniert nur bei der Initialisierung.

Speicherbelegung:

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]	s[8]	s[9]
A	?	s	t	r	i	n	g	\0	
65	32	115	116	114	105	110	103	0	

Eine Zuweisung wie:

```
s="Another";
t=s;
```

verursacht einen Fehler.

Der Code

```
char *s={"Example"};
char *t;
t=s;
```

verursacht zwar keinen Fehler, `s` wird jedoch nicht auf `t` kopiert, sondern `t` erhält die Adresse von `s` zugewiesen.
D.h.: Änderungen, die durch `t` durchgeführt werden, wirken sich unmittelbar auf `s` aus, da `s` und `t` auf denselben Speicherplatz zeigen !!

Eine Möglichkeit einen String zu kopieren ist:

```
char s[10]={"Example"},t[10];
 int i=0;

do
t[i]=s[i];
while(s[i++]);
```

Weiteres Beispiel:

```
char s[10]={"Example"}, *t;
 int i=0;

do t[i]=s[i];
while(s[i++]);
```

Dieses Programm verursacht keinen Fehler beim Compilieren. `t` ist jedoch eine nicht initialisierte Zeigervariable, sie zeigt auf nicht reservierten Speicher. Das Verhalten des Programms ist unvorhersehbar, da die Schleife `t[i]` fremde Speicherbereiche überschreibt.
Korrekt wäre:

```
char s[10]={"Example"},*t;
 int i=0;

t=(char*) malloc(10); //reserviert Speicher
do
 t[i]=s[i];
while(s[i++]);
```

Zur Manipulation von Strings gibt es mehrere Bibliotheksfunktionen. Sie sind in `<string.h>` deklariert.

6.6 Zeiger auf Funktionen

Werden zum Beispiel gebraucht, um Funktion einer anderen Funktion als Parameter zu übergeben.

Beispiel:
Es sollen die Nullstellen einer beliebigen Funktion bestimmt werden.

```
float NullStelle (float anfg, float ende, float eps, float (* fn)(float x));
```

`anfg, ende`: Legen das Intervall fest
`eps`: Um wieviel der Funktionswert von 0 verschieden sein darf (Abbruchbedingung)

Die konkrete Funktion wird als Parameter übergeben:

```
float (* fn)(float x):
```

Stellt den Zeiger auf eine Funktion dar, die einen `float`-Wert übergibt und einen `float`-Wert als Parameter hat.

Achtung:

```
float *fn(float x);
```

Stellt eine Funktion dar, die einen `float`-Zeiger zurückgibt. Nicht mit dem obigen Beispiel verwechseln.

Folgender Ausdruck berechnet sin(dPhi) oder cos(dPhi):

```
(i==1 ? sin : cos)(dPhi);
```

Ein Programm, das die Nullstellen von 2 verschiedenen Funktionen ausgibt, könnte etwa folgendermaßen aussehen:

```
float NStelle(float anfg, float ende, float eps,
              float (* fn)(float x));
float Polynom2(float x);
float Polynom3(float x);

// ----------------------------
int main(void)
{
 float UGrenze,OGrenze,Eps,NSt;

 UGrenze = -2;
 OGrenze = 0;
     Eps = 1.E-5;

     Nst = NStelle(UGrenze, OGrenze, Eps, Polynom2);

 printf("Nullstelle Polynom2: %f\n",Nst);

 UGrenze = 5;
 OGrenze = 10;
     Eps = 1.E-6;

     Nst = NStelle(UGrenze ,OGrenze, Eps, Polynom3);

 printf("Nullstelle Polynom3: %f\n",Nst);
 exit 0;
}
// ----------------------------
float Polynom2(float x)
{return (x-1)*(x+1);}
// ----------------------------
float Polynom3(float x)
{return (x-1)*(x+1)*(x+7);}
// ----------------------------
float NStelle(float anfg, float ende, float eps,
              float (* fn)(float x))
{...}
// ----------------------------
```

Eine Funktion, die ein Polynom mit beliebig vielen Koeffizienten berechnet:

```
float Polynom(int iOrder, float *Coef, float x)
{ int i;
 float sum,xs;

 for(i=0,sum=0,xs=1;i<iOrder;i++)
    {if(i>0)xs*=x;
     sum+=xs*Coef[i];
    }
 return sum;
}
```

typedef mit Funktionen:

```
typedef int (*PFI) (char*, char*);

PFI strcmp, numcmp; /*Deklaration von zwei Funktionen */
```

Pointer auf eine Funktion, die als Argumente zwei Pointer auf char hat und int zurückliefert.

6.7 Elementare Datenstrukturen

Die grundlegende Arbeitsweise eines Programms (Funktion)::
Für viele Anwendungen ist die Wahl der richtigen Datenstruktur wesentlich. Bei geeigneter Wahl der Datenstruktur sind oft nur noch einfache Algorithmen zur Problemlösung notwendig. Durch die richtige”Wahl der Datenstruktur und des Alorithmuses kann durch Einsparung von Zeit und/oder Speicherplatz ein Programm entsprechend optimiert werden.
Eine Datenstruktur ist kein passives Objekt; es müssen immer die Operationen berücksichtigt werden, die mit ihr ausgeführt werden sollen.

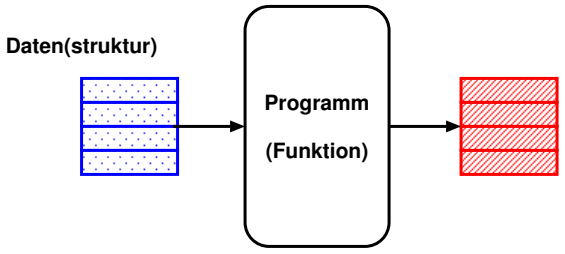

Abbildung 6.1: Arbeitsweise eines Programms

6.7.1 Felder (*arrays*)

Die grundlegenste Datenstruktur, die in vielen Programmiersprachen als Grundelement definiert ist.

Ein **Feld** (*array*) ist eine feste Anzahl von einzelnen Datenelementen, die zusammenhängend gespeichert werden und über einen **Index** zugänglich sind.

Das `i`-te Element eines Feldes `a` wird als `a[i]` bezeichnet. Der Programmierer ist dafür verantwortlich, daß etwas Sinnvolles auf einer Feldposition `a[i]` gespeichert wird, bevor darauf Bezug genommen wird. (Nichterfüllen dieser Forderung ist ein häufiger Programmierfehler).
Durch die sequentielle Anordnung ergibt sich folgendes Hauptmerkmal für Felder:

Bei bekanntem Index ist ein Zugriff auf jedes Element in konstanter Zeit möglich

Die Größe eines Feldes muß im voraus bekannt sein. Meist muß die Größe zur Compilezeit feststehen, in manchen Programmierumgebungen ist es möglich, die Größe eines Feldes erst zum Zeitpunkt der Prorammausführung festzulegen. Dadurch kann der Anwender eines Programms die benötigte Größe eines Feldes angeben und es muß nicht der Programmierer die maximal mögliche Größe des Feldes im vorhinein festlegen und dadurch Speicherplatz verschwenden.
In C ist es möglich durch Zeiger und dynamische Speicherplatzzuweisung diesen Mechanismus zu nutzen.
Trotzdem bleibt es eine wesentliche Eigenschaft von Feldern:

Die Größe muß vor der Verwendung bekannt sein und feststehen

Felder sind deshalb grundlegende Datenstrukturen, da sie in in direktem Zusammenhang zu Speichersystemen in praktisch allen Computersystemen stehen. Um in Maschinensprache auf einen Speicherplatz zuzugreifen ist die Angabe einer Adresse notwendig. Damit stellt der gesamte Speicher eines Computers ein Feld dar, wobei die Speicheradressen den Feldindizes entsprechen. Die meisten Compiler übersetzen Programme, in denen Felder auftreten, effizient in Maschinensprache, indem sie direkt auf den Speicher zugreifen.

Eine weitere Möglichkeit zur Strukturierung von Informationen ist die Verwendung von zweidimensionalen Tabellen, die in Zeilen und Spalten angeordnet sind.

Eindimensionale Felder entsprechen **Vektoren**.
Zweidimensionale Felder entsprechen **Matrizen**.

6.7.2 Verkettete Listen (*linked lists*)

Eine verkettete Liste (*linked list*) ist eine sequentielle Anordnung von Elementen. Die explizite sequentielle Anordnung wird dadurch erreicht, daß jedes Element Teil eines **Knotens** (*nodes*) ist, der auch eine "Verkettung" zum nächsten Knoten enthält.

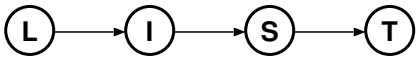

Abbildung 6.2: Verkettetung von Daten

Vorteile verketteter Listen gegenüber Feldern:

1. Die Größe der Liste kann sich ändern.
2. Die maximale Größe der Liste muß nicht im voraus bekannt sein.
3. Höhere Flexibilität beim Umordnen einzelner Elemente.

Nachteil verketteter Listen gegenüber Feldern:

1. Die höhere Flexibilität beim Umordnen einzelner Elemente geht auf Kosten der Zugriffszeit auf ein beliebiges Element der Liste

C stellt elementare Operationen bereit, die es gestatten verkettete Listen direkt zu implementieren. Weiters stellt C++ Bibliotheken bereit, welche die Verwendung verketteter Listen wesentlich vereinfachen.

Ein Listenknoten kann durch folgende Struktur beschrieben werden:

```
struct node
 { int key;
   struct node *next;
 };
```

Eine Liste setzt sich aus **Knoten** (*nodes*) zusammen. Wobei jeder Knoten ein Datenfeld (hier `key`; eine ganze Zahl) und einen Zeiger zum nächsten Knoten der Liste (`struct node *next`) enthält.
Das Datenfeld (`key`) ist hier nur der Einfachheit halber eine ganze Zahl, es könnte beliebig komplex sein.

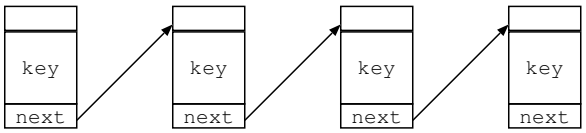

Abbildung 6.3: Einfach verkettete Liste

Mit dieser Darstellung ergeben sich 2 Probleme:

1. Wohin zeigt `next` des letzten Listenknotens
2. Wo beginnt die Liste

Lösung:

1. Der `next`-Zeiger des letzten Listenknotens ist der `NULL`-Zeiger
2. Das zweite Problem läßt sich mit einem sogenannten *Pseudoknoten* lösen. Der Pseudoknoten enthält kein Datenfeld. Dieser Knoten besteht nur aus einem Zeiger auf das erste Listenelement, man nennt ihn meist Kopf (*head*).

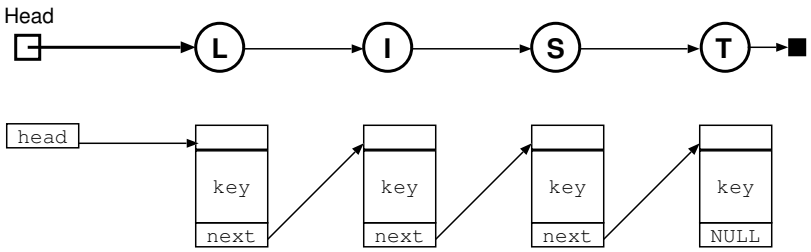

Abbildung 6.4: Pseudoknoten

Diese explizite Darstellung macht es nun möglich gewisse Operationen wesentlich effizienter auszuführen als bei einem Feld.

Verschieben eines Elements:

In einem Feld müssten wir jedes Element verschieben um Platz für das neue Element zu schaffen. In der verketteten Liste ändern wir nur die Verkettungen (Zeiger) wie in der folgenden Abbildung gezeigt wird.
Vor allem bei umfangreichen Datenfeldern oder Listen ergibt sich hier eine Zeitersparnis, da das oftmalige kopieren von großen Datenmengen vermieden wird. Es brauchen nur Zeiger umkopiert werden.

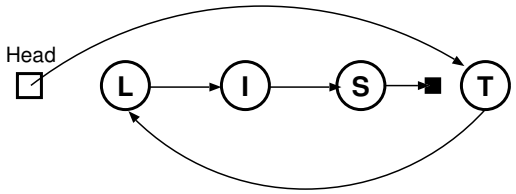

Abbildung 6.5: Verschieben eines Listenelementes

Vertauschen des ersten und letzten Listenelements

1. Der Knoten der T enthält wird auf den Knoten mit L ”verbogen”.
2. Der Kopf zeigt auf T.
3. Der Knoten der S enthält zeigt auf NULL (Ende).

Mit der Veränderung von nur 3 Verkettungen können die Elemente in einer beliebig langen Liste vertauscht werden. Bei der Reihenfolge der Veränderungen ist zu beachten, daß kein Zeiger auf ein Listenelement ”verloren geht” (führt man Schritt 3 zuerst aus, geht der Zeiger auf L verloren).

Einfügen eines Elements:

In einem Feld ist diese Operation naturgemäß sehr umfangreich, da die Größe um 1 wächst und wieder Elemente verschoben (umkopiert) werden müssen. Die folgende Abbildung zeigt wie dies bei einer Liste durchgeführt wird.

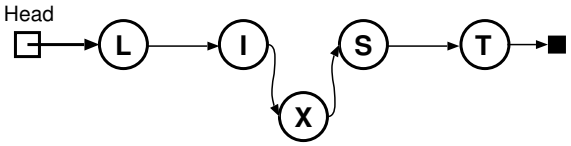

Abbildung 6.6: Einfügen eines Listenelementes

Einfügen eines Listenelements

1. X wird in einen Knoten eingefügt, der auf S zeigt.
2. Der Knoten der I enthält zeigt auf den neuen Knoten.

Nur 2 Verkettungen müssen für diese Operation in einer beliebig langen Liste verändert werden.

Entfernen eines Elements:

Bei diesem Vorgang verringert sich die Größe der Liste um 1 In einem Feld müssen wieder Elemente verschoben (umkopiert) werden. Die folgende Abbildung zeigt wie dies bei einer Liste durchgeführt wird. Entfernen eines Listen-

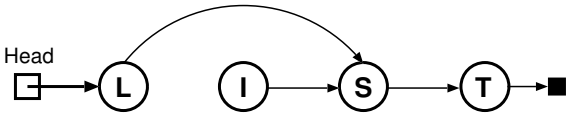

Abbildung 6.7: Entfernen eines Listenelementes

elements

1. Man läßt den Knoten der L enthält auf S zeigen.

Der I enthaltende Knoten existiert zwar noch (er zeigt sogar immer noch auf S) und sollte auf irgend eine Weise beseitigt werden. entscheidend ist jedoch, daß er kein Bestandteil der Liste mehr ist. Er kann nicht mehr erreicht werden, indem man vom *Kopf* aus den Verkettungen folgt.

Es gibt auch Operationen, die für verkettete Listen **nicht** gut geeignet sind:

Finden des k-ten Elements:

Auffinden eines Elements, wenn dessen *Index* gegeben ist.
In einem Feld erreicht man dies einfach durch den Zugriff auf `a[k]` (konstante Zugriffszeit). In einer Liste müssen `k` Verkettungen durchlaufen werden (Zugriffszeit von `k` abhängig).

Entfernen des Elements **vor** einem gegebenen Element:

Auch hier muß die Liste soweit durchlaufen werden bis jener Knoten gefunden wird, der auf das entsprechende Element zeigt. Diese Operation ist erforderlich um einen gegebenen Knoten aus einer Liste zu entfernen. Dieses Problem kann dadurch umgangen werden, daß man die Operation auf ”Entfernen des nachfolgenden Elements” umändert.

Um mit einer Liste entsprechend arbeiten zu können sind folgende Funktionen zu implementieren:

1. Initialisieren
2. Nachfolgendes Element entfernen
3. Nach einem gegebenen Element einfügen

Beispiel:

```
struct node
{         int key;     // Vereinfachte Daten
  struct node *next;   // Zeiger auf naechstes Element
};

//Initalisiert die Liste
  void  initialize(int v);
// Loescht den auf t folgenden Koten
  void  delete_next(struct node *t);
// Fuegt v auf t folgenden Koten ein
struct node *  insert_after(int v, struct node *t);
// Durchlaufen der Liste und drucken von key und next
void print_list(void);
// Aufsuchen des k-ten Elements
int get(int k);
// ------------------
struct node *head;  //Pseudoknoten zur Verwaltung
// ------------------
void initialize(int v)
{ head = (struct node *) malloc(sizeof *head);
  head->next = NULL;  //Leere Liste
  head->key = v;
}
// ------------------
void delete_next(struct node *t)
{if(t->next)
   {struct node *x=t->next->next;
    free(t->next); //Freigeben des Speichers
    t->next = x;
   }
}
// ------------------
struct node * insert_after(int v, struct node *t)
{struct node * x;
 x = (struct node *) malloc(sizeof *x);
 x->key  = v;
 x->next = t->next;
 t->next = x;
 return x;
}
// ------------------
void print_list(void)
{struct node * x;
 for(x=head; x; x=x->next)printf("This:%p Key:%3d Next:%p\n",x,x->key,x->next);
}
```

```
// ------------------
int get(int k)
{struct node * x;
 int i;
 for(x=head,i=0; i<k && x->next; x=x->next)i++;
 if(i<k)
   {printf("Liste zu klein\n");
    return 0;
   }
 return x->key;
}
// ------------------
main()
{struct node *x;
 initialize(0);
 x=insert_after(1,head);
 x=insert_after(2,x);
 x=insert_after(3,head);
 x=insert_after(4,x);
 print_list();
 printf("Delete after %p\n",x);
 delete_next(x);
 print_list();
 printf("list[1]=%d\n",get(1));
 printf("list[4]=%d\n",get(4));
}
```

Zyklische Liste:
Eine weitere gebräuchliche Vereinbarung zum Beenden einer Liste besteht darin, daß man den letzten Knoten wieder auf den ersten Knoten zeigen läßt. Man spricht dann von der *zyklischen Liste*. Dies bietet einem Programm eine einfache Möglichkeit eine Liste, solange sie nicht leer ist, immer wieder zu durchlaufen.

Doppelt verkettete Liste:
Die Operation "Auffinden eines Elements **vor** einem gegebenen Element" kann durch die sogenannte *doppelt verkettete Liste* erleichtert werden. Dabei werden für jeden Knoten zwei Verkettungen festgelegt: Eine zum vorhergehenden und zum nachfolgenden Element.

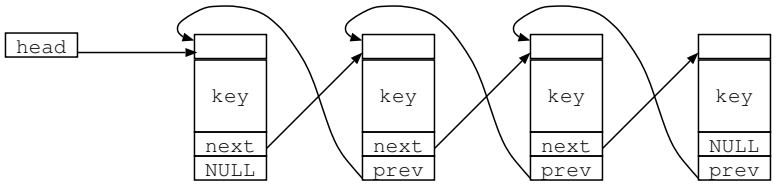

Abbildung 6.8: Doppelt verkettete Liste

6.7.3 Stapel (*stacks*)

In den bisherigen Datenstrukturen konnten Elemente fast beliebig eingefügt und entfernt werden. In der Praxis ist es oft möglich diesen Zugriff einzuschränken und trotzdem flexible Datenstrukturen zu erhalten, da weniger Operationen unterstützt werden müssen. Die wichtigste Datenstruktur mit beschränktem Zugriff ist der **Stapel** (*stack*). Es treten nur 2 Grundoperationen auf:

1. Ein Element auf dem Stapel **ablegen** (*push*).
   D.h. am Anfang einfügen.

2. Ein Element vom Stapel **entnehmen** ().
   D.h. vom Anfang entfernen.

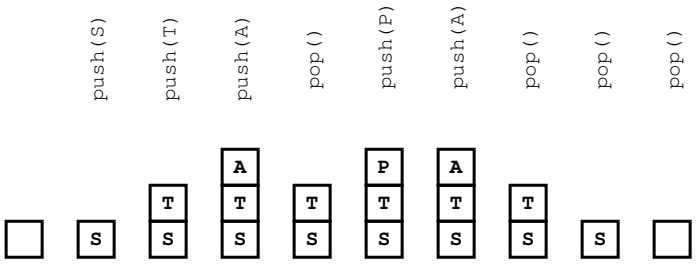

Abbildung 6.9: Stapel (stack)

Das funktioniert ähnlich wie wenn man ein Blatt nach dem anderen auf einen Stapel legt. Wenn man Zeit hat nimmt man ein Blatt vom Stapel und bearbeitet es. Dabei kann manches Blatt am Boden des Stapels für längere Zeit liegen bleiben. Doch es wird einem hoffentlich gelingen dem Stapel hin und wieder ganz abzuarbeiten.
Oft ist ein Programm der Art organisiert, daß manche Aufgaben aufgeschoben werden, während andere erledigt werden müssen. Deshalb ist der Stapel die grundlegende Datenstruktur für viele Algorithmen.

Beispiel: Auswertung arithmetischer Ausdrücke
`5*(((9+8)*(4+6))+7)`
Mit einem Stapel können die Zwischenergebnisse einer solchen Rechnung leicht gespeichert werden. Zur Berechnung sind folgende Stackoperationen notwendig:

```
push(5);
push(9);
push(8);
push(pop()+pop());
push(4);
push(6);
push(pop()+pop());
push(pop()*pop());
push(7);
push(pop()+pop());
push(pop()*pop());
printf("%d\n",pop());
```

Die Reihenfolge der Operationen wird durch die Klammern und durch die Vereinbarung, daß von links nach rechts vorgegangen wird, bestimmt. Für nicht kommutative Operatoren (-/) ist ein komplizierterer Kode erforderlich.

Implementation eines Stacks mittels verketteter Listen:

```
struct node
{         int key;
  struct node *next;
};

//Initalisiert den Stapel
  void  initialize(int v);
// Element einfuegen
  void push(int v);
// Element entfernen
  int  pop(void);
// ------------------
struct node *head, *t;
// ------------------
void initialize(int v)
{ head = (struct node *) malloc(sizeof *head);
  head->next = NULL;
  head->key = v;
}
// ------------------
void push(int v)
{t = (struct node *) malloc(sizeof *t);
 t->key  = v;
 t->next = head->next;
 head->next = z;
}
// ------------------
int  pop(void)
{int i;
 t = head->next;
 head->next = t->next;
 i=t->key;
 free(t);
 return i;
}
// ------------------
```

Wenn die maximale Größe eines Stapels im vorhinein bekannt ist, kann der Stack mit Hilfe eines Feldes dargestellt werden:

```
#define MAX_STACK_SIZE 100
int stack[MAX_STACK_SIZE + 1];
int p;  // stack pointer

void stack_init();
void push(int v);
 int pop(void);
 int stack_empty(void);
//---------------
void stack_init()
{p=0;}
//---------------
void push(int v)
{if(p<MAX_STACK_SIZE) stack[p++]=v;
 else printf("Stack overflow\n");
}
//---------------
int pop(void)
{if(!stack_empty)return stack[--p];
 else
 {printf("Stack underflow\n");
  return 0;
 }
}
//---------------
int stack_empty(void)
{return !p;}
//---------------
```

Schlangen:
Eine weitere Datenstruktur mit zwei Zugriffsarten ist die **Schlange** (*queue*). Es treten folgende 2 Grundoperationen auf:

1. Ein Element am Anfang in die Schlange **einfügen** (*put*).
2. Ein Element vom Ende der Schlange **entfernen** (*get*).

Gemäß des Beispiels mit den Blättern, werden hier Zettel die zuerst abgelegt werden auch zuerst abgearbeitet. Stapel arbeiten nach dem Prinzip *" last in, first out"* (LIFO), für Schlangen gilt jedoch *" first in, first out"* (FIFO).
Queues können ebenso wie Stacks, entweder als verkettete Liste oder mit Hilfe von Feldern realisiert werden.

6.7.4 Bäume (*trees*)

Die bisher behandelten Strukturen waren alle eindimensional, ein Element folgte dem anderen. **Bäume** (*trees*) sind zweidimensional verkettete Datenstrukturen. Alltägliche Beispiele für eine Baumstruktur sind der Familienstammbaum, die Verzeichnisstruktur auf der Festplatte des PCs, das Organisationsschema eines Großbetriebs oder die Organisation eines Sportturniers (Finale, Semi-, Viertelfinale, ...).

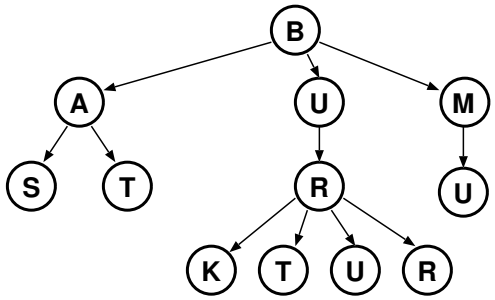

Abbildung 6.10: Baum (tree)

Ein Baum ist eine (nicht leere) Menge von **Knoten** und **Kanten**.

Knoten: Ein Objekt, das einen Namen und andere mit ihm verknüpfte Informationen tragen kann

Kante: Eine Verbindung zwischen zwei Knoten

Pfad: Eine Liste von aufeinanderfolgende Knoten, die durch Kanten im Baum verbunden sind

Wurzel: Einer der Knoten im Pfad

Für die Definition wesentliche Eigenschaft eines Baums:
Zwischen der Wurzel und jedem beliebigen anderen Knoten eines Baums, gibt es genau einen Pfad.
Bäume werden hier mit der Wurzel an der Spitze gezeichnet. Jeder Knoten außer der Wurzel besitzt damit genau einen Knoten, der sich genau über ihm befindet und wird als **direkter Vorgänger** (*parent*) bezeichnet. Im angegebenen Beispiel ist B die Wurzel und B der direkte Vorgänger von A oder A der direkte Vorgänger von S.
Die Knoten unmittelbar unter einem Knoten werden seine **direkten Nachfolger** (*children*) genannt.
Knoten ohne Nachfolger werden Blätter oder Endknoten oder *äußere* Knoten genannt. *Nichtend*knoten heißen demgemäß *innere* Knoten.
Jeder Knoten kann als Wurzel eines Unterbaumes aufgefaßt werden, der aus ihm selbst und den Knoten unterhalb besteht.

Die Knoten eines Baumes lassen sich in Ebenen einteilen. Die Ebene eines Knotens ist die Anzahl der Knoten auf dem Pfad des jeweiligen Knotens zur Wurzel.

Falls jeder Knoten eine bestimmte Zahl von direkten Nachfolgern hat, spricht man von einem **n-närem Baum**.
Der einfachste Typ eines n-nären Baums ist der **binäre Baum**. Das ist ein geordneter Baum, der aus zwei Typen von Knoten besteht:

1. Äußere Knoten (ohne Nachfolger)
2. Innere Knoten mit genau zwei direkten Nachfolgern

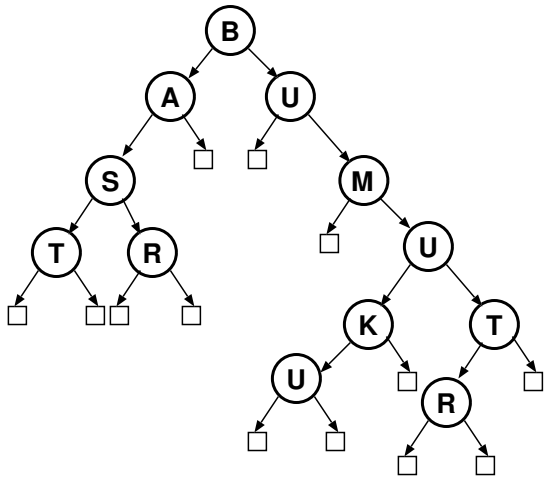

Abbildung 6.11: Binärer Baum

In einem **vollen** binären Baum ist jede Ebene mit Ausnahme der letzten vollständig mit inneren Knoten ausgefüllt.
Ein **vollständiger** binärer Baum ist ein voller binärer Baum, bei dem alle inneren Knoten der untersten Ebene links von den äußeren Knoten dieser Ebene erscheinen.

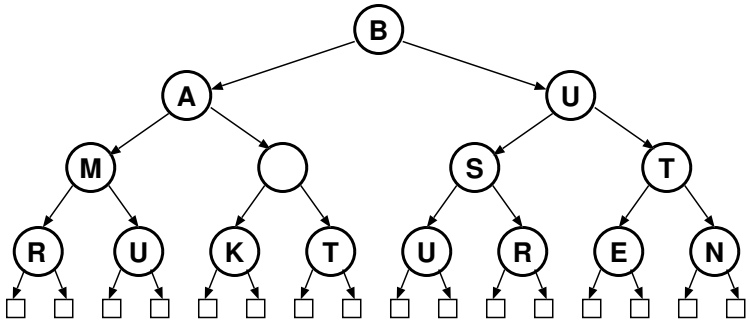

Abbildung 6.12: Vollständiger binärer Baum

Die Effizienz beim durchlaufen von binären Bäumen ist dann besonders hoch wenn sie vollständig sind.

Beispiel für einen Baumknoten:

```
struct tree_node
{int key;     // Vereinfachte Daten
 struct tree_node *left, *right;  // Linker, rechter Nachfoler
};
```

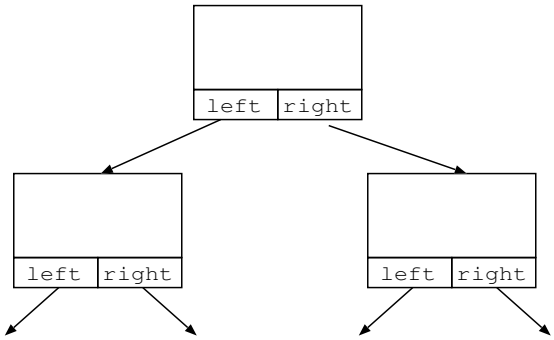

Abbildung 6.13: Baumknoten

7 C++

7.1 Strukturiertes Programmieren

7.1.1 Funktionen

Eine Funktion ist ”kleines Programm” im Programm. Bestimmte Anweisungen sind unter einem Namen zusammengefaßt. Sie kann in einem Programm auch mehrmals aufgerufen werden. Durch Einsparung von Wiederholungen kann Platz gespart werden. Die Lesbarkeit wird durch Zerteilung in kleinere, übersichtlichere Module verbessert. Große Programme zerfallen in kleine übersichtliche Quelltext-Teile.

– Teamarbeit –

In BASIC, FORTRAN,Pascal heißen Funktionen Subroutinen oder Prozeduren

Richtlinien für eine ”gute” Funktion:
Überlegen einer präzisen, genauen Aufgabenstellung. Verwendung von prägnanten, selbsterklärenden Namen.

gut: `ClearScreen, Summe, ZeigeFehler`

eher schlecht: `mx0815, s1, BildschirmLoeschenUndInitalisieren`

Die Länge einer Funktion sollte ca. 2 Seiten (100 Zeilen) nicht überschreiten.

Schreiben eines Programms mit Funktionen:

- Die Deklaration eines Prototyps (beschreibt die Funktion) erfolgt meistens vor `main`. Sie endet mit Strichpunkt `;`. Für Bibliotheksfunktionen geschieht dies in den HEADER-Dateien. Bei längeren Programmen sollte man auch die eigenen Prototypen in HEADER-Dateien zusammenfassen.
- Aufruf der Funktion in mindestens einer Anweisung.
- Schreiben des Quelltextes (Implementation). Die Implementation besteht aus einer Kopfzeile (Funktionsdeklaration) und dem weiteren Code. Prototyp und Funktionsdeklaration müssen genau übereinstimmen.

Funktionen können Werte zurückliefern und Parameter übernehmen. Ist dies nicht der Fall, so sollte das Schlüsselwort `void` anstelle des leeren Platzes eingefügt werden!

z.B.

```
void ErrorMsg(char *szMsg);
int GetChar(void);
void ClearScreen(void);
```

7.1.2 Funktionen und Variablen

In einer Funktion können fast alle Anweisungen stehen und alle möglichen Variablen definiert werden. Anweisungen einer Funktion können **nicht** auf Variablen fremder Funktionen zugreifen.
Variablen die in einem Funktionsblock deklariert werden sind **"Lokale Variable"**
Vorteile:

1. In einer Funktion kann der Wert einer lokalen Variablen nicht von außerhalb geändert werden.
2. Die Namen lokaler Variablen brauchen nicht eindeutig zu sein.
3. Lokale Variablen existieren erst dann im Speicher, wenn die Funktion abläuft. Dadurch wird Speicherplatz gespart; viele Funktionen können den selben Platz verwenden (jedoch nicht gleichzeitig).

Lokale Variable:

Sie existieren erst im Speicher, wenn die Funktion aktiv ist d.h. zur Laufzeit ausgeführt wird.

Vorteil: Läuft die Funktion nicht ab, so verbrauchen die lokalen Variablen keinen Speicherplatz.

Beim Start einer Funktion werden lokale Variable am Stack abgelegt. Nach dem Ablauf der Funktion wird der Speicherbereich wird wieder freigegeben und andere Variable können ihn benutzen.
STACK: Speicherbereich, der nach Bedarf wächst und schrumpft.

Um die Verwaltung dieses Stacks braucht sich der Programmierer nicht kümmern, sie geschieht automatisch (Compiler).

Lokale Variable sind während des Funktionsablaufes geschützt. Danach werden sie überschrieben; d.h. sie ändern sich von einem Funktionsaufruf zum anderen (nicht einmal das ist sicher). Sie werden durch Typ- und Namensangabe im Funktionsblock deklariert.

Externe Variable:

Externe Variable werden außerhalb von `main` in einem anderen Modul deklariert. Soll nun eine Funktion auf eine solche Variable zugreifen, so muß diese dort als extern deklariert werden.

Lokale und globale Variable unterscheiden sich nur in der Adresse und in der Art der Speicherung (Stack bzw. Datensegment).

Beispiel:
Üblicherweise wird ein langes Programm in mehrere Dateien aufgeteilt.
z.B.

Datei1:	Globale Variable + main
Datei2-Datei3:	Funktionen der versch. Kategorien

Datei1: Alle Funktionen in dieser Datei (`funktion1` und `funktion11`) können auf `ERR_NO` zugreifen, da die Definition von `ERR_NO` außerhalb aller Funktionsblöcke erfolgt ist. Die Variable `x` ist lokal, in keiner anderen Funktion außer `function1()` ist der Zugriff auf sie möglich (auch in `main` nicht).

```
int ERROR_NO;
...
main();
{...
 ERROR_NO=...;
}

funktion1(...)
{double x=7.59;
 ...}
funktion11(...)
{...}
```

Datei2: Nur `funktion2` kann auf `ERR_NO` zugreifen, da das `extern`-Statement innerhalb des Funktionsblocks von `funktion2` steht.

```
funktion2(...)
{extern int ERROR_NO;
 ...
}
funktion21(...)
{...}
```

Datei3: Alle Funktionen können auf `ERR_NO` zugreifen, da das `extern`-Statement außerhalb der Funktionsblöcke steht.

```
extern int ERR_NO
funktion3(...)
{...}
...
```

Statische Variablen:

Lokale Variable sind temporär:
Ihre Werte gehen von einem Funktionsaufruf zum anderen verloren.
Soll eine Variable ihren Wert behalten (das heißt nicht daß sie von außen zugänglich ist), so muß sie als `static` deklariert werden. Statische Variablen werden an einer festen Speicheradresse im Datensegment (nicht am Stack) gespeichert. Die Initialisierung erfolgt nur einmal.

```
#include <stdio.h>

int InkrStatic(void);
int InkrLokal(void);

main()
{int i;
 printf("Aufruf von InkrStatic:\n");
 for(i=1;i<=10;i++)
     printf(" %d",InkrStatic());

 printf("\nAufruf von InkrLokal:\n");
 for(i=1;i<=10;i++)
     printf(" %d",InkrLokal());
}
// -------------------------
int InkrStatic(void)
{static int iWert=1;
 return iWert++;}
// -------------------------
int InkrLokal(void)
{int iWert=1;
 return iWert++;}
// -------------------------
Ergebnis:
 1.Schleife: 1 2 3 4 5 6 7 8 9 10
2. Schleife: 1 1 1 1 1 1 1 1 1 1
```

7.1.3 Parameterübergabe

Funktionen erhalten ihre Argumente von der aufrufenden Prozedur. Sie werden dabei in die entsprechenden Typen konvertiert.

Funktionsprototyp: Enthält die Variablendeklaration.

Der Name der Variablen muß nicht unbedingt angegeben werden. Es würde z.B. genügen:

```
long Fakt(int);
void circle(int,int,int);
```

Parameternamen sind jedoch zu bevorzugen, da sie bereits einen Hinweis auf den Zweck des Wertes geben.
z.B.:

```
void circle(int x,int y,int iRadius);
```

Übergabe von Werten:
An die Funktion werden immer nur Kopien der Werte übergeben. Nicht die Variablen, sondern nur deren Werte gelangen in die Funktion. Die Funktion kann den Wert nicht ändern.
z.B.:

```
void EnterChar(int cValue)
{ printf("Enter a character: ");
  cValue=getchar();
}
```

Bei einem Aufruf von EnterChar wird die Variable des aufrufenden Programms nicht verändert.

```
int c;
c='A';
EnterChar(c);
//Noch immer ist c='A' !!
```

Übergabe von Referenzen:
Oft ist es notwendig, daß eine Funktion eine übergebenen Parameter ändern soll. z.B. `EnterChar` soll den übergebenen Wert direkt ändern.

Lösung: Den Parameter als Referenz (Zeiger) zu übergeben.

Referenzparameter beginnen mit dem Zeichen `&`. Damit gestattet der Compiler der Funktion den Zugang zur Variablen. Anstatt einer Kopie übergibt das Programm die Speicheradresse der Variablen.

```
void EnterChar(int &cValue)
{printf("Enter a character: ");
cValue=getchar();
}
```

Bei einem Aufruf von `EnterChar` wird nun die übergebene Variable geändert.

```
int c;
c='A';
EnterChar(c);
/* c hat nun den eingegebene Wert !!*/
```

Vorgabeargumente:
Ein Vorgabeargument ist optionales Argument, d.h. ein Parameter, den man beim Aufruf einer Funktion nicht angeben muß. Wird kein Wert für diesen Parameter bereitgestellt, so wird ein Vorgabewert als Argument übergeben.
z.B.

```
// Zeichnet iCount Zeichen c in Zeile iRow, Spalte iCol
void Underline(int iRow, int iCol, int iCount, char c='-');
```

Der Prototyp dieser Funktion hat vier Parameter, der letzte hat einen Vorgabewert. Man kann Underline jetzt auf 2 Arten aufrufen:

1. `Underline(4,1,40);`
   Zeichnet 40 "-"-Zeichen ab Zeile 4 Spalte 1.

2. `Underline(4,1,40,'=');`
   Zeichnet 40 "="-Zeichen ab Zeile 4 Spalte 1.

Regeln für Vorgabeargumente:

- Vorgabeargumente müssen als Werte übergeben werden. Ein Name eines Vorgabearguments kann nicht als Referenz (mit `&`) übergeben werden.

- Die Werte von Vorgabeargumenten können Festwerte oder Konstantendefinitionen sein, aber keine Variablen.

  ```
  const int n;
  int k;

  example(int iValue=n); // ERLAUBT!!
  example(int iValue=k);// NICHT erlaubt!!
  ```

- Vorgabeargumente müssen zuletzt im Prototyp stehen. Nach dem ersten Vorgabeargument müssen alle solche sein.

  ```
  /* ERLAUBT:*/
  void fn(int i, float p=3.14,char c='$');

  /* NICHT erlaubt:*/
  void fn(float p=3.14, int i, char c='$');
  ```

  Sehr flexibel sind z.B.:

  ```
  #define MAX_LEVEL 100
  void SetLevel (int iLevel=MAX_LEVEL);
  ```

7.1.4 Inline-Funktionen

Funktionsaufrufe brauchen relativ viel Zeit (ablegen und zurückholen der Parameter am Stack). Daraus können sich Auswirkungen auf die Laufzeit ergeben, wenn eine Funktion in einer Schleife mehrere tausend Mal aufgerufen wird.

Abhilfe: Die Inline-Funktion:

Sie kann alle Anweisungen wie eine gewöhnliche Funktion ausführen. Sie wird meist vor main (Header-Datei) deklariert und definiert. z.B.:

```
inline    int Add10(int i) {return (i+10);}
inline double Square(double d){return d*d;}
```

Der Funktionsrumpf folgt unmittelbar nach der Deklaration. Er muß nicht in einer Zeile stehen. Der Compiler setzt die Funktion an der Stelle des Aufrufes direkt in das Programm ein (ähnlich einem Makro). Es reduziert sich der "Verwaltungsaufwand" des Funktionsaufrufes; das Programm braucht weniger Zeit.
Regeln für Inline-Funktionen:

- Bei zu langen Funktionen ignoriert der Compiler das reservierte Wort inline (Grenze ist unsicher).
- Zu viele Inline-Funktionen können Programme unnötig groß machen.
- Makros ähneln den Inline-Funktionen. Inline-Funktionen sind jedoch zu bevorzugen (Typprüfung).
- Vorteile kommen nur an kritischen Programmstellen zum Tragen. Zu viele Inline-Funktionen optimieren das Programm sicher nicht.

7.2 Was ist anders in C++

7.2.1 Referenzen

Eine Referenzen ist ein **alternativer Name** für ein Objekt.

Sie werden fast ausschließlich verwendet um Funktions-Argumente und -Resultate zu übergeben

```
void v_incr(int i)
{ i++;}
// *************************
void r_incr(int &i)
{ i++;}
// *************************
void p_incr(int *i)
{ (*i)++;}
```

```
// *************************
main()
{i=1;
 v_incr(i);    //i=1;
 i=1;
 r_incr(i);    //i=2;
 i=1;
 p_incr(&i);   //i=2;
}
```

Allgemeines über Referenzen:

Referenzen **müssen** initialisiert werden (L-Wert).

```
int i=1;
int & r=i; // r und i bezeichnen damit das selbe int Objekt
int x=r;        //x=1;
r=2;            //i=2!!!
```

Vorsicht beim Anwenden von Operatoren auf Referenzen!

```
int ii=0;
int & rr=ii;
rr++; //erlaubt, jedoch ii wird erhöht!
```

7.2.2 Variablen-Definition an beliebiger Stelle

In C++ können Variablen nicht nur am Beginn eines Blocks `{ ... }` (z.B. einer Funktion), sondern an beliebiger Stelle (d.h. dann wenn sie gebraucht werden) definiert werden:

```
void f()
{ int ia=10;
  for(int i=0;i<ia;i++){....}

  int iend=i;
  double length,width;
   ...
  length=5.;width=2.5;

   double area=length*width;
}
```

7.2.3 Überladen von Funktionen

C: Unterschiedliche Funktionen haben auch unterschiedliche Namen

C++: Wird von einer Funktion die selbe oder eine gleichwertige Aufgabe an unterschiedlichen Objekten durchgeführt, so ist der gleiche Funktionsname sinnvoll (overloading)

```
void print(int);           //schreibt int
void print(const char*);   //schreibt string
void print(double);
void print(long);
```

Der Compiler entscheidet durch Vergleich der deklarierten und aktuellen Argumenttypen, welche Funktion beim Aufruf gemeint ist. Existieren mehrere Funktionen mit gleichen Namen, so wird die mit den am besten passenden Argumenttypen aufgerufen.
Gibt es keine passende Funktion oder ist der Aufruf mehrdeutig → Fehler beim Compilieren

```
void f(void)
{ print(1L);   //->print(long)
  print(1.0);  //->print(double)
  print(1);    //Fehler; mehrdeutig long oder double
}
```

Die genauen Regeln zum Auffinden der passenden Funktion sind kompliziert. Siehe z.B.:
Bjarne Stroustrup, "Die C++ Programmiersprache", Addison-Wesley 1992

Diese Technik ist ein wesentlicher Bestandteil und wird für die elementaren Operationen von C++ selbst genutzt.
z.B.: + Operator für `int, double, ...`

Es wird jedesmal eine andere Funktion für die jeweils passenden Typen aufgerufen:

```
   int operator+(int,int);
double operator+(double,double);

int i=2,j=3,k;
k=i+j
k=operator+(i,j);  //Aufruf von int operator+(int,int);

double x=2.4,y=3.0,z;
z=x+y;
z=operator+(x,y);  //Aufruf von double operator+(double,double);
```

Viele Operatoren können in C++ überladen werden. daraus ergibt sich, daß in C++ eine wesentlich strengere Typprüfung bei Funktionsparametern stattfindet als in C.

7.2.4 Die Operatoren new und delete

- `new`:
  Reserviert freien Speicherplatz. Die Mindestgröße wird zwischen `[]` übergeben. Zurückgeliefert wird ein Zeiger auf das Objekt. Ist zuwenig Speicher vorhanden wird `NULL` zurückgegeben. `new` initialisiert den neuen Speicherplatz nicht!

- `delete`:
  Gibt den vorher mit `new` besetzten Speicherplatz wieder frei.

Diese Operatoren sind den Funktionen `malloc` bzw. `calloc` und `free` vorzuziehen!

Achtung:
`new` und `delete` bzw. `malloc`und `free` gehören zusammen!

```
double *p = new double [100]; //reserviert Platz fuer100-doubles
int i;
for (i=0;i<100;i++)p[i]=....
...
delete []p;
// ---------------
int *pi = new int;   //reserviert Platz fuer 1 int
...
delete pi;
```

Wurde für mehrere Objekte (siehe obiges Beispiel 1.Teil) Speicher reserviert, so sind beim `delete`-Operator nach dem Zeiger die ”leeren” eckigen Klammern anzugeben. Beim reservieren für ein einzelnes Objekt ohne eckige Klammern bei `new` müssen die Klammern auch bei `delete` weggelassen werden.

7.2.5 Bereichs-Operator ”::” (*scope operator*)

Wenn eine lokale Variable den selben Namen wie eine globale Variable trägt, so kann im Gültigkeitsbereich der lokalen Variablen mit der normalen Syntax nicht auf die globale Variable zugegriffen werden Der Operator `::` ermöglicht eine eindeutige Variablenansprache. Im Klassenkonzept von C++ wird dadurch ebenfalls ein eindeutiger Zugriff auf die jeweiligen Komponenten gewährleistet.

```
int x;   // externe globale Variable x

void func1(void)
{ int x;           // lokale Variable x; die
                   // externe globale Variable x wird verdeckt
 x = 17;           // Zuweisung auf lokale Variable
::x = 18;          // Zuweisung auf globale variable
}                  // end of func1
```

7.2.6 Ein- und Ausgabefunktionen in C++

Ebenso wie in C werden auch in C++ viele grundlegende Funktionen von Bibliotheken zur Verfügung gestellt und sind nicht als Teile der Sprache definiert. Allerdings wurde bei der Standardisierung der Sprache bereits ein grundlegender Satz von Bibliotheksklassen festgelegt, der viele häufig verwendete Datenstrukturen und Methoden in C++ zur Verfügung stellt. Diese Bibliothek ist unter dem Namen STL (Standard Template Library) bekannt.
Ein Teil der STL ist den Ein- und Ausgabefunktionen gewidmet, die in der *iostream*- bzw. in verwandten Klassen definiert sind. Wenn auch in weiten Bereichen dieses Skriptums die `printf()` Notation verwendet wird, und dies im Rahmen von C++ vollkommen akzeptabel und richtig ist, so spricht doch einiges für die Verwendung der *iostream* und verwandten Bibliotheksklassen. Insbesondere die exakte Bestimmung der Datentypen beim Kompiliervorgang verhindert einige klassische C-Fehler, die immer wieder durch die Verwendung ungeeigneter Formate vorkommen.
Die theoretischen Grundlagen dieser Bibliotheken werden erst in den folgenden Kapiteln beschrieben, daher soll an dieser Stelle der C++ -eigene Ein- und Ausgabemechanismus in Form von einigen grundlegenden Beispielen beschrieben werden. Die Bezeichnung kommt von eigenen *stream*-Objekten, aus denen die einzugebenden Daten fließen, bzw. in die die Ausgabe strömt. Dieses Strömen wird durch die dazu missbrauchtenßhift-Operatoren << für die Ausgabe und >> für die Eingabe angezeigt. Das altbekannte ”Hello WorldProgramm sieht in den Grundzügen wie folgt aus:

```
#include <iostream>
...
cout << "Hallo, da bin ich!" << endl;
```

Wie üblich werden die notwendigen Definitionen in der header-Datei *iostream* eingelesen, die in diesem Fall die Standard- Ein- und Ausgabeströme `cin` und `cout` definiert. Der Text wird dann in den Ausgabestrom (der zum Terminal führt) eingeleitet. An Stelle des altbekannten Zeilenvorschubes `\n` im Text tritt nun der Manipulator `endl` (=end of line) für den Ausgangsstrom, mit demselben Effekt. Das Format wird auf Grund des Datentyps erkannt.
Das Einlesen vom Terminal erfolgt vom Strom `cin`, wobei auch hier das Format durch den Typ der eingelesenen Variablen bestimmt wird.

```
int i;
double f;
...
cout << "Bitte eine ganze Zahl: ";  // hier kein endl
cin >> i; // liest vom Standard-Eingabestrom und speichert das
          // Ergebnis auf die Variable i
cout << "und eine mit Komma: ";
cin >> f;
cout << "Kontrolle: "<< i << " und " << f << endl;
```

Die automatische Formatierung kann durch Manipulatoren wie `setprecision()` beeinflusst werden. Dazu einige Beispiele:

```
cout <<3.1415926 << endl;
cout << setprecision(4) << 3.1415926 << endl;
cout << "Nachkommastellen: "<< cout.precision()<<endl<<endl;
cout << 19 << "   Oktal: "<< oct << 19
     << "   Hex: "<< hex << 19 << endl;
```

Beim Einlesen von Zeichenketten ist zu beachten, dass diese wortweise eingelesen werden, d. h. normalerweise werden Leerzeichen vor dem Wort übersprungen, und dann das Wort bis zum nächsten Leerzeichen eingelesen. Ein Hürde dabei ist es, nicht mehr einzulesen als Platz im Programm zur Verfügung steht. Durch den `setw()` Manipulator kann man die Breite des Feldes von Ein- und Ausgabeströmen begrenzen. Zum Unterschied von den vorigen Manipulatoren gilt dieser Befehl aber immer nur für das nächste Element im Strom.

```
cin >> setw(10) >> c;
cout << "Echo 1: " << c << endl;
cout << "oder: " << setw(14) << setfill('.') << c <<endl;
cout << "Echo 2: " << c << endl;
cout.setf(ios::left);
cout << "oder: " << setw(14) << setfill('.') << c <<endl;
```

Ganz leicht kann man nun auch auf Dateien schreiben oder von ihnen lesen, indem man die Ströme `cout` und `cin` durch Ströme ersetzt, die mit den entsprechenden Dateinamen assoziiert sind. Dazu benötigt man die *fstream* Klassen (file stream).

```
#include <fstream>
...
ofstream outfile("test.dat", ios::out); // erzeugt den Strom outfile
                                        // und verknüpft ihn mit
                                        // test.dat
outfile << "das steht dann in test.dat...";
outfile.close();
```

Andere häufig verwendete flags zum Öffnen einer Datei sind ios::in (zum Einlesen von der Datei) und ios::app (hängt die Ausgabe an eine existierende Datei an.

7.3 Objektorientiertes Programmieren

Drei grundlegende Eigenschaften kennzeichnen das objektorientierte Programmieren (OOP)

1. **Kapselung:**
   Daten und Funktionen (Aktionen oder Methoden), die diese Daten manipulieren werden zusammengefaßt und in eine Klasse eingeschlossen, so daß sie nur über **Elementfunktionen** (*member functions*) erreichbar sind.

2. **Vererbung:**
   Neue **abgeleitete Klassen** (*inherited classes*) können die Daten und Funktionen aus einer oder mehreren Basisklassen übernehmen (erben). Neue Funktionen und Daten können hinzugefügt werden. Daraus erhält man eine **Klassenhierarchie**.

3. **Polymorphie:**
   Funktionen in abgeleiteten Klassen können den selben Namen haben, obwohl sie jeweils anders implementiert sind, sie werden als **virtuelle Funktionen** bezeichnet.

8 Klassen

Beim objektorientierten Programmieren werden die Daten und die Funktionen, welche die Daten manipulieren zu einer **Klasse** (*class*) zusammengefaßt. Eine Klasse sieht prinzipiell aus wie eine Struktur (*struct*), nur daß bei einer Klasse wesentlich mehr Möglichkeiten vorhanden sind. Dies soll hier anhand einer Klasse `Punkt`(`Point`) erklärt werden. Damit einfachste Manipulationen an einem Punkt in der Ebene (z.B. Bildschirmpixel) durchgeführt werden..

```
// Klassendefinition
// Meist in einer eigenen Header Datei
class Point
{private:
   int x;     //Daten der Klasse
   int y;

 public:    //Elementfunktionen, Aktionen
 //Konstruktor
   Point(const int Nx=0,const int Ny=0);

 //Weitere Funktionen
   int GetX(){return x;}
   int GetY(){return y;}
   void MoveTo(const int Nx,const int Ny);
   void Draw(void);
};
```

Die Daten und die Definitionen der Elementfunktionen, welche eine ”sichere” Manipulation der Daten gewährleisten, werden in einer **Klassendefinition** zusammengefaßt. Kurze Elementfunktionen können auch als `inline` direkt in der Definition implementiert werden. Diese Definition ist nur eine ”Vorlage” für den Compiler, damit wird noch kein eigentliche Objekt generiert.

```
// Implementation als inline oder als eigener Funktionsblock
// Kann auch in einer anderen Datei stehen
// !! Bereichsoperator :: notwendig, da Daten gekapselt!

Point::Point(const int Nx, const int Ny)
{ x=Nx;
  y=Ny;
}
void Point::MoveTo(const int Nx, const int Ny)
{x=Nx;
 y=Ny;
}
```

Längere Methoden können auch in einem eigenen Funktionsblock implementiert werden, der sich auch in einer separaten Datei befinden kann. Dabei ist darauf zu achten, daß der gesamte eindeutige Name der Methode aus dem Namen der Klasse und dem eigentlichen Funktionsbezeichner besteht, als Trennelement fungiert hier der Bereichsoperator (::).

```
// ********************************************
// Verwendung von Point:
// ********************************************

main()
{Point NullPunkt, Center(5,4);

 printf("Center: x:%d y:%d\n",
        Center.GetX() , Center.GetY());

 //Zeiger
 Point *pAny;
 pAny=&NullPunkt;

 pAny->MoveTo(12,0);
 ....
 //Arrays
 Point Poly[4]={{0,0},{5,0},{1,6},{6,6}};

 for(int i=0;i<4;i++)Poly[i].Draw();

 //Dynamisch
 pAny=new Point(12,13);
 pAny->Draw();
....
 delete pAny;

 //Kapselung
int i = Center.x    // Fehler
                    // Nur Mitglieder (members) der Klasse
                    // Haben Zugriff auf Daten (private)
                    // Zugriff nur ueber Memberfunctions
                    // (public) moeglich

int i = Center.GetX(); //OK
}
```

Die Verwendung ist analog zu einer gewöhnlichen Variablen, sie besteht aus dem Namen der Klasse, gefolgt von einem gültigen Bezeichner (Variablennamen). Erst durch diese explizite Erstellung des Klassenobjekt wird Speicher belegt.

Wie bei einer Struktur (`struct`) erhalten wir durch die Operatoren `.` und `->` Zugriff auf die Klassenelemente. Elementfunktionen können mit diesen Operatoren, ausschließlich für eine Variable des entsprechenden Typs, aufgerufen werden.
Ziel beim OOP ist es die Daten der jeweiligen Klassen nur mit den ”sicheren” Elementfunktionen, die im `public`-Bereich der Klasse stehen, zu verändern. Nur Elementfunktionen haben Zugriff auf die Daten, die im `private`-Bereich der Klasse stehen.
Die objektorientierte Denkweise unterscheidet sich vom prozeduralen Programmieren:
Der Funktionsname ist eine **Botschaft** (*message*), die dem manipulierten Objekt, dem **Empfänger** (*receiver*) übermittelt wird. Die **Daten der Botschaft** (*message data*) sind die dabei übergebenen Funktionsargumente.

```
    Center  ->  MoveTo    (6,10)
Empfänger       Botschaft Daten
```

Eine andere Sprechweise ist:
Die Methode `MoveTo` wird mit den Daten `(6,10)` auf das Objekt `Center` angewendet.
Unter ”sicheren” Elementfunktionen versteht man, daß der Programmierer in der Funktion geeignete Vorkehrungen trifft, daß stets eine Konsistenz der Objektdaten im `private`-Bereich der Klasse gewährleistet wird.

z.B. Klasse `Point` Methode `MoveTo`:
Falls durch die Daten der Punkt auf eine Stelle außerhalb des gültigen Bereichs (minimale/maximale Bildschirmkoordinaten) bewegt wird, so sind geeignete Maßnahmen zu treffen, wie etwa eine Fehlermeldung oder zyklische Bewegung des Punktes.

Eine Klasse (*class*) ist einer Struktur (*struct*) weitgehend gleichzusetzen, es gibt jedoch einen zusätzlichen Zugriffsschutz:

1. `private`:
   Daten können nur über Elementfunktionen manipuliert werden. Funktionen können nur innerhalb von anderen Elementfunktionen aufgerufen werden.

2. `public`:
   Auf Daten und Funktionen kann von überall zugegriffen werden. Daten und Methoden im `public`-Bereich werden wie in einer Struktur behandelt.

3. `protected`:
   Daten und Funktionen sind `public` für Zugriffe aus abgeleiteten Klassen und `private` für alle anderen Zugriffe

4. `friend`:
   Gehört eine Funktion nicht zu einer Klasse und soll sie dennoch Zugriff auf `private`-Elemente der Klasse erhalten, so muß sie als `friend` deklariert werden. Auch andere Klassen können als `friend` deklariert werden

Ein Zeiger auf ”sich selbst”:
Die Adresse des Empfängers ist innerhalb einer Elementfunktion durch die Zeiger-Variable `this` zugänglich.

```
class Point
{private:
   int x,y;

 public:
   int GetX(){return this->x;}
};
```

Auf alle Klassenelemente innerhalb der Klasse mit Hilfe des `this`-Zeigers nach dem Schema `this->`*Variablenname* zugegriffen werden, dies ist normalerweise bei ”gewöhnlichen” Klassenelementen nicht notwendig, da sie auch durch den Variablennamen alleine eindeutig angesprochen werden können. Eine Verwendung dieser Schreibweise bei direkter Manipulation von Zeigern als Klassenelemente ist jedoch manchmal erforderlich.

8.1 Konstruktoren

Konstruktoren sind spezielle Elementfunktionen, die beim Initialisieren von Klassenobjekten implizit (automatisch) aufgerufen werden. Anhand der Beispielklasse `String` soll dies erklärt werden. Eine vollständige Klasse ”string” ist jedem C++-System meist als template-Klasse in einer Bibliothek (*library*) beigefügt. Mit Hilfe einer geeigneten C++Klasse können die ”Unzulänglichkeiten” von Strings (Arrays) (Indexprüfung, Aneinanderfügen, ...) beseitigt werden. Dies geschieht jedoch auf Kosten der Laufzeit.

Beispielklasse ”String”:

```
class String
{  char *Buf;
   int iLen;
 public:
   String(void)     {Buf=NULL; iLen=0;}
   String(const String &s)
   String(const char *st);
   String(const char *s1, const char *s2);
   ~String(void)    {delete [] Buf;}
};
String::String(const String &s)
 {Buf = new char [s.iLen+1];
  strcpy(Buf,s.Buf);
  iLen=s.iLen;
 }
```

```
String::String(const char *st)
 { Buf = new char [strlen(st)+1];
  strcpy(Buf,st);
  iLen=strlen(st);
 }
```

Jede Klasse besitzt spezielle Elementfunktionen.
Eine davon ist der Konstruktor, mit folgenden Eigenschaften:

- Er hat den selben Namen wie die zugehörige Klasse.
- Er kann keinen Wert zurückgeben, jedoch beliebige Parameter übernehmen.
- Er wird automatisch aufgerufen, wenn ein Objekt der Klasse erzeugt wird und ermöglicht damit eine ”sichere” Initialisierung der Klassenelemente.
- Eine Klasse kann mehrere Konstruktoren besitzen.
- Eine Klasse muß keinen Konstruktor haben.
- Er kann nicht direkt aufgerufen werden, sondern bei der Initialisierung von Klassenobjekten wird automatisch, gemäß den Regeln zum Überladen von Funktionen, der passende Konstruktor aufgerufen.
- Er ist meist `public`, damit auch ”fremde” Funktionen Objekte erzeugen können.

8.1.1 Der ”default”-Konstruktor

```
String(void) {Buf=NULL; iLen=0;}
```

Der ”default”-Konstruktor besitzt kein Argument und wird bei folgenden Deklarationen aufgerufen:

```
String S;
String *pS = new String;
```

Die Verwendung des Operators `new` bewirkt einen automatischen Aufruf des entsprechenden Konstruktors! Das unterscheidet ihn von den Funktionen `malloc()` bzw. `calloc()`.

```
String *pR = new String[10];
```

Der Konstruktor wird 10mal aufgerufen, jedesmal für ein Arrayelement.

Dynamische Speicherplatz Zuweisung in C++:

`new` und `delete` verwenden! `new belegt` nicht nur Speicherplatz (wie z.B. `malloc()`), sondern es initialisiert Speicher durch Aufruf des Konstruktors.

Wird kein ”default”-Konstruktor definiert, so generiert ihn der Compiler. Speicher wird reserviert und nach den Regeln von C initialisiert (static, global=0, local=?)

8.1.2 Der Kopier-Konstruktor

```
String::String(const String &s)
{ Buf = new char [s.iLen+1];
  strcpy(Buf,s.Buf);
  iLen=s.iLen;
}
```

Man erkennt den Kopier-Konstruktor daran, daß sein Argument ist eine Referenz (`&`) auf ein Objekt der eigenen Klasse ist. Er wird aufgerufen, wenn ein Klassenobjekt durch ein anderes initialisiert wird:

```
String s1;         // default Konstruktor
String s2 = s1;    // Kopier-Konstruktor
```

Er wird weiters (implizit vom Compiler) aufgerufen, wenn ein Klassenobjekt als Wert an eine Funktion übergeben wird, um den Funktionsparameter auf den Stack zu kopieren.
Wird kein Kopier-Konstruktor definiert, so verwendet der Compiler `memcpy()`. Bei Klassen die Zeiger enthalten um z.B. Felder zu verwalten, verursacht dies meist ein sogenanntes ”shallow copy” (seichte Kopie). D.h. es werden nicht alle Werte des Feldes sondern nur die Zeiger kopiert.

```
String s1="Ein String";
String s2 = s1;
```

Ist kein Kopier-Konstruktor definiert, so ergibt sich folgende Speicherbelegung:

Adresse	Name	Wert
0x0000	s1	-
0x0000	Buf	0x1000
0x0004	iLen	10
0x0008	s2	-
0x0008	Buf	0x1000
0x000C	iLen	10
...	...	...
0x1000	-	"Ein String\0"

Dieses Ergebnis wäre sicher unerwünscht, da nur die Zeiger (`Buf`) kopiert werden und damit alle Änderungen von `s1.Buf` sich auf `s2.Buf` auswirken (gleiche Zeiger). Die (Neu)Definition des Kopier-Konstruktors bewirkt folgendes:

Adresse	Name	Wert
0x0000	s1	-
0x0000	Buf	0x1000
0x0004	iLen	10
0x0008	s2	-
0x0008	Buf	0x2000
0x000C	iLen	10
...	...	...
0x1000	-	"Ein String\0"
0x2000	-	"Ein String\0"

Damit werden die Strings tatsächlich kopiert. Für den ”neuen” String wird ein neuer Zeiger für den Beginn des Feldes `Buf` bei einer neuen Speicheradresse (hier 0x2000) angelegt und jedes Element von `Buf` dorthin kopiert. Dieser Vorgang wird als ”deep copy” bezeichnet.

8.1.3 Der Konversions-Konstruktor

```
String(const char *st)
```

Das Argument eines Konversions-Konstruktors ist ist ein Objekt eines anderen Datentyps (hier z.B. char *). Er wird bei folgenden Deklarationen aufgerufen:

```
String s1 = "initialized";
String s2("initialized");
```

Dabei findet eine Typ-Konversion von `char`-Array in `String` statt.

```
// Funktion mit einem String Arg
void f(String s);

char szB[128] = gets();
// Input mit gets()

f( String(szB) );  //Konversionskonstruktor
f( (String) szB ); //Konversionskonstruktor
```

Für den Aufruf von `f()` muß `szB` in ein `String`-Objekt konvertiert (cast) werden. Der Compiler verwendet dazu den `char*`-Konstruktor, da dies der passende Datentyp ist. Auch bei der direkten Typkonversion `(String) "Zeichen Kette"` wird dieser Konstruktor implizit verwendet.
Es können natürlich beliebige Konversions-Konstruktoren, für alle möglichen Typumwandlungen definiert werden. Z.B.:

```
String(int);   // Umwandlung einer int-Zahl
               // in den entsprechenden String
```

8.1.4 Andere Konstruktoren

Es sind weitere Konstruktoren mit beliebigen Argumenten möglich, die bei Bedarf aufgerufen werden:
z.B.

```
String(const char *s1, const char *s2);
```

bei

```
String s1("Teil1", "Teil2")
```

8.1.5 Weitere Bemerkungen

Die Deklaration eines Zeigers auf ein Objekt einer Klasse ruft noch keinen Konstruktor auf.

```
String *pS;     // noch kein Konstruktor
pS=new String("New String");    //erst hier
```

ist äquivalent zu:

```
String *pS = new String("New String");
```

Es können auch Felder von Objekten initialisiert werden. Der entsprechende Konstruktor wird für jedes Element aufgerufen.

```
String StArray[10];
String *p=new String[10];

String s = "Ein String";
String Array[]=
    {String(),              // Default Konstruktor
    "Hallo"                 // char * Konstruktor
    String("Freunde"),      // char * Konstruktor
    s,                      // Kopier Konstruktor
    String(s),              // Kopier Konstruktor
    String("Noch", "zwei") // 2-Arg Konstruktor
};
```

Vorsicht:

```
String Kurt();
```

deklariert keinen String, sondern eine Funktion mit String als Rückgabewert:

```
String Kurt(void);
```

8.2 Der Destruktor

Der Destruktor hat den selben Namen wie die Klasse, nur mit ~ vorangestellt. Er ist das Komplement zum Konstruktor, d.h. er wird automatisch aufgerufen, wenn das Objekt den Gültigkeitsbereich verläßt. Dies ist der Fall wenn der entsprechende Programmblock durch } abgeschlossen wird, oder wenn nach dem vorherigen Aufruf von `new`, durch den Operator `delete` Speicher wieder freigegeben werden soll.
Mit dem Destruktor wird das Objekt deinitialisiert, d.h. der "Arbeitsplatz kann so wieder aufgeräumt werden, wie er vorgefunden wurde".

z.B.:

```
~String(void) { delete [] Buf; }
```

Wie der Konstruktor, kann auch der Destruktor nicht direkt aufgerufen werden. Er kann auch keinen Rückgabewert und kein Argument haben. Die letzte Eigenschaft ist "leider" eine Einschränkung, aber durch die Sprachdefiniton fix vorgegeben.

8.3 Spezielle Klassenelemente

8.3.1 static-Datenelemente einer Klasse

In einer Klasse dürfen auch `static`-Datenelemente verwendet werden. Sie sind dann für alle Instanzen einer Klasse gleich.
Typische Anwendung:
Instanzenzähler (Zählt wie viele Objekte einer Klasse generiert wurden)

```
class Window
{private:
 ...
 public:
 static int NumOfObj;

 void InitVideoSystem(void);
 void CloseVideoSystem(void);

Window(void)
 {if(++NumOfObj==1) InitVideoSystem(); }

~Window(void)
 {if((NumOfObj--==1) CloseVideoSystem();
...
};

#ifdef WINDOW_H_COMP
int Window::NumOfObj=0;
#endif WINDOW_H_COMP
```

`static`-Klassenelemente werden wie normale globale Variable behandelt. Die explizite Initialisierung darf nur einmal compiliert werden.
Dies wird im vorigen Beispiel durch die Makros `#ifdef, #endif` bewirkt.
Ist der vorherige Code in z.B. in der Datei `window.h` untergebracht, dann müssen in genau einem Programm-Modul folgende Zeilen stehen:

```
#define WINDOW_H_COMP 1
#include "window.h"
```

in allen anderen Modulen:

```
#include "window.h"
```

Auch wenn kein Objekt der Klasse existiert, kann auf `static-public` Daten (auch über Nicht-Elementfunktionen) von außen zugegriffen werden.

```
void main()
{ ...
   if(Window::NumOfObj!=0)
     {prinf("Fehler: Fenster offen\n");
      exit(1);}
   exit(0);

 Window W;
}
```

Der Bereichsoperator "::" ist notwendig, da verschiedene Klassen statische Variablen mit gleichen Namen besitzen können. Statische Daten können auch als private deklariert werden, dann ist nur noch der Zugriff über Elementfunktionen möglich:

```
class Window
{private:
...
  static int NumOfObj;
 public:
  Window(){/*...*/}
  ~Window(){/*...*/}
  static int NumWindows(void) {return NumOfObj;}
...
};
```

Auch Elementfunktionen können als `static` deklariert werden. Nur sie haben Zugriff auf statische Klassenelemente. Sie dürfen nicht als `inline` deklariert werden. Sie können auch aufgerufen werden, wenn explizit kein Objekt der Klasse existiert.

```
main()
{if(Window::NumWindows()!=0)
    {......}
 Window *w = new Window;
}
```

Statische Klassenelemente haben keinen this-Zeiger!

8.3.2 const-Datenelemente

Klassen können auch Datenelemente mit dem `const`-Modifizierer enthalten. Sie verhalten sich wie normale Datenelemente, können jedoch nicht verändert werden.

```
class Window
{private:
  ...
  const int Color;
 public:
  Window(int InitC){/*...*/}
  ...
};
```

Es gibt jedoch Probleme bei der Initialisierung solcher Klassenelemente:
Folgender Konstruktor wird nicht compiliert, da auf ein `const`-Objekt geschrieben wird.

```
Window::Window(int InitC)
{Color=InitC; //Fehler!!!
}
```

Die Initialisierung `const`-Klassenelementen ist nur mittels **Initialisierungsliste** möglich. Dies ist eine spezielle Syntax um Klassenelemente zu initialisieren:

```
Window::Window(int InitC) : Color(InitC)
{ ... }
```

Dabei wird nach der Kopfzeile des Konstruktors ein Doppelpunkt : eingefügt, danach folgt die Initialisierungsliste in der Form *Variablenname(Anfangswert)*. Es sind auch mehrere, durch Kommas getrennte, Initialisierungen möglich.
Die Implementation des Konstruktors `{ }` muß vorhanden sein; er kann jedoch leer sein.

```
class IntArray
{ int *Array;
  int ActSize;
  static const int DefaultSize;

 public:
  IntArray(void) : ActSize(DefaultSize), Array(new int [DefaultSize])
   { }
};
const IntArray::DefaultSize=GetSize();
```

`GetSize()` liest die Größe des Arrays z.B. aus einer Konfigurationsdatei oder von der Kommandozeile.
Ist die Konstante bereits zur Compilier-Zeit bekannt, kann die Initialisierung gemäß folgendem Beispiel erfolgen:

```
class IntArray
{enum {DEF_SIZE=128};
 int *Array;
 int ActSize;

 public:
  IntArray(void) : ActSize(DEF_SIZE),
                   Array(new int [DEF_SIZE]){ }
};
```

Da das "`enum`" lokal und "`private`" ist, haben nur Elementfunktionen Zugriff.
Ein `#define DEF_SIZE 128` wäre zwar auch möglich, doch dann wäre `DEF_SIZE` im gesamten Programm zugänglich und nicht `private`.

8.3.3 const-Objekte einer Klasse

Es ist auch möglich ein `const`-Objekt einer Klasse zu erstellen:

```
const String S="Eine Konstante";
```

Dabei ist jedoch folgendes zu beachten:
Der Compilier muß nun wissen welche Elementfunktionen ”sicher” sind, damit er sie auf ein `const`-Objekt der eigenen Klasse anwenden darf. Er muß die ”sicheren” Methoden, das sind jene, die ein Klassenelement nicht verändern (nur lesen) von den ”unsicheren” Funktionen, die das Objekt verändern, unterscheiden können.
z.B. Beispielklasse `String`:

```
S.GetLength();
S.print();
```

sollen ”sichere” Elementfunktionen sein, die **keine** Änderungen an den Klassenelementen vornehmen.

```
S.makeupper();
```

soll eine ”unsichere” Methode sein, die alle Kleinbuchstaben des Strings in Großbuchstaben umwandelt und damit Veränderungen vornimmt.
Die Unterscheidung erfolgt durch anfügen des Schlüsselwortes `const` an die Funktionsdefinition und den Funktionskopf.

```
class String
{  char *Buf;
   int iLen;
 public:
   String (char *s)
          {Buf=new char [iLen=strlen(s)+1];
           strcpy(s,Buf);
          }
  ~String(void)
      {delete [] Buf;}

   void print(void) const; // SICHER
   void print(void);       // UNSICHER
   void makeupper();       // UNSICHER
int GetLength(void) const {return iLen;} // SICHER
};
String::print(void) const  // SICHER
 {printf(%s (CONSTANT)\n",Buf);}

String::print(void)       // UNSICHER
 {printf(%s (VARIABLE)\n",Buf);}
```

Gemäß den Richtlinien zum Überladen vom Funktionen ist es sogar möglich zwei Funktionen mit dem gleichen Namen zu definieren, die sich sonst nur durch das `const`-Prädikat unterscheiden.

```
main()
{  const String c = "Fest";
   String v = "Veraenderlich";

   c.print(); //OK const print wird verwendet
   v.print(); //OK print wird verwendet
   c.makeupper(); //FEHLER c ist const
   v.makeupper(); //OK
}
```

Ausgabe:

```
Fest(CONSTANT)
Veraenderlich (VARIABLE)
```

Im obigen Beispiel wird auf das ”konstante” `String`-Objekt `c` dann die `print`-Methode mit dem `const`-Prädikat angewendet. Auf das `String`-Objekt mit dem Namen `v` die ”normale” `print`-Methode angewendet.

8.4 Beziehungen zwischen Klassen

Es gibt 3 Möglichkeiten wie man Klassen kombinieren kann:

1. `friend`:
   Eine Klasse erhält Zugriff auf die private-Daten einer anderen.
2. Das Objekt einer Klasse ist Element einer anderen Klasse
3. Abgeleitetete Klasse:
   Man erhält eine neue Klasse durch Erweiterung einer anderen Klasse.

8.4.1 Friend

Eine Klasse erhält Zugriff auf die private-Daten einer anderen durch das Schlüsselwort `friend`.
Beispiel:
Verwalten einer verketteten Liste, durch aufteilen in 2 Klassen:

1. Listenelement mit Knoten + Daten
2. Manipulation der Listen

Durch die Zeile `friend class Liste;` erhält die Klasse `Liste` Zugriff auf alle `private`-Elemente der Klasse `Element` und kann damit die Liste durch direkte Manipulation des Zeigers `Next` und von `iWert` die Liste verwalten.

```
class Liste;  //Vorausdeklaration
class Element
{ friend class Liste;
  private:
    int iWert;
    Element *Next;

    Element(int v){iWert=v; Next=0;}
};

class Liste
{ Element *Kopf;
  Element *AnsEnde(void);
 public:
  Liste(int v){Kopf=new Element(v);}
  Liste(){Kopf=0;}
  int Anzeigen(void);
  void Einfuegen(int v=0);
  void Anhaengen(int v=0);
  int IstLeer(void){return Kopf==0;}
  ~Liste();
};

Liste::~Liste()
{Element *tmp, *Z=Kopf;
 while(Z)
   {tmp=Z;
    Z=Z->Next;
    delete tmp;
   }
}

void Liste::Einfuegen(int v)
{// Am Anfang einfuegen
 Element *Z = new Element(v);
 Z->Next = Kopf;
 Kopf = Z;
}
```

```
void Liste::Anhaengen(int v)
{// Ans Ende anhaengen
  Element *Z = new Element(v);
  if(Kopf == 0)Kopf=Z;
  else
    {Element *Vorg, *Akt;
     for(Vorg=Akt=Kopf; Akt; Akt=Akt->Next)Vorg=Akt;
     Vorg->Next = Z;
    }
}

int Liste::Anzeigen(void)
{if (Kopf == 0)
     {printf("Liste leer\n!");
      return 0;
     }
 int i=0;
 Element *Z=Kopf;
 while(Z)
   {i++;
    printf("Element %d: %d\n",i,Z->iWert);
    Z=Z->Next;
   }
 return i;
}

//Testen
main ()
{Liste IL;  //Leere Liste
 if(IL.IstLeer()) printf("Tatsaechlich leer\n");

 IL.Anzeigen();

 for(int i=0; i<10; i++) IL.Anhaengen(i*2);

 IL.Anzeigen();
 IL.Einfuegen(100);
 IL.Anzeigen();
}
```

8.4.2 Eine Klasse ist Element einer anderen Klasse

Als einfaches Beispiel sollen die Angestellten eines Betriebs verwaltet werden. Als relevante Größen sind der Name (`String`), das Gehalt (`double`) und das Alter (`unsigned`) in die Klasse aufzunehmen.
Es gibt einen einfachen Test um zu entscheiden, ob die Klasse Element der ursprünglichen Klasse sein soll, oder ob die neue Klasse durch Vererbung aus der Ausgangsklasse erhalten werden soll:
Trifft das "hat-ein" Attribut zu, so ist die Klasse ein Element der der Ausgangsklasse.
Der Angestellte **hat einen** Namen →
Der "*Name*" (mit Datentyp `String`) ist Element der Klasse.
Daraus ergibt sich folgende Definition für den "Angestellten":

```
class Angestellter
{  String Name; // "hat einen" Namen
   double Gehalt;
 unsigned Alter;
 public:
   double ZeigeGehalt(void){return Gehalt;}
     void print(void) const ;
};
```

8.4.3 Die abgeleitetete Klasse

Man erhält eine neue Klasse durch Erweiterung einer ursprünglichen Klasse.
Man nennt die ursprüngliche Klasse **Basisklasse** und die neue Klasse **abgeleitete Klasse**. Diese Vorgehensweise wird als **Vererbung** (*inheritance*) bezeichnet.
In dem vorher begonnenen Beispiel sollen die Angestellten auch kategorisiert werden. Es soll z.B. ein Abteilungsleiter erfaßt werden können. Er soll die selben Eigenschaften wie ein Angestellter haben, nur soll zusätzlich sein Umsatz erfaßt werden.
Für den Test von vorher gilt:
Trifft das "ist ein" Attribut zu, so ist die Klasse eine abgeleitete Klasse der ursprünglichen Klasse.
Ein Abteilungsleiter **ist ein** Angestellter →
Der "*Abteilungsleiter*" wird von `Angestellter` abgeleitet.

```
   // Basisklasse:
class Angestellter
{    String Name;
     double Gehalt;
   unsigned Alter;
 public:
   double ZeigeGehalt(void){return Gehalt;}
     void print(void) const ;
};
```

Die neue, abgeleitete Klasse erbt die Eigenschaften der Basisklasse. Dabei wird die Kopfzeile der neuen Klasse wie gewohnt angegeben, durch einen Doppelpunkt ":" getrennt, folgt nach dem Schlüsselwort `public` der Name der Basisklasse.

```
  // Abgeleitete Klasse:
class Verkaufsleiter : public Angestellter
{                   //Name, Alter, ... geerbt
   double Umsatz;  //Zusatzliche Elemente
 public:
   void print(void) const ;
};
```

Außer den Konstruktoren, Destruktoren und dem Operator= werden alle `public` und `protected` Datenelemente und Funktionen an die abgeleitete Klasse vererbt.
Der `this`-Zeiger der abgeleiteten Klasse und der Basisklasse sind identisch!
Die `public` Methoden der Basisklasse können auch für die abgeleitete Klasse aufgerufen werden:

```
  Angestellter A;
Verkaufsleiter V;
printf("Gehalt Angest.: %f  Gehalt Verkaufsl.: %f\n",
                     A.ZeigeGehalt(), V.ZeigeGehalt());
```

Ein Zeiger auf die abgeleitete Klasse kann auf einen Zeiger der Basisklasse ohne Typumwandlung zugewiesen werden, aber nicht umgekehrt!

```
  Angestellter A, *pA;
Verkaufsleiter V, *pV;

pA = &A;  // OK
pV = &A;  // Fehler!
pA = &V;  // OK abgeleitete Klasse
pV = &V;  // OK
```

Beim Zugriff über Zeiger können Objekte einer abgeleiteten Klasse als Objekt der Basisklasse behandelt werden:

```
f(Angestellter *);
g(Verkaufsleiter *);

main()
{Verkaufsleiter V;
   Angestellter A;
 f(&V); //OK
        //Stille Konversion zum Zeiger auf Basisklasse

 g(&A); //FEHLER beim Compilieren
}
```

Abgeleitete Klasse hat jedoch kein Recht auf Zugriff zu den `private` Daten der Basisklasse:

```
void Verkaufsleiter::print() const
{ printf("Mein Name:");
  Name.print(); //FEHLER: Name ist private
}
```

Eine richtige Lösungsmöglichkeit wäre Aufruf von `Angestellter::print()`

```
void Verkaufsleiter::print() const
{ Angestellter::print();
  // jetzt zusätzliche Angaben drucken
}
```

Ganz fatal wäre:

```
void Verkaufsleiter::print() const
{ print();
  // jetzt zusätzliche Angaben drucken
}
```

Dies verursacht einen endlosen rekursiven Aufruf von:

```
Verkaufsleiter::print()
```

Man kann auch für ein Objekt der abgeleiteten Klasse einen Aufruf einer Basisklassen-Methode erzwingen:

```
main()
{Angestellter A;
 Verkaufsleiter V;

 A.print();
 V.print();
 V.Angestellter::print();  // ruft print der Basisklasse
}
```

Initialisierung und Zuweisung bei abgeleiteten Klassen

Bei Klassen die andere Klassen als Elemente beinhalten, sind bei der Initialisierung (Konstruktoren, Zuweisungsoperator) folgende Richtlinien zu beachten:

- Der Kopier-Konstruktor soll eine Initialisierungsliste mit allen Elementen besitzen
- Der Zuweisungsoperator (`operator=`) soll alle Elemente explizit auf die entsprechenden Zielobjekte kopieren

Beispiel:

```
class Angestellter
{  String Name;
   double Gehalt;
   unsigned Alter;
 public:
 Angestellter(char *N, double G, short A) :
      Name(N), Gehalt(G), Alter(A) {}
// Kopierkonstruktor
 Angestellter(Angestellter &R) :
      Name(R.Name), Gehalt(R.Gehalt), Alter(R.Alter) {}
// Zuweisungsoperator
 Angestellter &operator=(Angestellter &R)
  {  Alter=R.Alter;
      Name=R.Name;
    Gehalt=R.Gehalt;
   return *this;
  }
...// Weitere Methoden
};
```

Ebenso gelten folgende Regeln für abgeleitete Klassen:

- Der Kopier-Konstruktor soll alle Kopier-Konstruktoren der Basisklassen in der Initialisierungsliste aufrufen.
- Der Zuweisungsoperator(`operator=()`) soll alle Zuweisungsoperatoren der Basisklasse mit einem cast

  ```
  (*(base*)this)=src
  ```

  aufrufen. Ansonsten werden die Datenelemente der Basisklassen nicht korrekt kopiert (”shallow copy”).

Beispiel:

```
class Verkaufsleiter: public Angestellter
{ double Umsatz;
  String Abteilung;
 public:
  Verkaufsleiter(char *N, double G, short A, double U, char *Abt):
                 Angestellter(N,G,A), Umsatz(U), Abteilung(Abt){ }
// Kopierkonstruktor
  Verkaufsleiter(Verkaufsleiter &V) :
      Umsatz(V.Umsatz), Abteilung(Abt),
      Angestellter (V) { }

// Zuweisungsoperator
Verkaufsleiter &operator=(Verkaufsleiter &V)
 {    Umsatz=V.Umsatz;
   Abteilung=V.Abteilung;
   *((Angestellter*)this = V;
   return *this;
 }
};
```

8.4.4 Mehrfachvererbung

Es können prinzipiell auch 2 Basisklassen an eine abgeleitete Klasse vererbt werden. In der Praxis kommt diese Vorgehensweise selten vor. Es treten dabei oft mehrdeutige Funktionsaufrufe auf, die anhand des folgenden Beispiels erklärt werden sollen. Es gibt zwei Basisklassen: `Fruit` und `Tree`, die abgeleitete Klasse `AppleTree` erbt die Eigenschaften sowohl von `Fruit` als auch von `Tree`.

```
class Fruit
{ public:
  void identify(void) { printf("fruit\n"); }
};
// ------------------
class Tree
{ public:
  void identify(void) { printf("tree\n"); }
};
// ------------------
class AppleTree : public Fruit, public Tree
{};
```

Wird nun ein Objekt des Typs `AppleTree` erstellt, und die Methode `identify()` aufgerufen, so kann der Compiler nicht herausfinden, ob `Fruit::identify()` oder `Tree::identify()` gemeint ist →
Mehrdeutigkeit

```
AppleTree *Granny = new AppleTree;
Granny->identify();   // ERROR!
```

Der Programmierer muß nun explizit angeben, welche der Methoden er wünscht.

```
Granny->Fruit::identify();    // OK
Granny->Tree::identify();     // auch OK
```

Es ist natürlich auch möglich und naheliegend, in der abgeleiteten Klasse (`AppleTree`) eine neue Methode mit gleichen Namen `AppleTree::identify()` zu definieren und dann diese zu verwenden

```
AppleTree::identify(void)
{Fruit::identify();
 Tree::identify();
}
```

8.4.5 Virtuelle Funktionen (Polymorphismus)

Werden aus einer Basisklasse mehrere Klassen abgeleitet, so können in den jeweiligen abgeleiteten Klassen, Methoden mit dem gleichen Namen definiert werden. Bei der Verwendung von Zeigern kann der Compiler nicht herausfinden welche der Methoden gemeint ist. Diese Problematik, soll anhand eines Beispiels gezeigt werden.
Es gibt eine Basisklasse `fruit` mit der Methode `identify()`. Die beiden neuen Klassen `apple` und `orange` werden beide von `fruit` abgeleitet und haben beide eine Methode `identify()`.

```
class fruit
{public:
  char *identify(void) { return "fruit "; }
};

class apple : public fruit
{public:
  char *identify(void) { return "apple "; }
};

class orange : public fruit
{public:
  char *identify(void) { return "orange "; }
};
```

Bisher kein Problem. Generiert man einzelne Objekte ohne Zeiger, arbeitet alles bestens:

```
main()
{ fruit F;
  apple A;
 orange O;

 printf("%s %s %s\n",F.identify(), A.identify(), O.identify());
```

Das erwartete Ergebnis ist:
`fruit apple orange`

Kommt nun ein Programmierer auf die Idee mit Zeigern zu arbeiten und möchte er gerne den obigen Merksatz verwenden:
Ein Zeiger auf die abgeleitete Klasse kann auf einen Zeiger der Basisklasse zugewiesen werden.

Dann gibt es wie folgendes Beispiel zeigt ein Problem:

```
main()
{fruit * obst[3];  //Drei Basisklassenzeiger

 obst[0] = new fruit;
 obst[1] = new apple;  //Abgeleitete Klasse
 obst[2] = new orange; //Abgeleitete Klasse

 for (int i=0; i<3; i++)
 printf("%s ", obst[i]->identify());
}
```

Das für den Anfänger vielleicht unerwartete Ergebnis:

```
fruit fruit fruit
```

Der Grund liegt darin, daß alle `obst[i]` vom Typ `fruit *` sind. Zur Compilier Zeit steht der der Variablentyp fest → **"early binding"**

Es wäre wünschenswert, wenn jedesmal die "richtige" Methode aufgerufen würde.

Lösung: **Virtuelle Funktionen**

Nach einer kleinen, aber wirkungsvollen Änderung der in der Basisklasse `fruit` erhält man den gewünschten Effekt. Man schreibt einfach in der Basisklasse das Schlüsselwort `virtual` vor die jeweilige Methode.

```
class fruit
{public:
  virtual char *identify() { return "fruit";}
};
```

Das Ergebnis lautet nun:

```
fruit apple orange
```

Eine **virtuelle Funktion** erlaubt abgeleiteten Klassen die Bereitstellung einer modifizierter Version dieser Funktion.
Durch die Deklaration in der Klasse `fruit` sind auch alle Methoden `identify()` in den abgeleiteten Klassen implizit virtuell, wenn sie die gleiche Signatur (Parameterliste, Rückgabetyp) besitzen.
Man spricht von **"late binding"**:
Der Typ der Zeigervariablen wird erst zur Laufzeit ermittelt!

Sonderfälle:
Konstruktoren können nicht virtuell definiert werden!
Destruktoren sollen in vielen Fällen als virtuell definiert werden, da sonst in der abgeleiteten Klasse der falsche Destruktor aufgerufen wird.

```
class base
{ public: base() {...}
         ~base() {...}
};

class derived : public base
{ public: derived(){...}
         ~derived(){...}
};

base *p;
p=new derived;  //Konstruktor base und derived
delete p;      // Ruft base::~base()
```

In der letzten Zeile des obigen Beispiels, wird bei `delete p;` nur der Destruktor der Basisklasse `base` aufgerufen, da der Zeiger `p` vom Typ `base` ist. Eventuelle Speicherfreigaben im Destruktor von `derived` bleiben unberücksichtigt.
Besser ist es den Destruktor in der Basisklasse als ”virtuell” zu definieren, dann wird bei Zeigerzuweisungen Basisklasse - abgeleitete Klasse, immer der ”richtige” Destruktor aufgerufen.

```
class base
{ public:
          base() {...}
 virtual ~base() {...}
}

class derived : public base
{ public:  derived(){...}
         ~derived(){...}
};
```

8.4.6 Abstrakte Basisklassen

Es ist möglich eine Klasse bereitzustellen, ohne daß jemals ein Objekt davon erstellt werden kann. Zweck dieser Klasse ist es, abgeleitete Klassen zu bilden und die entsprechende Funktionalität dort einzubauen. Solche Klassen enthalten oft nur wenige oder keine Datenelemente sowie Definitionen von virtuellen Funktionen, die dann erst in den abgeleiteten Klassen implementiert werden müssen. Anhand des Beispielklasse `Figur` soll gezeigt werden, wie man von dieser abstrakten Basisklasse ausgehend, praktisch beliebig viele ”geometrische Figuren”, als abgeleitete Klassen definieren kann. Für jede dieser ”Figuren” muß nun eine eigene Methode zur Flächenberechnung implementiert werden.
Die abstrakte Basisklasse `Figur` verfügt nur über ein Datenelement, das ihren Namen (Rechteck, Kreis, ...) enthält. Weiters ist im `protected`-Bereich ein Konstruktor definiert, damit ist es nicht möglich ein *Figur*-Objekt direkt zu erstellen. Im `public`-Bereich ist eine virtuelle Methode `Flaeche`, zur Berechnung des Flächeninhalts,

definiert. Das "*=0;*" am Ende dieser Definition bedeutet, daß diese Methode in der Basisklasse nicht implementiert wird, aber in jeder davon abgeleiteten Klasse eine solche Methode implementiert werden **muß**. Eine solche Methode wird als **rein virtuell** bezeichnet und ist nur zum Überladen deklariert. Die Methode `Identify()` dient lediglich zur Übergabe des Namens und `ShowObj()` zur Ausgabe von Name und Fläche.
Von solchen Klassen können keine Instanzen gebildet werden, sie sind nur zur Ableitung weiterer Klassen gedacht.

Abstrakt:	Abgeleitet:
Figur	Quadrat, Rechteck, ...

```
class Figur
{ char Name[80];
protected:
  Figur(char * n){strncpy(Name,n,79);}
public:
  virtual double Flaeche(void)=0;
  const char * Identify(void){return &Name[0];}

};
void ShowObj(Figur *f);
void ShowObj(Figur *f)
{ printf("Ich bin ein %s und meine Flaeche ist %f\n",
        f->Identify(),f->Flaeche());
}
```

Eine davon abgeleitete Klasse, wie hier z.B. `Rechteck`, **kann** nun ihre eigenen Daten (hier z,B. Länge und Breite als `l, b`) sowie eigene Methoden zur Verfügung stellen. Als einzige **muß** die Methode `Flaeche` implementiert sein, da sie in der Basisklasse als `virtual ... =0;` definiert wurde.

```
class Rechteck:public Figur
{  double l,b;
 public:
   Rechteck(const double x, const double y):
     l(x), b(y), Figur("Rechteck") {}
   double Flaeche(void){return l*b;}
};
```

Wie nach dem vorigen Schema bei `Rechteck` können nun beliebige weitere "*Figuren*" abgeleitet werden:

```
class Quadrat: public Figur
{  double a;
 public:
   Quadrat(const double x) : a(x), Figur("Quadrat") {}
   double Flaeche(void){return a*a;}
};
class Kreis: public Figur
{  double r;
 public:
   Kreis(const double x) : r(x), Figur("Kreis") {}
   double Flaeche(void){return r*r*M_PI;}
};
```

Nun kann folgendes ”triviale” Hauptprogramm, mit der Aufgabenstellung die Fläche aller möglichen Figuren zu berechnen und sie auszugeben, geschrieben werden. Diese Version ist aber nicht sehr flexibel bezüglich dem Hinzufügen ”neuer Figuren”.

```
main()
{ Rechteck R(5,7);
   Quadrat Q(6.23);
     Kreis K(12);
 ShowObj(&R);
 ShowObj(&Q);
 ShowObj(&K);
}
```

Die folgende ”verbesserte” Version nutzt die Möglichkeiten der Übergabe von Objekten mittels Basisklassen-Zeiger besser. Alle möglichen Objekte werden hier mit Hilfe einer `while`-Schleife durchlaufen. Das Hinzufügen einer ”neuen Figur” erfordert weniger Aufwand und kann an zentraler Stelle erfolgen.

```
main()
{ Figur *pF[]={ new Rechteck(5,7), new Quadrat(6.23),
                       new Kreis(12),NULL};
  int i=0;
  while(pF[i])ShowObj(pF[i++]);
}
```

Bei diesem kurzen Beispiel ist der verringerte Aufwand zum Einfügen einer neuen Figur vielleicht nicht unmittelbar einsichtig.
Im ersten Beispiel müssen, außer der zusätzlichen Klassendefinition (das ist jedoch in jedem Fall notwendig) zwei Zeilen an verschieden Programmstellen hinzugefügt werden.
z.B.:
4. Zeile: `Ellipse E(3,12);`
8. Zeile: `ShowObj(&E);`

Beim zweiten Beispiel braucht nur eine Zeile erweitert werden:
4. Zeile vor `NULL};: new Ellipse(3,12),`

Der Mehraufwand scheint hier bei diesem kurzen Beispiel sehr gering bis unbedeutend. Bei großen Programm-Projekten spielen solche Überlegungen eine wesentliche Rolle, da für Wartung und Verbesserungen von Programmen ein Vielfaches der Zeit aufgewendet werden muß, als für das Erstellen des Programms.

9 Überladen von Operatoren

In C++ ist es möglich, daß mittels Klassen neue Datentypen erzeugt und durch speziell definierte Methoden manipuliert werden. Es ist weiters möglich daß, die übliche Notation für die arithmetischen Operatoren `+ - * /` auch auf Objekte der Klasse angewendet werden können und die Wirkungsweise dieser Operatoren für jede Klasse neu definiert werden kann.

Es sollen die Grundlagen und die Vorgehensweise für das Umdefinieren (Überladen) von Operatoren anhand der Beispielsklasse "*Komplexe Zahlen*" (`Complex`) gezeigt werden:

```
class Complex
{ double re,im;

  public:
    Complex(){re=im=0;}
    Complex(double r, double i=0) {re=r; im=i;}
...
};
```

Wenn `a, b` zwei Objekte vom Typ `Complex` sind (z.B. `Complex a(1.,3.), b(4.,5.);`), dann erzeugt

```
Complex c = a + b;
```

den Funktionsaufruf:

`c = operator+(a,b);` oder `c = a.operator+(b);`

Wenn nun die Funktion `operator+(...)` entsprechend definiert ist, kann z.B. eine Addition mit Objekten des Datentyps *Complex* durchgeführt werden.
Es können nicht nur die arithmetischen Operatoren sondern auch folgende Operatoren Überladen werden:

+	-	*	/	%	^	&	|	~	!
=	<	>	+=	-=	*=	/=	%=	^=	&=
|=	<<	>>	>>=	<<=	==	!=	<=	>=	&&
||	++	--	->*	,	->	[]	()	new	delete

Das Überladen von Operatoren ist ein Spezialfall des Überladens von Funktionen und es gelten folgende Regeln:

- Es können keinen ”neuen” Operatoren definiert werden. (z.B. ** für Exponent)
- Die Rangfolge der Operatoren kann nicht verändert werden.
- Es wird zwischen unären und binären Operatoren unterschieden;
- Zumindest ein Argument einer Operatorfunktion muß ein benutzerdefinierter Typ sein.

Ein Operator wird in folgenden Funktionsaufruf konvertiert:

Typ `operator<op>(`Argumentliste`)`

Typ: gültiger C++ Datentyp
`<op>`: einer der Operatoren aus der obigen Liste
Die Operanden werden in der Argumentliste übergeben.

Je nachdem ob es sich um einen unären oder binären Operator handelt, ergibt sich folgende Argumentliste:

1. unärer Operator: 0 oder 1 Argument
2. binärer Operator: 1 oder 2 Argumente

Ein unärer Operator <op> kann verschieden interpretiert und damit auch implementiert werden:

1. Operator <op> ist Elementfunktion:
   z.B.:

   ```
   class Complex
   {...
     Complex operator-(void);
   };
    Complex A, B;
    B=-A;  // Aufruf: B = A.operator-()
   ```

   `operator-(void)` ist eine Elementfunktion der Klasse `Complex` er besitzt **kein** Argument.
   Allgemein: *AClass AClass*`::operator<op> (void);`

2. Operator <op> ist eine globale Funktion (`friend`):
   z.B.:

```
class Complex
{....
  friend Complex operator-(Complex);
};
 Complex operator-(Complex a){...}

Complex A,B;
 B=-A;  // Aufruf: B = operator-(A)
```

   `operator-(complex)` ist in diesem Fall eine globale Funktion (`friend`) und besitzt ein Argument.
   Allgemein: *AClass* `operator`<op>`(`*AClass*`);`

Ebenso kann ein binärer Operator <op> auf zwei verschiedene Arten interpretiert und implementiert werden:

1. Operator <op> ist Elementfunktion:
   z.B.:

```
class Complex
{....
  Complex operator-(Complex);
};
Complex A,B,C;
C=A-B;   // Aufruf  C = A.operator-(B)
```

   `operator-(Complex)` ist hier Elementfunktion von `Complex` und besitzt ein Argument.
   Allgemein: *AClass AClass*`::operator`<op>`(`*AClass*`);`

2. Operator <op> ist eine globale Funktion (`friend`):

```
class Complex
{....
  friend Complex operator-(Complex,Complex);
};
 Complex  operator-(Complex a, Complex b){...}

Complex A,B,C;
C=A-B;   // Aufruf  C = operator-(A,B)
```

   `operator-(Complex, Complex)` ist in diesem Fall eine globale Funktion (`friend`) und die Argumentenliste besteht aus zwei Argumenten.
   Allgemein: *AClass* `operator`<op>` (`*AClass, AClass*`);`

Beispiel: Operator += als Elementfunktion von `Complex`:

```
Complex Complex::operator+=(Complex a)
{ re+= a.re;
  im+= a.im;
  return *this:
}
```

Beispiel: Operator + als `friend`-Funktion von `Complex`:

```
class Complex
{....
  friend Complex operator+(Complex,Complex);
};

inline Complex operator+(Complex a, Complex b)
{ return Complex(a.re + b.re,  a.im + b.im); }
```

Wie bei Funktionen, können auch Operatoren mehrmals überladen werden:

```
Complex operator*(Complex,Complex);
Complex operator*(double, Complex);
```

Die zweite Version von `operator*(double, Complex)` ermöglicht die Multiplikation von einer reellen Zahl mit einer komplexen Zahl:

```
Complex a(2,3),b(5,8);
\\ Aufruf von operator(double, Complex);
Complex c = 2. * a;
\\ Aufruf von operator(Complex, Complex);
Complex d = a * b;
```

Es kann auch der Zuweisungsoperator `a=b;` überladen werden. Ein weiteres Beispiel, mit der bereits bekannten Klasse `String`, soll den Umgang mit dem Zuweisungsoperator bei Klassen, die Zeiger enthalten, verdeutlichen:

```
class String
{char *Buf;
 int iLen;
 public:
    String(int sz){ Buf = new char [iLen=sz];}
    ~String(void){delete [] Buf;}
};
```

Werden nun folgende Programmzeilen ausgeführt:

```
String s1(10), s2(20);
 s1=s2;
```

so gibt es in diesem Fall ein Problem mit der Zuweisung `s1=s2`.
In der ersten Zeile werden durch den Konstruktor zwei char-Arrays erzeugt. Die Zuweisung s1=s2 überschreibt den Zeiger `Buf` des Strings `s1` mit `Buf` von `s2`.
Bis zu diesem Zeitpunkt macht sich das Problem meist nicht bemerkbar. Erst wenn mit Ende der Gültigkeit von `s1` und `s2` der Destruktor aufgerufen wird, dann wird ein und derselbe Speicher zweimal deallociert! Was meist zu einer vorzeitigen Beendigung des Programms ("Programmabsturz") mit einer **"Speicherschutzverletzung"** (*"Segmentation fault"*) führt.

Abhilfe: Den Zuweisungsoperator nach folgendem Schema überladen.

```
String& String::operator=(const String& s)
{ if(this != &s)  //falls s=s
     {delete Buf; //den vorhanden Buffer loeschen
       Buf= new char[iLen=s.iLen];
       strcpy(Buf,s.Buf);
      }
  return *this;
}
```

Damit sind die Probleme jedoch nicht vollständig beseitigt:

```
void f()
{ String s1(10);
  String s2=s1; // Initialisierung, keine Zuweisung
}
```

Jetzt zeigt `Buf` von `s1` und `s2` wieder auf den selben Speicher, da der Default-Kopier-Konstruktor nur die Zeiger kopiert. Der Destruktor löscht dann zweimal den selben Speicher.

Abhilfe: Den Kopier-Konstruktor nach folgendem Schema definieren, damit ein sogenanntes "deep copy" durchgeführt wird.

```
String::String(const String &s)
{Buf = new char [iLen=s.iLen];
 strcpy(Buf,s.Buf);
}
```

10 Templates

”Computer sind dazu da, stupide Dinge ohne zu fragen, beliebig oft zu wiederholen.”

10.1 Funktionentemplates

Angenommen es soll folgendes Problem gelöst werden:
Gesucht ist eine Funktion, die das Minimum von zwei Werten `a,b` sucht.

Funktion: `min(a,b)`
`a,b`: typunabhängig

Dazu gibt es mehrere Lösungsansätze. Im folgenden werden die Vor- und Nachteile der verschiedenen Varianten besprochen:

1. Makros:

   ```
   #define min(a,b) ((a) < (b) ? (a) : (b))
   ```

   Dieses Makro arbeitet mit jedem Datentyp. Es vergleicht z.B. zwei `int`; auch zwei verschiedene Datentypen sind möglich z.B. `long` und `int`. Dabei sind die Typ-Konversions-Regeln zu beachten!

   Auch die anfangs sinnlos scheinenden Klammern um `a` und `b` sind wichtig.
   Angenommen die Klammern wurden weggelassen und das Makro wird mit `min(x-3,y+4)` aufgerufen. Daraufhin führt der Compiler eine reine Textersetzung durch und das Makro wird zu:
   `(x-3<y+4 ? x-3 : y+4)`
   Kennen Sie alle Vorrangregeln bezüglich der Operatoren `- < +`, um mit Sicherheit zu sagen, daß dieser Ausdruck gleich bewertet wird wie:
   `( (x-3)<(y+4) ? (x-3) : (y+4) )`
   was dem ursprunglichen Makro mit allen Klammern entspricht.

2. `inline`-Funktion:

   ```
   inline int min(int a, int b)  { return a < b ? a: b;}
   ```

   Dabei tritt folgendes Problem auf:
   Es können mit dieser Beispiels-Funktion nur `int`-Variablen verglichen werden. Bei anderen Datentypen erfolgt eine Fehlermeldung des Compilers.

Umständliche Lösung: Mehrfaches Überladen von `min()`.

```
inline int min(int a, int b) ...
inline long min(long a, long b) ...
inline int min(int a, long b) ...
...
```

Hat man eine der vielen Möglichkeiten vergessen, gibts wieder einen Fehler beim Compilieren.

3. Template:
   Mit Hilfe von Funktionentemplates kann man es dem Compiler überlassen, bei Bedarf eine mehrfach überladene Funktion zu erzeugen. Man muß dem Compiler nur eine geeignete **Vorlage** (*template*) zur Verfügung stellen.
   Folgendes Beispiel soll die Syntax erklären:

```
template <class type>
inline type min(type a, type b)
{ return a < b ? a : b;}
```

Aufruf:

```
   int a,b,c;
double x,y,z;

a=min(b,c);
z=min(x,y);
```

Der Compiler generiert (erst durch den Aufruf) automatisch in diesem Fall zwei überladene Funktionen `int min(int, int)` und `double min(double, double)`.
Durch die folgenden Programmzeilen versucht der Compiler ebenfalls eine Funktion zu erstellen.

```
   int i;
double y,z;
z=min(y,i);  //FEHLER BEIM COMPILIEREN
```

Da `x` und `y` verschiedene Datentypen haben, müßte die Funktion `double min(int, double)` lauten. Dies wird aber durch das vorgegebene Schema nicht abgedeckt, da hier `type` überall gleich ist.
Lösung:

```
template <class left_t, class right_t>
inline left_t min(left_t a, right_t b)
       { return a < b ? a : b;}
```

Ein Template ist eine Vorlage für den Compiler, das Argument (hier `type`) wie ein spezielles `typedef` zu behandeln und die entsprechende Funktion zu erzeugen.

10.2 Klassentemplates

In den vorigen Kapiteln wurde bereits ein Beispiel gezeigt, um mit Hilfe einer Klasse ein ”sicheres” `int`-Array zu verwalten. D.h. die Größe des Feldes zu ”beobachten” und den Zugriffsoperator ([]) entsprechend zu überladen, um Indexüberschreitungen abzufangen.

```
class IntArray
 {   int *p;
     int size;
  public:
   Array(int sz){p= new int[size=sz];}
   ~Array(void) {delete [] p;}
   int GetSize(void){return size;}
   int & operator[](int i)
     {if(!(i>0 && i<size)
 {/*...Fehler...*/
  exit(0);
 }
      return p[i];
     }
 };
```

Diese Klasse könnte doch als **Vorlage** (*template*) für ein ”sicheres” Array für alle möglichen Datentypen dienen. Man müßte nur eine entsprechende Textersetzung durchführen und den Datentyp `int` durch den gewünschten Typ ersetzen. C++ stellt durch die sogenannten **Klassentemplates** diese Möglichkeit zur Verfügung.
Im folgenden Beispiel wird eine Templateklasse definiert, die ein Feld eines beliebigen Datentyps mit Indexprüfung generiert. Die Definition beginnt mit der Kopfzeile `template <class T>` und im folgenden wird das vorige Beispiel übernommen, nur der Datentyp `int` wird an den entsprechenden Stellen durch `T` ersetzt.

```
template <class T>
class Array
 {  T *p;
    int size;
  public:
   Array(int sz){p= new T[size=sz];}
   Array(void) {p=new T[size=100];}
   ~Array(void) {delete [] p;}
   int GetSize(void){return size;}
   T & operator[](int i)
     {if(!(i>0 && i<size)
{...exit(0); /*...Fehler...*/ }
      return p[i];
     }
 };
```

Der Aufruf (die Instanzierung) dieses Templates, kann nun auf verschiedenste Arten erfolgen. Es folgen einige typische, der vielen möglichen Beispiele:

```
int i;
Array<double> x(100);
for(i=0;i<x.GetSize();i++)x[i]=i*0.5;
...
```

Beim Benutzen des Templates wird der Name der Templateklasse (hier `Array`) angegeben, gefolgt vom Datentyp, der zwischen spitze Klammern <> eingeschlossen ist, dahinter folgt der Name der Instanz, sowie eventuelle Parameter für den Konstruktor. Der Compiler erweitert das Template in eine Klassendefinition: Er ersetzt in diesem Beispiel alle `T` mit `int`.

```
Array<complex> cA(10);
Array <Array<int> > ArrayOfArrays(10);  //auch moeglich !!
```

Als Datentypen können alle Möglichkeiten ausgeschöpft werden, die C++ zur Verfügung stellt (Klassen, structs, Zeiger, Templateklassen, ...).

Zwischen den spitzen Klammern können auch mehrere Argumente an das Template übergeben werden:

```
template <class T, int sz>
class Array
 {  T *p;
    int size;
  public:
   Array(int s){p= new T[size=s];}
   Array(void) {p=new T[size=sz];}
   ~Array(void) {delete [] p;}
   int GetSize(void){return size;}
   T operator[](int i)
     {if(!(i>0 && i<size)
 {... exit(0); /*...Fehler...*/ }
      return p[i];
     }
 };
```

Hier wird eine Ausgangsgröße (default-Wert) für das Feld mitgegeben, für den Fall, daß der Default-Konstruktor aufgerufen wird.
Verwendung:

```
const int NCol=10, NRow=10;
Array< Array<int,NCol>, NRow> Matrix;
//Wie int Matrix[][] nur mit Indexprüfung

for (int r=0;r<NRow;r++) for(int c=0;c<Ncol;c++)Matrix[r][c]=i*3;
x[0][Ncol]=0 //Gibt Index-Fehler
```

Man erreicht durch Angabe eines Arrays als Datentyp in der Templateklasse für Arrays (`Array< Array<int,NCol>, NRow> Matrix;`) daß ein Array von Arrays, also eine Matrix, definiert wird. Die Indexprüfung findet dadurch ”automatisch” für Zeilen und Spalten statt.

Aus einem Klassentemplate können auch weitere Klassen abgeleitet werden:

z.B. Ein Stack beliebigen Typs als Array

```
#include <array.h> // Enthaelt obige Deklarationen fuer Array
template <class T, int sz>
class stack : public Array<T,sz>
{ int StackP;
 public:
 stack(void) : StackP(sz) { }
  int  IsFull(void) {return StackP<0;}
  int IsEmpty(void) {return StackP==GetSize();}
 void    Push(T &x);
    T &  Pop(void);
};

template <class T, int sz>
void stack<T,sz>::Push(T &x)
  {if(!IsFull()) operator[](--StackP)=x;
  }

template <class T, int sz>
T & stack<T,sz>::Pop(void)
  {static T Garbage;
    if(!IsEmpty()) return operator[](StackP++);
    return Garbage;
  }
```

Verwendung:

```
stack<int, 10>  IntSt;

for(int i=IntST.GetSize();--i>=0;)IntSt.Push(i*2);

while( !IntSt.IsEmty() ) printf("%d *", IntSt.Pop());
printf("\n");

// Ausgabe:
18 *16 *14 *12 *10 *8 *6 *4 *2 *0
```

Wenn eine Methode einer Templateklasse nicht `inline` implementiert ist, so müssen Member-Funktionen explizit als Templatefunktionen deklariert werden. D.h. die Kopfzeile die bei der Templateklassendefinition aufscheint, muß vor jeder Methode, sozusagen als Klassentyp, angegeben werden.

```
template <class T, int sz> void stack<T,sz>::Pop(void)
```

Nach dem zweifachen Doppelpunkt folgt dann die Signatur der Methode.

Bei den meisten C++-Systemen, wird eine Templateklassen-Bibliothek mit geliefert, die viele Klassen zur Darstellung bzw. Verwaltung von Komplexen Zahlen, Strings, verketteten Listen, ... zur Verfügung stellt.

11 Signale

Signale sind software-interrupts, mit denen das Betriebssystem einem Prozeß das Auftreten eines außergewöhnlichen Ereignisses mitteilt. Sie sind in `/usr/include/signals.h` und darin weiter inkludierten Dateien) definiert. Die Anzahl der verfügbaren Signale ist `NSIG`. Unter Linux gilt meist `NSIG=32`.
Weitere Informationen erhält man mit `kill -l` oder `man kill` bzw. `man signal`
Dieser Abschnitt ist Linux-orientiert, hinsichtlich weiterer Details siehe Beschreibung der GNU C-library, die ohne wesentliche Änderung für praktisch alle Unix-Systemen anwendbar ist. Andere Betriebssysteme (z.B. MS-Windows) unterstützen mitunter nicht alle Signale oder bedürfen anderer Modifikationen.

Verwendete Abkürzungen für defaults: (I)gnore, (T)erminate, (C)ontinue, (S)top

11.1 Program Error Signals

Default actions: Terminate, core-dump (Programmabbruch, Speicherauszug)

- `(T) SIGFPE` Fatal arithmetic error (z.B. Division durch 0)
- `(T) SIGILL` Illegal instruction (z.B. instruction-pointer im Datenbereich)
- `(T) SIGSEGV` Segmentation fault (Zugriff auf verbotene Speicherbereiche, tritt oft bei nicht-initialisierten Pointer auf)
- `(T) SIGBUS` Ungültiger oder nicht initialisierter Pointer (z.B. Versuch, auf ungerader Speicheradresse eine 4-Byte Variable zu dereferenzieren)
- `(T) SIGABRT` Programm entdeckte selbst einen Fehler und ruft abort auf

11.2 Termination Signals

Default action: Terminate (Programmabbruch)

- `(T) SIGHUP` Hang-up. Verbindung mit User-Terminal wurde unterbrochen. Controlling process (damit alle) Prozesse werden abgebrochen
- `(T) SIGINT` (program interrupt). Vom Benutzer normalerweise durch `<CTRL>-C` ausgelöst.
- `(T) SIGQUIT` Ähnlich wie `SIGINT`. Vom Benutzer normalerweise durch `<CTRL>-\` ausgelöst. Erzeugt core-dump. Debug-Hilfe!
- `(T) SIGTERM` Abbruch des Programms. Handler möglich.
- `(T) SIGKILL` Abbruch des Programms. Kein Handler oder Blocking möglich.

11.3 Alarm Signals

Default-action: Terminate (Programmabbruch)

(T) SIGALARM	Timer ist abgelaufen (z.B. in function `alarm()`).
(T) SIGVTALARM	CPU-timer (process + operating system) abgelaufen.
(T) SIGPROF	CPU-timer für Prozeß abgelaufen

11.4 Asyncronous Signals

Default action: Ignore

(I) SIGIO	File ist bereit für input oder output. Meist nur für Terminal implementiert.
(I) SIGURG	(urgent) Out-of-band data kommen an einem Socket an.

11.5 Job Control Signals

(I) SIGCHLD	Wird an parent-process gesandt, wenn ein child-process beendet wird (terminated oder stopped).
(I)(C/I) SIGCONT	Wird an einen stopped process gesandt → Fortsetzung (wird ignoriert, wenn der Prozeß läuft)
(I)(S) SIGSTOP	Hält einen Prozeß an (kann später fortgesetzt werden). Kein Handler oder Blocking möglich.
(I)(S) SIGTSTP	wie SIGSTOP, Handler möglich.
(I)(S) SIGTTIN	Wird an background-process gesandt, der vom Terminal lesen will.
(I)(S) SIGTTOU	Wird an background-process gesandt, der auf das Terminal schreiben will.

11.6 Weitere Signale

(T) SIGPIPE	Wird an Prozeß gesandt, der auf nicht geöffnete pipe oder auf ein 'unconnected socket' schreiben will.
(T) SIGUSR1	Frei für Benutzer
(T) SIGUSR2	Frei für Benutzer

11.7 Wichtige Funktionen zur Signal-Behandlung

```
sighandler_t signal(int signum, sighandler action)
```

Definiert privates Signal-handling (setzt Signal-handler auf)

`sighandler_t`: return-type von `signal`
`signum`: Signal (eines aus obiger Liste)
`action`: `SIG_DFL` default action, oder `SIG_IGN` ignore Signal, oder selbst definierte Funktion: `void xyz(int signum){....}` (`xyz`: beliebiger Funktionsname)

Rückgabewert: Vor dem Aufruf gültige action, oder EINVAL (Fehler), ungültiges `signum` oder `SIGSTOP` oder `SIGKILL`

Achtung: Der Handler `xyz` wird vom Betriebssystem aufgerufen, wenn das betreffende Signal ankommt, also hinsichtlich des Programmablaufs zu einem nicht unbedingt vorhersehbaren Zeitpunkt.

Daher: Kommunikation mit dem Programm ist nur über globale Variable möglich (bevorzugter Datentyp: `volatile sig_atomic_t`).

Der Handler kann jeden Programmschritt unterbrechen (auch mitten in einem C-Befehl oder - systemabhängig - mitten in einer Bibliotheksfunktion) und damit Schwierigkeiten auslösen, z.B. durch Aufruf von non-reentrant code.
Non-local jumps mit `setjmp` und `longjmp` sind möglich, aber (systemabhängig) gefährlich, weil z.B. Fertigstellung von I/O Vorgängen verhindert werden könnte.
In kritischen Programmphasen: Signale blockieren!

```
sighandler_t sigaction(...)
```

Ähnlich wie `signal`, erlaubt aber detailliertere Vorgaben, wie z.B. das Blockieren von Signalen. Siehe `man sigaction`

```
int raise(int signum)
```

Prozeß sendet das Signal `signum` an sich selbst

```
int kill (pid_t pid, int signum)
```

Sendet das Signal `signum` an den Prozeß mit `pid`. Der Erfolg hängt von den gesetzten permissions ab (z.B. kann man Prozesse anderer Benutzer nicht einfach mit `SIGKILL` beenden).

```
int pause()
```

Hält Ausführung des Programms an, bis ein Signal ankommt, das einen Handler aufruft oder das Programm beendet. Zuverlässiges Verhalten ist nicht ganz einfach zu programmieren. Rückgabewert: -1

```
int sigsuspend(const sigset_t *set)
```

Wie `pause`, setzt temporär die Maske `set` (siehe weiter unten). Umständlicher, aber sicherer als `pause`. Man blockiert zuerst das betreffende Signal und akzeptiert es erst mit dem Funktionsaufruf durch die Wirkung der Maske `set`.

```
char* strsignal(int signum)
```

Liefert einen Zeiger auf Text, der das Signal `signum` beschreibt

```
void psignal(int signum, const char* message)
```

Druckt Beschreibung von `signum` aus, sowie zusätzlich den String `message`

11.8 Masken zum Blockieren von Signalen

Prinzip: Man erzeugt im ersten Schritt eine Maske, die eine Liste von Signalen enthält und verwendet die Maske in einem zweiten Schritt zur Manipulation der Einstellung.

```
int sigemptyset(sigset_t *set)
```

Erzeugt eine leere Maske `set`. Die gewünschten Signale müssen einzeln eingetragen werden.

```
int sigfillset(sigset_t *set)
```

Erzeugt eine volle Maske `set` (enthält alle `NSET` Signale). Unerwünschte Signale müssen einzeln entfernt werden.

```
int sigaddset(sigset_t *set, int signum)
```

Gewünschtes Signal `signum` wird in die Maske `set` eingetragen.

```
int sigdelset(sigset_t *set, int signum)
```

Unerwünschtes Signal `signum` wird aus `set` entfernt.

```
int sigismember(const sigset_t *set, int signum)
```

Wenn `signum` in `set` enthalten ist, wird 1 zurückgegeben. Ist `signum` nicht in `set` enthalten ist, wird 0 returniert, oder -1 bei Fehler.

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)
```

`how`:
`SIG_BLOCK`: Signale in `set` werden blockiert, alle anderen Signale bleiben unverändert.
`SIG_UNBLOCK` Signale in `set` werden akzeptiert (unblocked), alle anderen Signale bleiben unverändert.
`SIG_SETMASK` Signale in `set` werden gesetzt, alle anderen nicht.

`set`: Maske der zu setzenden Signale

`oldset`: Optionaler Parameter, kann `NULL` sein. Gibt alte Maske (Werte vor der Veränderung) zurück.

Wenn ein Handler aufgerufen wird, wird das betreffende Signal automatisch für die Dauer der Ausführung blockiert und anschließend auf Default gesetzt. Im Handler kann das Signal selbst un-blocked und/oder neu aufgesetzt werden.

12 Low level I/O

12.1 Nichtformatierte Ein- / Ausgabe

Die low-level I/O-Funktionen unterscheiden sich von den ”normalen” dadurch, daß sie keine Formatierung vorsehen (wie `scanf` oder `printf`, usw.) und detaillierte Vorgaben über das Interface mit dem Betriebssystem erlauben. Die Kommunikation über (named und unnamed) Pipes und Sockets erfolgt mit Hilfe von low-level I/O-Funktionen völlig analog zum File-I/O. Viele Details im low-level I/O-Bereich sind vom Betriebssystem abhängig.

12.1.1 Elementare Funktionen

```
int open(const char *filename, int flags [, mode_t mode])
```

`filename`: Dateiname

`flags`:

`O_READONLY`	Nur lesen
`O_WRONLY`	Nur schreiben
`O_RDWR`	Schreiben und lesen
`O_APPEND`	Schreiben erfolgt immer am Ende der Datei unabhängig vom aktuellen File-pointer
`O_CREAT`	neue Datei wird erstellt, wenn sie noch nicht existiert, oder bestehende Datei wird verwendet
`O_EXCL`	Ergibt gemeinsam mit `O_CREATE` eine Fehlermeldung, wenn Datei schon existiert
`O_NOCTTY`	Falls Datei ein Terminal ist, soll sie nicht zum controlling terminal des Prozesses werden
`O_NONBLOCK`	`read()` wartet nicht auf input
`O_TRUNC`	Falls Datei beim Öffnen schon existiert, wird die Länge auf Null zurückgesetzt

Genau eines der ersten drei Flags muß gesetzt sein, die anderen können durch den OR-Operator | hinzugefügt werden.

`mode`: Optionale Angabe der Zugriffsrechte (anstatt umask shell-command). Selten verwendet (besser: `chmod()` oder `fchmod()` verwenden)

Rückgabewert: File-descriptor `filedes` oder -1 im Fall eines Fehlers (mit `errno()` oder `perror()` abfragen).

```
int close(int filedes)
```

Schließt das geöffnetes File mit `filedes`.

`size_t read(int filedes, void* buffer, size_t size)`

Liest bis zu `size` bytes und speichert sie in `buffer`. Die Daten sind nicht notwendigerweise Strings und es wird kein `\0` angehängt!

Rückgabewerte: Anzahl der tatsächlich gelesenen Bytes, kann weniger als `size` sein (hängt von den Einstellungen ab).
Bei EOF (end-of-file): 0 (außer wenn `size=0` ist)
im Fehlerfall: -1.

`size_t write(int filedes, void* buffer, size_t size)`

Schreibt bis zu `size` bytes von `buffer` auf das File mit dem File-descriptor `filedes`.

Rückgabewerte: Tatsächlich geschriebene Anzahl von Bytes.
Im Fehlerfall: -1.

`off_t lseek(int filedes, off_t offset, int whence)()`

Setzt die aktuelle Position in der Datei.
`offset`: Position in Bytes, relative zu `whence`

`whence`:
`SEEK_SET` ab File-Beginn
`SEEK_CUR` ab aktueller Position
`SEEK_END` ab File-Ende
Negative Werte: Richtung File-Anfang
Positive Werte: Erweiterung des Files (wird automatisch mit Null-bytes ausgefüllt)

Rückgabewerte: Neue Fileposition.
Aktuelle Position kann mit `lseek(filedes,0,SEEK_CUR)` abgefragt werden.
In GNU/Linux ist `off_t` als `long int` definiert.

`FILE* fdopen(int filedes, const char *opentype)`

Erzeugt einen stream (high-level file access) für `filedes`. `opentype` ist wie in `fopen()` definiert.

`int fileno(FILE* stream)`

Gibt den (low-level) file-descriptor für `stream` zurück.
Stream und low-level I/O nicht mischen!

`int fclean(FILE* stream)`

Leert den Buffer des Streams `stream`.
Output-stream: Daten werden ausgegeben.
Input-Stream: Daten werden zurückgegeben, sodaß sie später gelesen werden können.

12.1.2 Warten auf I/O

```
int select(int nfdes, fd_set* read_fds, fd_set* write_fds,
           fd_set* except_fds, struct timeval *timeout)
```

Mit Hilfe dieser Funktion kann die Ein-/Ausgabe mehrerer Filedeskriptoren gleichzeitig beobachtet werden. Das Programm wartet an der Stelle des Funktionsaufrufs von `select()`, bis auf irgendeinem der angegebenen Filedeskriptoren input oder output ansteht bzw. ein time-out auftritt und setzt dann fort.
Die Filedeskriptoren werden nach"read", "write" und "exception" unterschieden und in den entsprechenden Listen `read_fds`, `write_fds` und `except_fds` angegeben. Dabei können jeweils maximal `nfdes` Filedeskriptoren abgefragt werden. Jede der Listen kann `NULL` sein. Zur Vorbereitung und Abfrage stehen folgende Makros zur Verfügung:
`int FD_SETSIZE`
Anzahl der Einträge in jeder der Listen. Kann für nfdes angegeben werden. In GNU/Linux ist diese Zahl praktisch nicht limitiert.

`void FD_ZERO(fd_set *set)`
Erzeugt eine leere Liste von Filedeskriptoren.

`void FD_SET(int filedes, fd_set *set)`
Erzeugt `filedes` in die Liste `set` ein.

`void FD_CLR(int filedes, fd_set *set)`
Entfernt `filedes` von der Liste `set`.

`int FD_ISSET(int filedes, fd_set *set)`
Returniert `TRUE`, wenn `filedes` in der Liste `set` enthalten ist, sonst 0.

Rückgabewert: Anzahl der insgesamt (in allen drei Listen) anstehenden I/O-Operationen oder 0 im Falle von time-out. Die Listen sind bei der Rückgabe verändert und enthalten nur mehr diejenigen Filedeskriptoren, für die I/O's anstehen. Die Abfrage erfolgt durch `FD_ISSET()`.
Angabe von `time_out`: siehe `struct timeval`. Angabe des `NULL`-pointers bewirkt unendliche Wartezeit, Angabe von Null als Zeit bewirkt keine Wartezeit und daher nur die Abfrage, ob ein aktueller I/O ansteht.

12.1.3 Elementare Kontrolle über Dateien

```
int fcntl(int filedes, int command, ...)
```

Diese Funktion ist ziemlich komplex und erlaubt es viele mögliche Einstellungen vorzunehmen.

Beispiele:

`fcntl(filedes, F_SETFL, O_NONBLOCK);`
Ändert die Einstellung eines bereits geöffneten Files auf non-blocking Zugriff

`int iflags=fcntl(filedes, F_GETFL);`
Rückgabewert: gesetzte Flags
Die folgenden Abschnitte sind speziell auf Linux/Unix ausgerichtet. In den Übungsbeispielen sind auch Programme enthalten, welche die Verbindung über Internet-Sockets zu/von MS-Windows-Systemen aufbauen. Die erforderlichen Modifikationen betreffen nur einige Details der Initialisierung und in geringem Umfang der Funktionsparameter.

12.2 Pipes

Pipes sind Kommunikationskanäle, die zwischen Prozessen mit gemeinsamen Vorfahren aufgebaut werden können. Jede Pipe erlaubt den Datentransfer in eine Richtung. Sie wird vor der Verzweigung (Befehl `fork()`, siehe Child-Prozesse) aufgesetzt.

`int pipe(int filedes[2])`

Öffnet eine Pipe mit dem Filedeskriptor-Paar `filedes`.
`filedes[0]` zum Lesen von der Pipe
`filedes[1]` zum Schreiben in die Pipe
Schreiben und lesen: mit `read()` und `write()` oder mit
`scanf()` und `printf()`, wenn zuvor mit `fdopen()` ein Stream `(FILE*)` erzeugt worden ist.
Schließen mit `close()`.

`FILE* popen(const char* command, const char* mode)`

Ruft die Default-Shell auf, um das Programm `command` auszuführen und erzeugt eine Pipe zu diesem Subprozess.

`mode`:
`"r"`: Pipe liest vom Subprozess. Dieser erbt `stdin` vom Parentprozess.
`"w"`: Parentprozess sendet Daten zum Subprozess. Dieser erbt `stdout`.

`int pclose(FILE* stream)`

Schließt die mit `popen()` geöffnete Pipe.

12.2.1 Named Pipes (FIFO Special Files)

Erlauben die Kommunikation zweier Prozesse, die auf das gleiche Filesystem zugreifen können. Das Prinzip ist das gleiche wie für (unnamed) Pipes. Die Fifo-Datei (`filename`) wird von irgendeinem Prozeß erzeugt und bleibt danach wie eine normale Datei im Filesystem bestehen. Sie kann dann von jedem Prozeß zum Schreiben oder Lesen geöffnet werden. `mode` gibt wieder die Zugriffsrechte an.

```
int mkfifo(const char* filename, mode_t mode)
```

12.3 Sockets

Sockets sind ein sehr allgemeines Konzept, das die Kommunikation zwischen Prozessen unter anderem auch über Netzwerke erlaubt. Hier werden Sockets nur für Internet-Verbindungen (TCP) behandelt. Folgende Parameter werden häufig verwendet und zunächst definiert.

Communication Styles:
`int SOCK_STREAM` analog zu FIFO's und Pipes. Die Datenübertragung erfolgt "reliably" (*verläßlich*), das heißt man erhält Fehlermeldungen, wenn sie nicht erfolgreich ist, und die Daten erreichen den Empfänger in der richtigen Reihenfolge.

`int SOCK_DGRAM` Die Übertragung von Datagrams ist "unreliable" (*unverläßlich*). Sie können verlorengehen, in falscher Reihenfolge ankommen, auch in mehrfacher Ausfertigung, und der Sender erhält keine Fehlermeldungen.

`int SOCK_RAW` Low level access.

Socket Address:
Ein Socket wird zunächst mit der Funktion `socket()` (siehe unten) erzeugt. Wenn der kommunizierenden Partner diesen Socket ansprechen will, muß er es bezeichnen können. Diese Identifikation erfolgt durch Angabe der Internet-Adresse des Hosts und einer Port-Nummer. Die lokale Verbindung eines Sockets mit dem extern bekannten Namen besorgt die Funktion `bind()`.
Die Adresse eines Sockets wird generell (auch bei nicht-Internet-Sockets) in einer Datenstruktur `struct sockaddr` an die verschiedenen Funktionen übergeben:

```
struct sockaddr
{short int sa_family; // für Internet-Sockets: AF_INET
char sa_data[xx]      // Eigentliche Adresse (Länge xx)
}
```

Statt dessen kann die spezielle Form der Internet-Adresse (eventuell mit einem cast) verwendet werden:

```
struct sockaddr_in
{short int sin_family;       // für Internet-Sockets AF_INET
struct in_addr sin_addr;    // Host-Adresse, siehe weiter unten
unsigned short int sin_port // Port-Nummer
}
```

Daten, die an den remote host übertragen werden, müssen ins richtige Format gebracht werden (byte-order) oder im "network-format" übertragen werden. Dazu gibt es eine Reihe von Funktionen.

Erzeugung und Binden von Sockets:

```
int socket(int namespace, int style, int protocol)
```

Erzeugt neues Socket.

`namespace`: `PF_INET` Internet-namespace
`style`: `SOCK_STREAM` oder `SOCK_DGRAM`
`protocol`: 0 Default protocol

Rückgabewert: Filedeskriptor. Kann für `read()` und `write()` verwendet werden (Internet-Sockets sind bidirektionale Verbindungen)

```
int shutdown(int socket, int how)
```

Alternativ zu `close()` zu verwenden, wenn folgende Spezifikationen wichtig sind.

`how`:
0 Weiterhin ankommende Daten werden zurückgewiesen
1 Daten im output-buffer werden gelöscht, ausstehende Bestätigungen gesendeter Daten nicht weiter abgewartet, verlorene oder fehlerhafte Daten nicht nochmals gesendet.
2 beides.

Rückgabewert: 0 (OK) oder -1 (Fehler).

```
int bind(int socket, struct sockaddr* addr, size_t length)
```

Verbindet Socket mit einer Socket-Adresse

`socket`: Rückgabewert von `socket()`
`length`: `sizeof(addr)`
`addr`: Empfohlene Vorgangsweise:

```
addr.sin_family = AF_INET;
addr.sin_port = htons(port);
addr.sin_addr.s_addr = htonl(host)
```

Dabei ist

- `htonl()` eine Funktion, die das Datenformat für `long int` ins network-format umwandelt (host-to-network-long),
- und das Argument `unsigned long int host`: eines der folgenden
  - `INADDR_LOOPBACK` Spricht die eigene Maschine an, ohne die Adresse anzugeben (alternativ auch 127.0.0.1)
  - `INADDR_ANY` jede externe Adresse
  - `INADDR_BROADCAST` Broadcast-Adresse
  - `INADDR_NONE` Rückgabewert bei Fehlern
- oder eine Internetadresse im `long int`-Format.

Die Internetadresse im `long int`-Format erhält man durch einfache Umwandlung: beispielsweise 128.130.10.136:

```
128d = 0x80
130d = 0x82
 10d = 0x0a
136d = 0x88
```

also `0x80820a88 = 128*16`^{3} `+ 130*16`^{2} `+ 10*16 + 136 = 2156006024`
Der Befehl `telnet 2156006024` funktioniert daher auch.

Internet-Adressen:
Können als Namen oder als Zahl angegeben werden. Im Zahlenformat ist es eine vierstellige Zahlenkombination `Aa.Bb.Cc.Dd`, wobei `0 <= Aa, Bb, Cc, Dd >= 255` liegt. Sie wird in eine ”network number” und eine ”lokal network number” unterteilt.

Class A Netzwerke: `0 < A < 127` sehr große Netze, network-number = A (1 Byte)
Class B Netzwerke: `128 <= A < 191` mittlere Netze, network-number = Aa (2 Bytes)
Class C Netzwerke: `191 <= A < 255` mittlere Netze, network-number = Aa.B (3 Bytes)

A=0: reserviert für ”broadcast to all networks”
A=127 reserviert für ”loopback” (127.0.0.1 ist der eigene Host)

Internet Ports:
Erlaubte Portadressen: 0 bis 65535.
`0` bis `IPPORT_RESERVED` sind für Standardapplikationen reserviert (finger, telnet, usw.), die folgenden Adressen bis `IPPORT_USERRESERVED` können vom System automatisch verwendet werden (bis dorthin für den Eigenbedarf vermeiden!), darüber liegen die Server-Ports. (siehe `<netinet/in.h>`).

Verbindungsaufbau (Client)

```
int connect (int socket, struct sockaddr* addr, size_t length)
```

Sie müssen natürlich wissen, welchen Port Sie in der anderen Maschine ansprechen wollen, und es muß dort ein Server-Programm auf Ihre Verbindung warten.

Rückgabewert: 0 (OK) oder -1 (Fehler).

Warten auf eine Verbindung und akzeptieren (Server):

```
int listen(int socket, unsigned int n_pending)
int accept(int socket, struct sockaddr* addr, size_t length)
```

Der Server wartet auf eine ankommende Verbindung, und akzeptiert sie. Die Funktion `accept()` weist der zustandegekommenen Verbindung sofort automatisch einen neuen Socket zu (Rückgabewert von `accept()`, sodaß eine neu ankommender Verbindungswunsch erfüllt werden kann. Dabei können bis zu `n_pending` Verbindungen in die Warteschlange gereiht werden. Die Struktur `*addr` kann zum Prüfen des Clients verwendet werden.
Will man nur eine Verbindung akzeptieren, wird das alte Socket nach dem `accept()` mit `close()` gelöscht.

Den Client einer Verbindung erkennt man auch mit

```
int getpeername(int socket, struct sockaddr* addr, size_t* length_ptr)
```

Achtung auf den Pointer.

Schreiben und lesen:
Neben `read()` und `write()` kann man

```
int send(int socket, void* buffer, size_t size, int flags)
int recv(int socket, void* buffer, size_t size, int flags)
```

verwenden. Die Flags sind
`MSG_OOB` send (receive) out-of-band data (hohe Priorität)
`MSG_PEEK` (nur receive) Daten lesen, aber im buffer lassen
`MSG_DONTROUTE` (nur send, nur für Diagnostik/Wartung)

Weitere Funktionen:

```
struct hostent* gethostbyname(const char* name)
```

Liefert für einen Hostnamen die Daten in `/etc/hosts`

`name`: Host-Name
`struct hostent`: (Rückgabewert, NULL bei Fehler)

```
struct hostent
{char* h_name;      // Name (Text)
char** h_aliases;   // Null-terminierte Liste
int h_addrtype;     // AF_INET
int h_length;       // Länge jeder Adresse
char** h_addr_list; // Null-terminierte Liste aller Netzwerkadressen
                    // (bei mehreren gleichzeitigen Anschlüssen)
char* h_addr;       // = h_addr_list[0]
}
```

Alle Adressen in network-byte order

```
struct hostent* gethostbyaddr(const char* addr, int length, int format)
```

Liefert für einen Hostadresse die Daten in `/etc/hosts` und/oder vom Nameserver.

13 Child-Prozesse

13.1 Aufruf von Betriebssystem-Kommandos

`int system(const char* command)`

Startet eine neue shell (d.h. die default-shell `/bin/sh`) und führt command aus.

Rückgabewert: Status des shell-Prozesses (OK) oder -1 (Fehler)

Den Status erhält man durch Abfragen der Makros auf TRUE

`int WIFEXITED(int status)`: child-process beendet (mit `exit()`)

`int WEXITSTATUS(int status)`: returniert exit-status oder 0

`int WIFSIGNALED(int status)`: beendet durch unhandled signal, dann ist
`int WTERMSIG(int status)` dieser Rückgabewert das Signal

`int WIFSTOPPED(int status)` child-process ist "stopped", dann ist
`int WSTOPSIG(int status)` dieser Rückgabewert das Signal

Das aufrufende Programm wartet auf die Fertigstellung, bevor der nächste Befehl ausgeführt wird. Beim Aufruf von `command` wird der eingestellte Suchpfad `PATH` verfolgt.

13.2 Verzweigung eines Prozesses

Verzweigung in zwei parallel laufende Prozesse:

`int fork(void)`

Erzeugt einen Subprozeß, der eine (fast identische) Kopie des rufenden Prozesses ist. Lediglich folgende Parameter sind unterschiedlich:
Rückgabewert von `fork()`:
im child-process: 0
im parent-process: `pid` des neuen child-Prozesses
oder -1 (Fehler, kein child)

Der child-Prozeß erhält eine eigene `pid`, erbt alle offenen Filedeskriptoren, beginnt mit CPU-Zeit Null, erbt keine Alarme oder anstehende (pending) Signale (aber alle Masken).

`pid`: Process-identification number; Eigene `pid` kann man mit `pid_t getpid(void)` abgefragt werden.
`ppid`: Parent-process-ID; mit `pid_t getppid(void)` abfragbar.

Das parallele Ausführen identischer Programme ist natürlich i.a. sinnlos, doch kann sich jeder der beiden Prozesse durch Abfrage des Rückgabewerts von `fork()` sofort als parent oder child identifizieren und damit eigenständig verhalten.
Häufige Anwendung: child-Prozeß ruft mit einer der `exec`-Funktionen ein weiteres Programm auf und beendet damit sein Dasein.

13.3 Ausführung eines weiteren Programms

Mit `exec(...)` kann ein Programm ausgeführt werden. Hier wird ein neuer Prozeß (neue `pid`) gestartet, das rufende Programm wird gleichzeitig beendet.

```
int execv(const char* filename, char *const argv[])
```

`filename`: Auszuführendes Programm
`argv[]`: Zu übergebende Parameterliste. `argv[0]` ist immer `filename` ohne Pfad. Der letzte übergebene Parameter muß der `NULL`-Pointer sein.
Sucht nicht im Pfad PATH nach `filename`!

```
int execl(const char* filename, const char *arg0,...,NULL)
```

Ganz analog zu `execv()`, nur werden die Parameter einzeln angeführt.

```
int execve(const char* filename, char *const argv[], char* const env[])
```

Erlaubt Übergabe von environment-Strings (`name=value`). Der letzte Wert muß wieder `NULL` sein.

```
int execvp(const char* filename, char *const argv[])
int execlp(const char* filename, const char *arg0,...,NULL)
```

Diese beiden Varianten suchen im Pfad `PATH` nach filename.

```
pid_t waitpid(pid_t pid, int* status_ptr, int options)
```

Returniert in `status_ptr` Status-Information über den Prozeß mit `pid`.
Falls `pid = -1` oder `pid = WAIT_ANY` angegeben wird, wird der Status irgendeines existenten child-Prozesses returniert.
Für `pid = 0` wird der Status irgendeines existenten child-Prozesses der gleichen Prozeß-Gruppe (wie die des rufenden Programms) returniert.

`options`: `WNOHANG` auf die Abfrage nicht warten `WUNTRACED` Abfrage nach stopped und terminated childs

Rückgabewert: child-pid (OK) oder 0 (keine Daten für `WNOHANG`) oder -1 (Fehler)

```
pid_t wait(int* status_ptr)= pid_t waitpid(-1, &status, 0)
```

Vereinfachte Version

A Linux, ein paar wichtige Befehle

A.1 Der Editor

Sie können jeden beliebigen Editor verwenden, der am System installiert ist. Empfohlen und von den Vortragenden verwendet wird emacs und nedit.

A.1.1 Emacs

Aufruf, z.B. zum Editieren der Datei `programm.c`:
`emacs programm.c &`
`emacs` kann auch ohne Dateinamen aufgerufen werden (dann wird der Name spätestens beim Abspeichern angegeben oder der erstellte Text geht verloren).
Das abschließende Zeichen `&` ist optional und bewirkt, daß im Arbeitsfenster weitergearbeitet werden kann (andernfalls ist es blockiert, bis der Editor wieder geschlossen wird).
Beachten Sie, daß Sie die Statuszeile/Eingabezeile am unteren Rand des Editors brauchen und daher sehen können müssen. Gegebenenfalls müssen Sie die Fenstergröße geeignet nachjustieren. Emacs kann mit Maus und Menüs bedient werden, aber schneller ist es, nach einiger Übung Tastaturbefehle zu verwenden, z.B.:

`CTRL-X-CTRL-S`: die im Fenster sichtbare Datei sichern

`CTRL-X-CTRL-F`: eine neue Datei öffnen (gegebenenfalls in einem neuen Fenster)

`CTRL-X-CTRL-C`: im Fenster sichtbare Datei schließen (schließen einer noch ungesicherten Datei bewirkt einen Bestätigungsdialog)

`CTRL-C`: Suchen nach einer Zeichenkette

`CTRL-Leertaste`: Anfangspunkt eines markierten Bereichs setzen (der Endpunkt ist danach die jeweilige Cursor-Position)

`CTRL-W`: Löschen des markierten Texts; wird im Zwischenspeicher abgelegt

`CTRL-Y`: Einfügen des Zwischenspeichers an der Cursor-Position

`CTRL-SHIFT-Minus`: Undo (wirkt zyklisch, also UndoUndo, so daß z.B. gelöschte Eingaben wiederkehren!)

`ESC-X-REPL-TAB-STR-TAB`: Suchen-und ersetzen

Sieht kompliziert aus, ist es aber nicht:
Mit ESC-X werden alle Befehle eingeleitet, die keinen shortcut haben. Der gewünschte Befehl heißt replace-string und könnte so eingegeben werden. Kürzer

ist es, die automatische Befehlsvervollständigung zu verwenden, d.h., nach Eingabe einiger Anfangsbuchstaben die TAB Taste zu drücken. Sobald die eingetippten Buchstaben ausreichen, um den Befehl zu identifizieren, wird er vervollständigt.

A.1.2 Nedit

Aufruf, z.B. zum Editieren der Datei `programm.c`:
`nedit programm.c &`

`nedit` ist ein nahezu selbsterklärender Editor mit "Windowsartigen" Menüs, Tastaturbefehlen und Mausbedienung.

A.2 Das Dateisystem

Das Dateisystem in Linux/Unix ist hierarchisch gegliedert und hat eine einzige Wurzel *(root)* (nicht mehrere Parallelverzeichnisse, wie in der Microsoft-Welt: C:, D:, E:,...Z:). Dateisysteme, die auf verschiedenen Platten oder Partitionen residieren, werden als Sub-Dateisysteme (beliebiges Unterverzeichnis) an beliebiger Stelle des Dateibaums eingefügt (mit dem `mount` Befehl).

Trennzeichen der Verzeichnisebenen ist / (statt \ , wie in DOS/Windows).

Die Arbeitsverzeichnisse der Lehrveranstaltungsteilnehmer sind (im Praktikumsrechner) im Dateipfad /home/XXX/ zu finden, wobei der Name XXX vom Systemadministrator für jede Lehrveranstaltung vorgegeben wird. Der Lehrveranstaltungsleiter weist Ihnen oder Ihrer Übungsgruppe ein user-account zu (z.B. edv1di02, Groß-Kleinschreibung beachten!) mit dem der Einfachheit halber auch die Wurzel ihrer persönlichen Arbeitsverzeichnisse (home-directory) bezeichnet wird, also /home/LV123/edv1di02 ist das home-directory des users edv1di02 in der Lehrveranstaltung LV123. Darin können Sie nun Ihre Dateien und Unterzeichnisse anlegen. Alle von Ihnen angelegten Dateien können von Ihnen nach Belieben verändert und gelöscht werden und Sie können auch Zugriffsberechtigungen setzen (z.B. anderen Benutzern das Lesen, Kopieren, Löschen, Ausführen, usw. verbieten oder erlauben).

Dateien werden bezeichnet (benannt):

- Durch eine Zeichenkette, wobei die Zeichen `a-z,A-Z,0-9`, der Punkt . (kann auch mehrmals in einem Namen vorkommen) und die Zeichen `+ - , _` . unproblematisch sind. Ein Punkt am Beginn eines Dateinamens versteckt die Datei im normalen Listing. Andere Zeichen und Umlaute sind beschränkt möglich (z.T. muß dann die besondere Wirkung und Interpretation eines Zeichens durch das Betriebssystem außer Kraft gesetzt werden). Im Prinzip kann ein Dateiname jedes Zeichen außer / enthalten. Es wird zwischen Groß- und Kleinschreibung unterschieden!

Dateien werden angelegt:

- Ein leere Datei durch den Befehl `touch`, z.B.:
  `touch MeineDatei`
- Durch Kopieren mit `cp` (copy)
  `cp MeineDatei MeineKopierteDatei`
  Durch andere Befehle oder Programme, die Dateien erzeugen, z.B. Editor, Compiler, usw.

Dateien werden gelöscht:

- Eine einzelne Datei durch den Befehl `rm` (remove):
  `rm MeineDatei`
- Mehrere Dateien unter Verwendung von wildcards:
  wobei `*` ein Token für eine beliebig lange Zeichenkette ist und `?` für ein einziges (oder kein) Zeichen (wie in DOS/Windows)
  `rm Meine*`
- Mehrere Dateien unter Verwendung von regular expressions (werden hier nicht näher behandelt)
  `rm Meine[Datei,KopierteDatei,leereDatei,A].dat`
  Löscht jede der folgenden Dateien, wenn vorhanden:
  `MeineDatei.dat, MeineKopierteDatei.dat, MeineleereDate.dat, MeineA.dat`

Es erfolgt keine Rückfrage (wie unter DOS/Windows), die angebenen Dateien - egal wieviele es sind - werden unwiederbringlich gelöscht.

Verzeichnisse werden angelegt (im gerade eingestellten Verzeichnis):

- Durch den Befehl `mkdir`:
  `mkdir MeineUebungsbeispiele`

Verzeichnisse werden angelegt (in einem anderen Verzeichnis)
z.B. in `/home/Maus`):

- Durch den Befehl `mkdir`:
  `mkdir /home/Maus/MeineUebungsbeispiele`

Verzeichnisse werden gelöscht:

- Durch den Befehl `rm -r`
  `rm -r MeineUebungsbeipiele`

  Anmerkung: Es werden alle Dateien und Unterverzeichnisse gelöscht! Auch bei Verzeichnissen kann man wildcards und/oder regular expressions verwenden. Aufpassen!
  `rm -r ~/*`
  löscht ohne Rückfrage Ihre sämtlichen Dateien und Verzeichnisse
  `rm -r /*`
  löscht (mit Administratorrechten) ohne Rückfrage das gesamte Dateisystem!

Verzeichnis wird zum gerade eingestellten Arbeitsverzeichnis:

- Mit dem Befehl `cd` (change directory)

`cd MeineUebungsbeispiele`	...	→ angegebene Verzeichnis
`cd`	...	→ home directory
`cd ~`	...	→ home directory (`~` = home directory)
`cd ~/MeineUebungsbeispiele`	...	`MeineUebungsbeispiele` im home directory
`cd␣..`	...	eine Ebene zurück (Leerzeichen vor .. beachten)

Programmaufruf
z.B. Programm prog im Verzeichnis `/home/LV123/e123456`:

- Angabe des Programmnamens und des Pfads:
  `/home/LV123/e123456/prog`
- Wenn `/home/LV123/e123456` das gerade eingestellte Arbeitsverzeichnis ist, genügt
  `./prog`
- Der Aufruf

  ```
  prog
  ```

  funktioniert nur, wenn das laufende Arbeitsverzeichnis im Suchpfad enthalten ist (Systemeinstellung!)

Das aktuelle Arbeitsverzeichnis wird mit

- `pwd` (print working directory) angezeigt.

Hilfe und Hinweise:

Wenn ein Hilfetext in einem Befehl eingebaut ist, kann er oft mit der Option `--help` aufgerufen werden, z.B.:

```
ls --help
```

Gibt Hilfe zum `ls`-Befehl mit all seinen Optionen.
Oft ist der daraufhin ausgegebene Text mehr als eine Bildschirmseite lang, so daß er nur zum Teil lesbar ist. Abhilfe schaffen die Befehle `more` (standard Unix) und `less` (Linux).

Sie können die Ausgabe eines Befehls in den Input-Speicher eines anderer Befehls hineinschreiben, hier mit der Wirkung, daß der Input seitenweise ausgegeben wird:

```
ls --help | less
```

Sie können mit den page-up und page-down Tasten seitenweise und mit den Cursortasten zeilenweise blättern oder mit q(uit) beenden.

- Meist sind help-Texte als `"man-pages"` verfügbar:
  `man ls`
- Weitere Erklärungen findet man oft mit `info`:
  `info gcc`

Kennt man den Befehl nicht, oder ein Schlüsselwort, das im help-Text vorkommt, ist `apropos` nützlich:

```
apropos file
```

sucht alle Zeilen in help-Texten und listet sie auf dem Bildschirm, die das Wort `"file"` enthalten. Sind das zu viele, kann man jene herausfiltern, die eine weiteres Wort enthalten, z.B. new:

```
apropos file | grep new
```

Hilfe zum grep Befehl finden Sie mit `man grep` !

A.3 Kommunikation zu/von externen Rechnern

Verbindungen zum Server: `server1.physprak.tuwien.ac.at (128.130.49.10)`

1. Clients im EDV-Praktikumsraum Physik
2. Über alle Rechner `*.tuwien.ac.at` (auch Terminal-Server)
   Nur mit `ssh` (forwarding"von X11 möglich)

   ```
   ssh -l username server1.physprak.tuwien.ac
   ssh -f -l username server1.physprak.tuwien.ac.at /usr/X11R6/bin/xterm -ls
   ```

   siehe auch: `man ssh`

Backup oder kopieren von Daten:

- Über Netzwerk mit `scp` (secure-copy) auf einen anderen Unix-Rechner (z.B. Studentenaccount)

```
scp [opt] source dest
[opt] : -r : Verzeichnisse rekursiv kopieren
[source] oder [dest]: username@host:pfadname
                      (username oder hostname kann weggelassen werden)
```

siehe auch: `man scp`

- via e-mail zu jedem anderen Rechner
- `ftp` nur über einen Umweg
  z.B. `ftp` von zu Hause zum Studentenaccount, `scp` vom Studentenaccount zu unserem Server

B Make

Mit dem GNU-Programm '*make*' können vor allem umfangreiche Programmierprojekte, egal mit welcher Programmiersprache sie realisiert werden, effizient verwaltet werden. '*Make*' bestimmt automatisch welche Teile des Programms neu compiliert werden müssen und startet die dazu notwendigen Befehle.
Es empfiehlt sich '*make*' bereits bei einfachen Projekten zu verwenden, da man sich dabei schon viel Tipparbeit ersparen kann. Bei großen Programmen liegt der Vorteil darin, daß nur jene Module neu compiliert werden, bei denen dies notwendig ist. Damit kann, wenn jedesmal beim Ausbessern von Fehlern neu compiliert werden muß, sehr viel Zeit eingespart werden, weil nicht das ganze Programm sondern nur jene Teile bei denen die Änderungen vorgenommen wurden, tatsächlich compiliert werden.
Schon aus diesem Grund empfiehlt es sich ein Programm in mehrere Module zu aufzuteilen.

B.1 Der Ablauf von '*make*'

'Make' entnimmt seine Informationen aus einer Textdatei, die üblicherweise den Namen `Makefile` hat und vom Benutzer mit jedem beliebigen Texteditor erstellt werden kann. Ähnlich einem Shell-Skript (Batch-Datei) sind darin die Beziehungen zwischen den einzelnen Quelltext-Modulen, sowie die Befehle für das Erstellen des lauffähigen Programms (*executeable*) enthalten. Normalerweise erhält man ein Programm durch das Linken mehrerer Objekt-Dateien (`*.o`), letztere werden durch compilieren der Quelltext-Dateien (`*.c`) erstellt.
Sobald ein geeignetes '*Makefile*' erstellt wurde, genügt nach jeder Änderung eines Quelltextes, die Eingabe des Kommandozeilen-Befehls (*shell command*):
`make`
damit werden alle notwendigen (im '*Makefile*' angegebenen) Neucompilierungen gestartet, um das lauffähige Programm neu zu erstellen.
Das Programm '*make*' verwendet eine spezielle Datenbank sowie das Erstellungsdatum der einzelnen Dateien, um herauszufinden welche der Dateien neu erstellt werden müssen und ruft dann die entsprechenden Befehle aus der Datenbank auf.
Der Ablauf von '*make*' kann zusätzlich auch mit Kommandozeilen-Parametern gesteuert werden.
Detailierte Information über '*make*' erhalten Sie mit den Befehlen:

`man make` oder `info make`

B.2 Erstellen von '*Makefiles*'

Das Erstellen eines einfachen '*Makefiles*' soll an Hand eines simplen Beispiels erklärt werden:
Eine beliebige Anzahl von Zahlen (float) soll von der Tastatur eingelesen werden. Danach sollen sie sortiert am Bildschirm ausgegeben werden.
Dieses einfache Projekt soll in zwei C-Quelltext-Dateien sowie eine Header-Datei aufgeteilt werden:

1. `func.cpp`:
   Enthält die Implementationen, der dafür notwendigen Funktionen zum Einlesen, Sortieren und Anzeigen.

   ```
   int iReadArray(float *fA)
   void ShowArray(float *fA, int iLength)
   void SortArray(float *fA, int iLength)
   ```

2. `func.h`:
   Enthält die Deklarationen (Prototypen) der obigen Funktionen.

3. `array.cpp`:
   Enthält das Hauptprogramm (die Funktion `main()`).

Um das lauffähige Programm `array` zu erhalten, müssen folgende Schritte durchgeführt werden:

1. `gcc -c func.cpp` ...compilieren ergibt `func.o`
2. `gcc -c array.cpp` ...compilieren ergibt `array.o`
3. `gcc -o array array.o func.o` ...linken ergibt lauffähiges `array`

Es muß nun ein '*Makefile*' editiert werden, daß dem Programm '*make*' mitteilt wie das Compilieren und Linken der einzelnen Module zu erfolgen hat. Wenn '*make*' `array` durch compilieren und linken wieder erstellt, muß es dabei folgendes beachten:

- Wurde `func.cpp` geändert, so sind die Schritte 1 und 3 durchzuführen.
- Wurde `array.cpp` geändert, so sind die Schritte 2 und 3 durchzuführen.
- Wurde `func.h` geändert, so sind die Schritte 1 bis 3 durchzuführen, da `func.h`, sowohl in `func.cpp` als auch in `array.cpp` inkludiert ist.
- Wurde `array` geändert (z.B. gelöscht), so sind die Schritte 1 bis 3 durchzuführen.
- Wurde `Makefile` geändert, so sind ebenfalls die Schritte 1 bis 3 durchzuführen.

Mit dem '*Makefile*' kann man '*make*' auch noch "mitteilen", daß verschiedene Kommandos auch explizit aufgerufen werden sollen. Z.B. das Löschen von verschiedenen temporären Dateien, oder das Erstellen einer Sicherungskopie der Quelltexte, oder ...

Ein einfaches '*Makefile*' enthält Regeln (*rules*) folgender Bauart:

```
TARGET : DEPENDENCIES
<TAB>    COMMAND
<TAB>    ...
```

Das 'ZIEL' (*TARGET*) ist normalerweise ein Dateiname, der durch ein Programm (Befehl) erzeugt wird. Typische Beispiele für ein 'ZIEL' sind ausführbare Programme oder Objekt-Dateien. Es kann aber ebenso der Name einer Aktion sein, die durchgeführt werden soll, wie z.B. `clean` oder `zip`.

Eine 'ABHÄNGIGKEIT' (*DEPENDENCY*) ist eine Datei, die benötigt wird um das 'ZIEL' zu erstellen. Das 'ZIEL' kann oft auch von mehreren Dateien abhängig sein, die dann alle mit Leerzeichen als Trennsymbol angegeben werden.

Ein 'BEFEHL' (*COMMAND*) ist eine Aktion, die von '*make*' durchgeführt werden soll. Eine Regel kann mehrere 'BEFEHLE' enthalten, jeder in einer eigenen Zeile.

Hinweis: Am Anfang jeder Befehlszeile ist **unbedingt** ein TABULATOR-Zeichen einzufügen! (Auf diese Besonderheit vergißt man leicht.)

Ändert sich eine der 'ABHÄNGIGKEITEN', dann wird der 'BEFEHL' einer Regel ausgeführt, um die entsprechende 'ZIEL'-Datei zu erstellen. Es kann auch Regeln ohne 'ABHÄNGIGKEITEN' geben, wie z.B. jene mit dem 'TARGET' `clean`, die alle nicht mehr benötigten Dateien löscht.

Neben den Regeln kann ein '*Makefile*' noch weitere Informationen enthalten, die aber für unser einfaches Beispiel nicht notwendig sind. Für kompliziertere Anwendungen können auch die Regeln aufwendiger werden. In dem vorliegenden Beispiel passen aber alle Regeln in das oben vorgegeben Muster.
Das folgende '*Makefile*'-Beispiel beschreibt nun '*make*' den Weg um das ausführbare Programm `array` zu erhalten.

```
#Simple version
array: array.o func.o
        gcc -o array array.o func.o
#
array.o: array.cpp func.h
        gcc -c array.cpp
#
func.o: func.cpp func.h
        gcc -c func.cpp
#
clean:
        rm func.o array.o array
```

Das #-Zeichen leitet einen Kommentar ein. Text, ab diesem Zeichen bis zum Ende der jeweiligen Zeile, wird von '*make*' ignoriert. Mit Hilfe von \<Zeilenvorschub> können lange Zeilen in mehrere Zeilen aufgeteilt werden.

B.3 Aufruf von '*Makefiles*'

Man kann nun '*make*' auf verschiedene Arten aufrufen.
Die einfachste ist:

```
make
```

Dabei sucht nun '*make*' nach dem ersten 'ZIEL', das in `Makefile` aufgefunden werden kann und versucht nun alle 'ABHÄNGIGKEITEN' aufzulösen, um zu diesem 'ZIEL' zu gelangen. In unserem Beispiel wird also '*make*' versuchen das 'ZIEL' `array` zu "erreichen", in dem es den 'BEFEHL' `gcc -o array array.o func.o` ausführt. Dazu müssen jedoch vorher die 'ABHÄNGIGKEITEN' `array.o func.o` erfüllt sein, d.h. diese beiden Dateien müssen vorhanden sein. Existiert eine der beiden Dateien nicht, so sucht nun '*make*' nach einem entsprechenden 'TARGET'. Diese werden hier entweder als `array.o:` oder `func.o:` vorgefunden. D.h. existiert eine dieser Dateien nicht, wird der dem 'ZIEL' zugeordnete 'BEFEHL' `gcc -c func.cpp` oder `gcc -c func.cpp` ausgeführt.
Davor werden jedoch die 'ABHÄNGIGKEITEN' dieser 'TARGETS', entweder `array.cpp func.h` oder `func.cpp func.h` geprüft. Es bleiben dann zwei Möglichkeiten:

1. Eine oder mehrere dieser drei Dateien existieren nicht:
   '*make*' sucht nun nach einer Regel, um `func.h, func.cpp` oder `array.cpp` zu erhalten. Eine solche Regel kann aber in unserem einfachen Beispiel nicht gefunden werden. Damit bricht '*make*' mit einer Fehlermeldung wie "`No rule to make target ...`" ab.

2. Das Erstellungsdatum einer dieser drei Dateien ist neuer als eines der vorhergehenden 'ZIELE', dann werden die entsprechenden 'BEFEHLE' ausgeführt, d. h. `array.o:` oder `func.o:` werden neu compiliert.

Eine weitere Möglichkeit ist, '*make*' explizit mit einem 'TARGET' aufzurufen:

```
make clean
```

In diesem Fall wird das ausführbare Programm `array` und die beiden Objekt-Dateien `func.o` und `array.o` gelöscht. Das 'ZIEL' `clean` ist keine Datei, sondern nur eine Aktion. `clean` ist auch keine 'ABHÄNGIGKEIT' in irgendeiner anderen Regel, damit verwendet '*make*' dieses 'TARGET' nur, wenn es explizit aufgerufen wird. Weiters ist zu beachten, daß diese Regel auch keine eigenen 'ABHÄNGIGKEITEN' enthält, damit ist der einzige Zweck dieser Regel die zugehörigen 'BEFEHLE' einfach auszuführen. Solche Regeln, die sich nicht auf Dateien beziehen und nur Aktionen

sind heißen auch ”phony targets”.

Es ist zu beachten, daß ’*make*’ nicht ”weiß” wie ein bestimmter ’BEFEHL’ arbeitet. Es ist die Aufgabe des Programmierers solche Befehle anzugeben, welche die ’ZIEL’-Datei erstellen. ’*make*’ kann nur ”folgsam” jene Befehle, die der Programmierer in der jeweiligen Regel angegeben hat, ausführen um damit die ’ZIEL’-Datei auf den aktuellen Stand bringen.
Als Voreinstellung beginnt ’*make*’ mit der ersten Regel, die es im ’*Makefile*’ vorfindet. Man nennt dies das ”default goal” (Hauptziel). Als ”goals” bezeichnet man ’ZIELE’ die ’*make*’ versucht als Letzte zu erstellen.
Im vorliegenden Beispiel ist das ”goal” die ausführbare Datei `array` zu erstellen. Deshalb muß diese Regel als erste im ’*Makefile*’ stehen.

B.4 Weitere Bemerkungen zu ’*Makefiles*’

’*Makefiles*’ können sehr umfangreich gestaltet werden und es gibt noch viele weitere Möglichkeiten, die in ’*Makefiles*’ eingebaut werden können, um deren Wartung zu vereinfachen und deren Flexibilität zu erhöhen.

- Es können Variablen definiert werden.
- Viele Regeln und Abhängigkeiten sind bereits vordefiniert.
- Es gibt noch zusätzliche Programme, die in Zusammenhang mit ’*make*’ viele Aufgaben automatisch erledigen.
  Z.B.: `makedepend` durchsucht Quelldateien nach inkludierten Header-Dateien und fügt diese automatisch in die Abhängigkeiten ein.

Dies wird an Hand der Übungsbeispiele erläutert.

Ein ”unwissender” Benutzer kann mit Hilfe eines mitgelieferten ’*Makefiles*’ und ’*make*’ aus den Quelltexten entsprechende ausführbare Dateien erstellen, also Programmpakete installieren. Die meisten Linux-Programme werden so verteilt und können so einfach an die jeweilige Systemarchitektur (Prozessortyp, Kernelversion, ...) angepasst werden.

C Debugger

Da die meisten Programme mit mehr oder weniger schwerwiegenden Fehlern behaftet sind, wird ein großer Anteil der Zeit bei der Entwicklung eines Programms zur Fehlersuche verwendet. Bei ganz kleinen Programmen kann man die Fehlersuche behelfsmäßig mit zusätzlichen `print`-Anweisungen durchführen. Spätestens dann, wenn man den Ablauf einer mit mehreren verschachtelten `if`-Anweisungen versehenen Schleife, Zeile für Zeile verfolgen möchte, wünscht man sich einen komfortablen Debugger.
Ein **Debugger** ist ein Programm, daß es gestattet den Ablauf eines anderen Programms bei einer beliebigen Programmstelle anzuhalten, dabei den Wert einer jeden Variablen zu untersuchen und eventuell auch noch zu verändern. Man kann dadurch leichter Programmstellen auffinden, bei denen unerwünschte Effekte (Fehler) auftreten. Weiters wäre es hilfreich, wenn das zu untersuchende Programm nicht nur in Maschinensprache, sondern auch im Quelltext jener Sprache in der es erstellt wurde vorliegt. Ein Debugger sollte es auch ermöglichen, wenn das Programm durch eine Fehlerbedingung (Segmentation fault, ...) beendet wird, im Nachhinein zu untersuchen an welcher Stelle im Programm dieser Abbruch aufgetreten ist.

In der ”Windows-Welt” gibt es einige sogenannte ”Entwicklungsumgebungen”. Das ist eigentlich ein komplettes Programm, das Editor, Compiler, Linker, Debugger, Projektverwaltung, usw. in einer grafischen Umgebung integriert hat.
Unter Linux spielen solche ”Umgebungen” aus folgenden Gründen nur eine untergeordnete Rolle.

- Sie sind ohne Bildschirmgrafik nicht zu bedienen. Da Linux sehr oft auf Server-Systemen ohne grafische Oberfläche läuft, müssen diese Werkzeuge (Editor, Compiler, Linker, ...) auch ausschließlich von der Kommandozeile zu bedienen sein.

- Es steht unter Linux, mit X-Windows eine sehr mächtige grafische Oberfläche zur Verfügung. Da die alle Komponenten für die Programmentwicklung bereits mit dem Betriebssystem mit geliefert werden, kann man sich eine ”Entwicklungsumgebung”, nach den eigenen Vorlieben leicht selbst zusammenstellen.
  Man öffnet ein Terminalfenster (’xterm’) startet von dort aus einen beliebigen Editor. Vom Terminalfenster aus kann man jedesmal das Compilieren der Module (’make’) durchführen. Zur Fehlersuche wird der GNU-Debugger (’gdb’), ein sehr mächtiges Kommandozeilen-Programm, mitgeliefert.
  Für viele Kommandozeilen-Programme gibt es sogenannte ”grafische Frontends”. Das sind Programme, die über grafischer Oberfläche die Eingabe und die Einstellungen entgegennehmen. Im ”Hintergrund” läuft dann ein Kommandozeilen-Programm. Die Ausgabe wird dann wieder auf einer Windowsoberfläche dargestellt.

C.1 gdb

'*gdb*' (GNU-Debugger) ist ein mächtiger Debugger der ohne grafische Oberfläche auskommt. Um eine Programm zu debuggen, muß beim Compilieren mit '*gcc*' der Schalter (*switch*) `-g` angegeben werden.

```
gcc -g -o myprog myprog.c
```

Für die Fehlersuche startet man '*gdb*' mit:

```
gdb myprog
```

Damit startet der Debugger mit einer eigenen Befehlseingabe:

```
(gdb)
```

Hier können unzählige Kommandos eingegeben werden, um 'Breakpoints' zu setzen, Variablen auszugeben, Online-Hilfe, ...

Das Problem bei allen solchen Programmen ist, daß man sich eine Menge an Befehlen auswendig merken muß, um effizient damit arbeiten zu können.
Detailierte Informationen erhalten Sie entweder über die 'Online-Hilfe', indem Sie an der Befehlszeile von '*gdb*' `help` eingeben, oder mit den Befehlen:

`man gdb` oder `info gdb`

Um diese Probleme mit den Kommandozeilen-Programmen zu umgehen, werden mit jedem "besseren" Linux-System mehrere grafische Frontends zu '*gdb*' mitgeliefert. Eines davon ist '*ddd*' (Data Display Debugger).

C.2 ddd

'*ddd*' wird am einfachsten über die Kommandozeile gestartet.

```
ddd myprog &
```

Falls X-Windows installiert ist, startet ein Programm mit Windowsoberfläche, Menü, Popup-Windows und Maussteuerung. Im Hauptfenster wird der Quelltext angezeigt. Über Mausbedienung und verschiedene Menüpunkte können "Breakpoints" gesetzt werden, die Werte aller Variablen angezeigt, das Programm Zeile für Zeile durchlaufen werden und noch vieles mehr.
Viele der Bedienungselemente sind selbsterklärend, außerdem gibt es im Menü einen Hilfe-Knopf. Wie üblich unter Linux gibts auch Hilfe mit:

`man gdb` oder `info gdb`

Am besten mit "learnig by doing" ausprobieren.

Wird ein Programm durch eine Fehlerbedingung (Segmentation fault, ...) beendet, so kann die Stelle an der diese Bedingung aufgetreten ist, herausgefunden werden. Dazu muß das sogenannte 'core-file' mit Hilfe des Debuggers untersucht werden.
Beendet ein Programm unerwartet durch eine Fehlerbedingung, so erstellt im Normalfall das Betriebssystem eine Datei mit dem Namen `core`. Diese Datei enthält ein Abbild des Speichers zum Zeitpunkt des "Absturzes".
'*ddd*' bzw. '*gdb*' können in vielen Fällen, daraus jene Quelltextzeile ermitteln, in der das Ereignis aufgetreten ist. Der Aufruf

```
ddd myprog core &
```

ermöglicht dies.
Daraufhin wird die entsprechende Zeile im Quelltext durch ein spezielles Symbol markiert.

Sollte keine Datei namens 'core' erstellt wurden sein, dann liegt das an der Konfiguration des Systems. Mit dem Befehl

```
ulimit -a
```

erhalten sie die Liste spezieller Systemeinstellung, darunter die maximal erlaubte Größe von 'core files' in 1024-byte Blocks. Z.B.:

```
core file size (blocks)     0 <-----------
data seg size (kbytes)      unlimited
file size (blocks)          unlimited
max locked memory (kbytes)  unlimited
max memory size (kbytes)    unlimited
open files                  1024
pipe size (512 bytes)       8
stack size (kbytes)         unlimited
cpu time (seconds)          unlimited
max user processes          1023
virtual memory (kbytes)     unlimited
```

Mit dem Befehl

```
ulimit -c 20000
```

setzen sie die maximale Größe von 'core files' auf 20 Mbyte. Dies sollte in den meisten Fällen genügen um eine entsprechende Datei zu erhalten.
Wenn ein 'core file' nicht mehr für Debugging-Zwecke benötigt, wird kann es ohne bedenken gelöscht werden.

D Grafische Datendarstellung

In Naturwissenschaft und Technik benötigt man Werkzeuge zur Darstellung von Meßdaten und Funktionen. Eines der meist verwendeten Linux-Programme für diese Aufgabenstellung ist '*gnuplot*'. Damit können sowohl funktionale Zusammenhänge als auch Meßdaten (2- und 3-dimensional) dargestellt werden. '*gnuplot*' kann auf 2 Arte aktiviert werden:

1. Interaktiv:
   Durch Eingabe de Befehls `gnuplot`. Daraufhin erscheint ein 'command prompt'

   ```
   gnuplot>
   ```

   Hier können die Befehle für '*gnuplot*' eingegeben werden, allfällige Grafiken erscheinen in einem X11-Window, deshalb sollte diese Art nur in Zusammenhang mit einem Displaymanger verwendet werden.
   Funktionen können einfach durch die Eingabe der Funktion geplottet werden.

   ```
   plot sin(x)+sin(2*x)
   ```

   führt zu folgenden Bild.

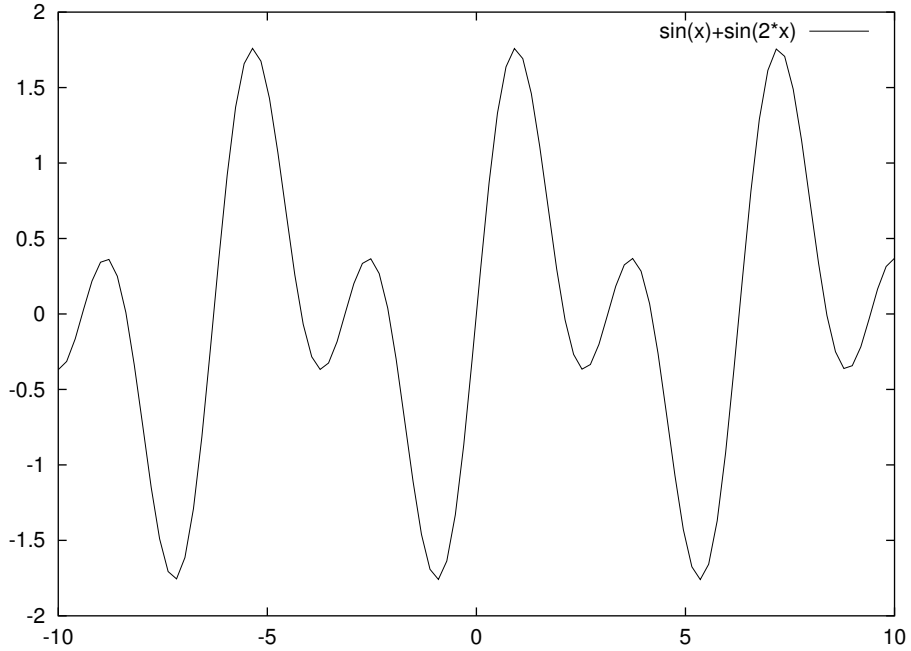

Abbildung D.1: `plot sin(x)+sin(2*x)`

Auf diese Art und Weise können alle Funktionen in C-Notation dargestellt werden.

Sollen Daten grafisch Dargestellt werden, müssen die Meßdaten in einer ASCII-Datei in Spalten eingeteilt vorliegen. Trennzeichen ist das Leerzeichen, Zeilen, die mit # beginnen werden ignoriert. Z.B. Dateiname `square.dat`:

```
#Messdaten
#Quadratische Abhaengigkeit
0 0
0.5 .25
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100
```

Mit dem Befehl `plot 'square.dat'` werden die Daten geplottet.

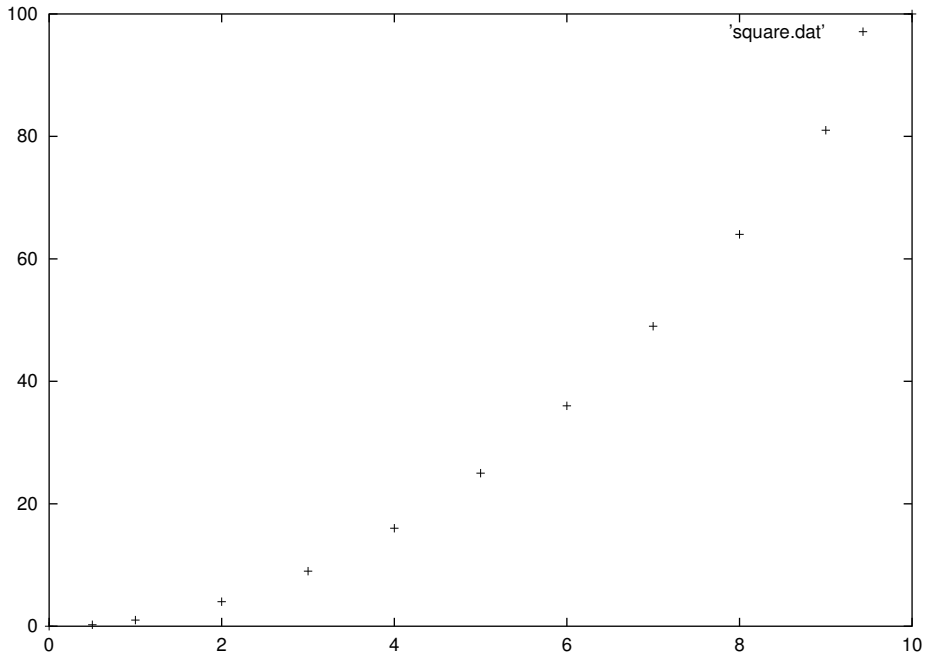

Abbildung D.2: Grafik mit Datenpunkten

Bei diesen einfachen Beispielen verwendet '*gnuplot*' Voreinstellungen für minimale und maximale Werte, Titelzeile, Achsenbeschriftung, Font- und Symbolgröße.
Mit den Befehlen `set ...` können viele Parameter eingestellt werden.

```
set title {"<title-text>"} {<xoff>}{,<yoff>} {"<font>,{<size>}"}
set xrange { [{{<min>}:{<max>}}] {{no}reverse} {{no}writeback} }
set yrange { [{{<min>}:{<max>}}] {{no}reverse} {{no}writeback} }
set xtics {axis | border} {{no}mirror} {{no}rotate}
            {  autofreq
              | <incr>
              | <start>, <incr> {,<end>}
              | ({"<label>"} <pos> {,{"<label>"} <pos>}...) }
...
```

'*gnuplot*' besitzt ein 'online-help'-System. Mit `help` erhält man ein allgemeine Hilfestellung, `help set` gibt Hilfe zum Befehl `set` mit allen seinen untergeordneten Befehlen.

2. Befehlseingabe über eine Datei oder 'pipe':
   Die Anweisungen an '*gnuplot*' können auch in einer Textdatei gespeichert werden und über eine 'pipe' an '*gnuplot*' weitergeben werden. Mit dieser Methode können einfach Bilder in den verschiedensten Ausgabeformaten erstellt werden.
   Erstellt man mit einem beliebigen Editor z.B. folgende ASCII-Datei (`Bsp1.gnuplot`):

   ```
   set terminal postscript
   set output "Beispiel1.ps"
   plot 'square.dat'
   ```

   Mit dem Kommandozeilenbefehl:

   ```
   cat Bsp1.gnuplot | gnuplot
   ```

   wird eine Postscript-Datei mit dem Namen `Beispiel1.ps` erstellt, die an jedem Ausgabegerät (meist Drucker), daß diese Sprache versteht dargestellt werden kann.
   '*gnuplot*' kann derzeit ca. 60 verschiedene Ausgabeformate, darunter auch Pixelgrafik (png), erstellen. Genauere Informationen erhält man mit:

   ```
   (gnuplot) help set terminal
   ```

   Das folgende etwas umfangreichere Beispiel zeigt eine Postscript-Grafik, mit Variablen, Achsenbeschriftung, mehreren Kurven mit verschiedenen Linientypen.

   ```
   #fig18.gnuplot
   reset
   ```

```
f=1
w=f*pi
set sample 10000
set linestyle 1 lw 1
set nokey
set xlabel '{/*1.5 t}' 29,2.5
set ylabel '{/*1.5 f(t)}' -1,8
set border 3 lt 1 lw 1
set xtics nomirror ('{/*1.5 0}' 0,\
                    '{/Symbol*1.5 p}' 1,\
                    '{/*1.5 2}{/Symbol*1.5 p}' 2,\
                    '{/*1.5 3}{/Symbol*1.5 p}' 3)
set noytics
set xzeroaxis lt 1 lw 1
set label 'fig18.eps' at 3.2,-1.55
set terminal postscript eps enhanced 18
set output "fig18.eps"
set rmargin 11
f1(x)=4./pi*sin(w*x)
f2(x)=4./pi/3.*sin(3*w*x)
f3(x)=4./pi/5.*sin(5*w*x)
set yrange [-1.4:1.4]
plot [x=0:3.]  f3(x) with line lt 0 lw 3,\
               f1(x)+f2(x) with line lt 0 lw 4,\
               f1(x)+f2(x)+f3(x) with line lw 1,\
               sgn(sin(w*x)) with lines lt -1 lw 4
```

Die Ausgabe-Datei `fig18.eps` kann auf 2 Arten erstellt werden:
`cat fig18.gnuplot | gnuplot` oder `gnuplot fig18.gnuplot`

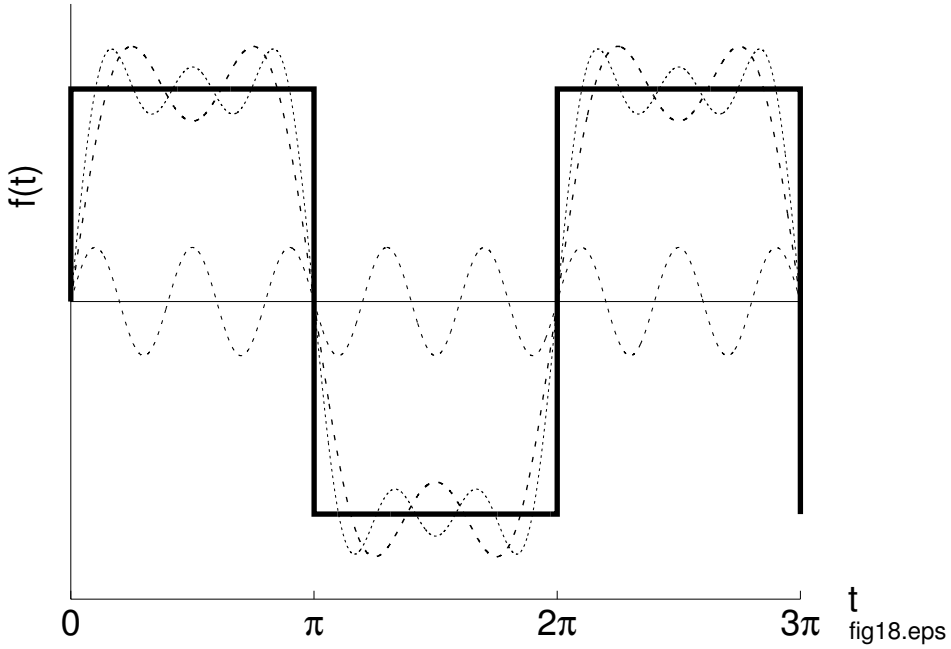

Abbildung D.3: Mehrfach-Plot mit LaTeX-Ausgabe

Ein weiteres Beispiel für LaTeX-Ausgabe:

```
reset
red=1
blue=3
black=7
xmin=-0.01
xmax=10
Dx=xmax-xmin
ymin=-0.01
ymax=11
Dy=ymax-ymin
a=1
b=9
f(x)=sin(2*x)+x
set size 0.4,0.4
set sample 1000
set linestyle 1 lw 1
set nokey
set border 0 lt black lw 1
set noytics
set noxtics
set xtics axis nomirror ('$x_1$' 1,'$x_2$' 2, '\dots' 3, '$x_n$' 9)
#set ytics axis nomirror
set xzeroaxis lt black lw 1
```

```
set yzeroaxis lt black lw 1
set label 'fig3-18' at 1.1*xmax,ymin-0.01*Dy
set label '$x$' at 1.02*xmax,0
set label '$y$' at 0+0.02*Dy,ymax
set label '$f(x)$' at xmax,ymax
set label '$\Delta x$' at b-0.1*Dx,0.5*f(b)

set arrow from 0,0.9*ymax to 0,ymax lt black lw 1
set arrow from 0.9*xmax,0 to xmax,0 lt black lw 1
#
set terminal pslatex color solid norotate
set output "fig19.tex"
set parametric
set yrange [ymin:ymax]
set xrange [xmin:xmax]
set trange [xmin:xmax]
set multiplot
plot t,f(t) with line lt blue lw 2
set trange [a:b]
set sample 9
plot t,f(t) with impulses lt red
set trange [a:b]
set sample 9
plot t,f(t) with lines lt red
```

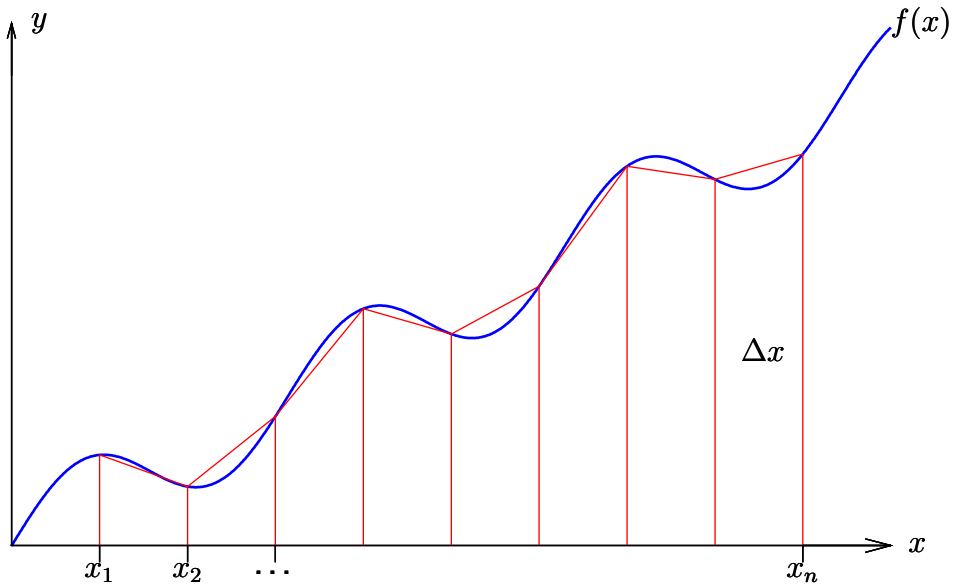

fig19.tex

Abbildung D.4: Graphik mit LaTeX-Ausgabe

Der Vorteil der Eingabe-Datei für '*gnuplot*' ist, daß die Erstellung der Grafik in einer ASCII-Datei (auf jeden System lesbar) vollständig dokumentiert ist.

E Preprocessing

E.1 Ablauf des Preprocessings

Preprocessor

1. Trigraph-Sequenzen werden aufgelöst. Das sind aus 3 Zeichen bestehende Codes für Zeichen, die im 7-bit ASCII code nicht darstellbar sind, z.B `??=` für `#`.
2. Die Sequenzen `\ \n` werden gelöscht (d.h. Verlängerungszeilen wieder zusammengefügt.
3. Der Quellcode wird in Token zerlegt, die Kommentare werden durch ein Leerzeichen ersetzt.
4. Die Macro-Directives werden aufgelöst (i.a. rekursiv).
5. Escape-Sequenzen in Zeichen-Konstanten und Zeichenketten werden durch ihre Werte ersetzt. Aneinander grenzende Zeichenketten werden zusammengefügt.
6. Eigentlicher Übersetzungsvorgang.

E.2 Macro directives

`#define` *identifier token-sequence*

Kann überall stehen und ab dort verwendbar. Ein '*identifier*' darf nicht mehrfach definiert werden (siehe `#undef`). z.B.:

```
#define FELDLAENGE 100

int Tabelle[FELDLAENGE];

#define PRINT printf("\nA B C"); printf("X Y Z\n");

{...
PRINT /* kein Strichpunkt! */
...}
```

`#undef` *identifier*

Macht *identifier* wieder "unbekannt", eine Neudefinition ist jetzt möglich. Das Ansprechen von nicht definierten Identifiern ist wirkungslos und kein Fehler.

`#define` *identifier(identifier-list) token-sequence*

Definition eines Macro-Aufrufs:

```
#define ABSDIFF(a,b) ((a)>b ? (a)-(b) : (b) -(a))

{...
 ABSDIFF(fVar1, fVar2)
 ...}
```

Kann für jeden Variablentyp eingesetzt werden. Die symbolischen Namen innerhalb der Macro-Definition werden nicht rekursiv als Macros interpretiert. Der rekursive Aufruf `ABSDIFF(ABSDIFF(a,b),c)` funktioniert. Die Klammern sollten sorgfältig gesetzt werden, weil die Token selbst komplexe mathematische Ausdrücke sein können.
z.B.

```
#define SQUARE(x) x*x

double a=5,b;
b = SQUARE(a-5);
```

Wird übersetzt zu `b = a-5*a-5;` !

Besser:

```
#define SQUARE(x) ( (x) * (x) )
```

Innerhalb des Macros hat # die spezielle Bedeutung eines Operators. Das Argument wird in Doppelapostrophe `" "` eingebettet und eventuell auftretenden Zeichen \ und `"`, wenn sie innerhalb von `" "` vorkommen, ein \ vorangestellt (Escape-Sequence).

```
#define tempfile(dir) #dir "/%s"

{...
tempfile(/usr/tmp)
 ...}
```

ergibt aufgelöst: `"/usr/tmp" "/%s"` und wird anschließend zu einem einzigen String verkettet.

ist ebenfalls ein Macro-Operator und verkettet zwei Token zu einem einzigen

```
#define cat(x,y) x##y

{...cat(var,123)
 ...}
```

ergibt: `var123`.

Rekursiver Aufruf cat(cat(1,2),3) funktioniert nicht.

`#include <filename>` und `#include "filename"`

wurden bereits ausführlich behandelt. Geschachtelte `#include` Direktiven sind erlaubt.

```
#define INCLUDEFILE <stdio.h>

#include INCLUDEFILE
```

ist legal.

E.3 Bedingte Compilation

Die Macros

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

bezeichen Passagen und Bedingungen für die Compilation einer Passage. Sie müssen mit der jeweiligen Bedingung allein in einer Zeile stehen.
`#if` *const-expression* ist erfüllt, wenn *const-expression* wahr (ungleich Null) ist.
`#ifdef` *identifier* ist erfüllt, wenn *identifier* definert ist (mit `#define`). Gleichbedeutend mit `#if defined` *identifier*. Hier ist `defined` ein Preprocessor-Operator, der auch mit `#elif` verwendet werden kann. Auch die Schreibweise `defined(`*identifier*`)` ist gültig.

`#ifndef` ist erfüllt, wenn emphidentifier nicht definiert ist. Gleichbedeutend mit `#if !defined` *identifier*.

`#elif` emphconst-expression wie `#if`
`#elif defined` *identifier* wie `#ifdef`
`#else` Letzte Alternative, darf nur einmal vorkommen
`#endif` Schließt Bedingung ab.

Beispiel:

```
#if defined(ALLES_OK) ok();
#elif defined(VARIANTE1) var1();
#elif VARIANTE2 > VARIANTE3 var2();
#else printerror();
#endif

#if LEVEL1 > 2
 #define SIGNAL 1
 #if STACKUSE == 1
  #define STACK 200
 #else
  #define STACK 100
 #endif
#else
 #define SIGNAL 0
 #if STACKUSE == 1
  #define STACK 100
 #else #define STACK 50
 #endif
#endif

#if DLEVEL == 0 #define STACK 0
#elif DLEVEL == 1 #define STACK 100
#elif DLEVEL > 5 display( debugptr );
#else #define STACK 200
#endif
```

`#pragma` *token-sequence*

Vom Compiler Hersteller definierte Aktion

`#error` *token-sequence*

Ausgabe einer Fehlermeldung, die token-sequence enthält.

```
#ifndef VAR1
#error Fehler weil VAR1 nicht definiert ist
#endif

#error in line __LINE__ /* funktioniert nicht */
```

Vordefinierte Namen (jeweils mit doppeltem undescore davor und danach):

`__LINE__`	Jeweilige Zeilennummer im Programm (dezimale Konstante)
`__FILE__`	Jeweiliger Dateiname (String)
`__DATE__`	Jeweiliges Datum mmm dd yyyy”(String)
`__TIME__`	Jeweilige Uhrzeit ”hh:mm:ss”(String)
`__STDC__ 1`	(dezimale Konstante), wenn der C-Compiler ANSI-konform eingestellt ist, sonst undefiniert (compilerabhängig). Die ANSI-Konformität kann oft als Option des Compilers eingestellt werden.
`__TIMESTAMP__`	Datum und Uhrzeit der letzten Änderung der Programmdatei (String)

```
#define FORMAT "\nSchleife in Zeile %i zum %i.ten Mal um %s Uhr\
 durchlaufen"

#define IMAX 10

void main()
{int i;
 for(i=0;i<IMAX;++i)printf(FORMAT,__LINE__,i+1,__TIME__);
}
```

Für `gcc` gibt es noch weitere vordefinierte Namen; siehe `info gcc`.

F C und FORTRAN

Wie C-Unterprogrammen in Fortran aufzurufen sind und umgekehrt ist vom Compiler abhängig. Informationen für den g77-Compiler stehen in `http://world.std.com/ burley/g77.html/ f2c-Skeletons-and-Prototypes.html#f2c%20Skeletons%20and%20Prototypes`.
Weitere Informationen und Beispiele gibt es in:
http :// www.starlink. rl .ac.uk/star /docs/sun209.htx/node4.html
Für den Austausch von Daten zwischen Fortran und C-Programmteilen ist zu beachten, dass in Aufrufen von Fortran-Unterprogrammen immer nur die Adressen übergeben werden.
Der Gnu g77 Fortran-Compiler hängt an die Fortran-Funktionsnamen ein Unterstreichungszeichen an. Dies muss in den Namen der entsprechenden C-Programmteile berücksichtigen.

F.1 Aufruf von Fortran aus C

Erzeuge eine Datei main.c:

```
#include <stdio.h>
#include <g2c.h>

extern integer summe_(integer *a, integer *b);

int main (void)
{
  integer n = 100, vektor[100], sum, iCount;
  for(iCount=0;iCount<n;iCount++) {vektor[iCount]=iCount+1; }
  /* Aufruf des Fortranprogrammes summe_ */
  /* Das Unterstreichungszeichen ist noetig, da der gnu g77 Fortran-Compiler
     ein solches Zeichen an den Fortran-Funktionsnamen anhaengt */
  /* Es werden in Fortran nicht die Werte der Variablen übertragen,
     sondern nur ihre Adressen */
  sum = summe_(&n, vektor);
  printf ("%ld\n", sum);
  return 0;
}
```

Erzeuge eine Datei summe.f:

```
      INTEGER FUNCTION SUMME (N, VEK)
      INTEGER VEK(100)
      SUMME=0.
      DO K=1,N
         SUMME=SUMME+VEK(K)
      ENDDO
      END
```

Dann führe aus:

```
gcc -c main.c
f77 -c summe.f
gcc -o test.out main.o summe.o
./test.out
```

oder erzeuge ein Makefile:

```
test: main.o summe.o
      gcc -o test.out main.o summe.o
```

F.2 Aufruf von C aus Fortran

Erzeuge eine Datei main.f:

```
      PROGRAM MAIN
      INTEGER VEK(100),SUM
      N=100
      DO I=1,N
         VEK(I)=I
      ENDDO
      CALL SUMME(N,VEK,SUM)
      WRITE(*,*) SUM
      END
```

Erzeuge eine Datei summe.c:

```
/* Beachte das Unterstreichungszeichen im Namen */
void summe_(int *n, int vek[100], int *sum) {
  int iCount;
  *sum=0;
  for(iCount=0; iCount<*n; iCount++) {
    *sum=*sum+vek[iCount];
  }
}
```

Dann führe aus:

```
gcc -c main.f
f77 -c summe.c
gcc -o test.out main.o summe.o
./test.out
```

oder erzeuge ein Makefile:

```
test: main.o summe.o
      gcc -o test.out main.o summe.o
```

Index