

Grundlagen der Informatik

Vorlesungsskript

Prof. Dr. T. Gervens, Prof.-Dr.-Ing. B. Lang, Prof. Dr.-Ing. C. Westerkamp

5	<u>FUNKTIONEN</u>	2
5.1	FUNKTIONSDEFINITION, -DEKLARATION UND -AUFRUF	3
5.1.1	FUNKTIONSDEFINITION	3
5.1.2	FUNKTIONSDEKLARATION:	4
5.1.3	FUNKTIONSAUFRUF:	4
5.2	FUNKTIONEN UND DER STACK	6
5.3	PARAMETERÜBERGABE	7
5.4	REKURSION	10
6	<u>C-PRÄPROZESSOR</u>	14
6.1	EINFÜGEN VON DATEIEN	14
6.2	TEXTERSATZ, MAKROS	14
6.3	BEDINGTE ÜBERSETZUNG	15

5 Funktionen

Größere Programme werden aus mehreren Gründen in kleinere Einheiten aufgeteilt:

- **Übersichtlichkeit:**
Kleinere Programmteile sind in ihrer logischen Struktur besser zu überschauen.
- **Modularität**
Verschiedene Teile können von verschiedenen Programmierern entwickelt werden.
Die gleiche Funktion kann von verschiedenen Stellen aufgerufen werden.
- **Verstecken von Implementierungsdetails (information hiding)**
Es werden die Details der Implementierung versteckt, die aus übergeordneter Sicht irrelevant sind. Nur die Schnittstelle zum Aufrufen der Funktion muss bekannt sein.

Ein C-Programm besteht i.a. aus mehreren Funktionen und globalen Datendeklarationen. Diese können über mehrere Quelldateien verteilt sein, d.h. getrennte Übersetzbarkeit ist möglich.

Es werden dann alle drei Dateien übersetzt, um die ausführbare Programmdatei zu erhalten. Die Dateien können gemeinsam übersetzt und zum ausführbaren Programm zusammengebunden werden:

```
cc file1.c file2.c file3.c
```

Der aus jeder Quelldatei erzeugte Objektcode wird in einer Objektcode wird in einer Datei abgelegt, welche im Namen weitgehend mit der Quelldatei übereinstimmt, aber statt mit „.c“ mit „.o“ endet. Gibt es in einer der Dateien, z.B. in der Datei file1.c ein Fehler, braucht nach Behebung des Fehlers nur diese neu übersetzt werden:

```
cc file1.c file2.o file3.o
```

Von den übrigen Dateien wird direkt der Objektcode angegeben, dann geht die Erzeugung des ausführbaren Programmcodes schneller.

Die Erzeugung des Objektcodes kann für jede Datei einzeln erfolgen. Der Compilerschalter „-c“ bewirkt, dass die Quelldatei nur zur Objektdatei übersetzt wird; der Bindeprozess muss dann separat durchgeführt werden:

```
cc -c file1.c
```

```
cc -c file2.c
```

```
cc -c file3.c
```

```
cc file1.o file2.o file3.o -o test.exe
```

5.1 Funktionsdefinition, -deklaration und -aufruf

Beispiel:

```
#include <stdio.h>

int power (int m, int n); /* Funktionsdeklaration */

int main() {
    int i=5;
    /* Funktionsaufruf */
    printf ("%d\n", power(2,i)); /* Funktionsaufruf */
}

/* Funktionsdefinition */
int power (int basis, int exp) {
    int i, p;
    p=1;
    for (i=1;i<=exp;i++)
        p=p*basis;
    return p;
}
```

5.1.1 Funktionsdefinition

Syntax:

```
<Funktionstyp> <Funktionsname>
    (<Parameterdeklarationen>) {
    <Vereinbarungen>;
    <Anweisungen>;
    return (<Ausdruck>);
}
```

Funktionstyp:

Jede Funktion besitzt einen Funktionstyp. Zugelassen sind alle bekannten Basistypen.

Eine Funktion, die keinen Wert zurückliefert, bekommt den Typ **void** (Prozeduren).

Ist kein Typ angegeben, wird als Typ **int** angenommen.

Funktionsnamen:

Namen werden nach den gleichen Regeln wie Variablennamen gebildet. Möglichst selbsterklärend wählen!

Parameterdeklarationen:

Der Funktion können Parameter übergeben werden. Alle Parameter sollen explizit deklariert werden. Diese Parameter werden auch Formalparameter genannt. Die runden Klammern müssen vorhanden sein.

Funktionsrumpf {...}:

Der Funktionsrumpf ist ein in sich abgeschlossener Block.

Return:

Die **return**-Anweisung gibt den durch den Ausdruck berechneten Wert an die aufrufende Stelle zurück. Runde Klammern dürfen auch fehlen. Fehlt der Ausdruck, so wird kein Wert zurückgegeben, sondern nur zur aufrufenden Stelle zurückgesprungen. Die **return**-Anweisung muss nicht, sollte aber möglichst am Ende stehen.

Auch die **main-Funktion** besitzt einen Funktionstyp und eine **return**-Anweisung. Fehlt diese, wird implizit **int** als Typ angenommen und ein **return** eingefügt. Viele Unix-Systeme fordern, einen int-Wert zurückzugeben. Da einige Compiler bei fehlendem **main**-Typ eine Warnung ausgeben, ist es empfehlenswert, **main** als **int** zu deklarieren und Null (normales Programmende) zurückzugeben, wenn das Programm normal endet.

5.1.2 Funktionsdeklaration:

Vor dem ersten Aufruf muss die Funktionen deklariert werden:

Syntax:

```
<Funktionstyp> <Funktionsname>  
    (<Parameterdeklarationen>);
```

Wird die Funktion vor dem ersten Aufruf definiert, kann eine zusätzliche Deklaration entfallen.

5.1.3 Funktionsaufruf:

Der Funktionsaufruf kann an jeder Stelle durch die Angabe des Funktionsnamens erfolgen, an der auch ein Wert des Datentyps stehen könnte. Beim Aufruf werden alle Werte der aktuellen Parameter in den Speicherplatz des jeweils zugehörigen formalen Parameters kopiert (**call by value**)

Syntax:

```
<Funktionsname> (<Ausdruck1>, <Ausdruck2>,  
                ..., <AusdruckN>)
```

Die Ausdrücke 1 bis N werden zunächst ausgewertet und die Werte der Ausdrücke den entsprechenden Parametern zugewiesen.

Das **Ergebnis** des Funktionsaufrufs ist (falls die Funktion nicht vom Typ **void** ist) die Auswertung des **return**-Ausdrucks.

Beispiele:

a) Hallo-Programm

```
#include <stdio.h>

void gruss (void);

int main (void) {
    gruss ();
    return 0;
}

void gruss() {
    printf("Hallo");
}
```

b) Polynomberechnung

```
double poly(double, double)

int main() {
    ...
    z=poly(a,b);
    ...
}

double poly (double x, double y){
    double z;
    z=x*x+2*y*y+3*x*x;
    return z;
}
```

Durch ANSI-C wurde die Funktionsbehandlung gegenüber früheren C-Versionen verändert. Häufig sieht man Funktionsdeklarationen und -definitionen noch in der alten Version. Beispiel b) sieht in der alten Version wie folgt aus:

```
double poly(); /* andere Deklaration */

int main() {
    ...
    z=poly(a,b); /* unverändert */
    ...
}

double poly (x,y) /* andere Definition */
double x,y;
{
    double z;
    z=x*x+2*y*y+3*x*x;
    return z;
}
```

Die neue Funktionsdeklaration macht es den Compilern leichter, Fehler bei der Parameterübergabe zu erkennen. Der alte Stil wird von den meisten Compilern noch geduldet. Wir verwenden jedoch ausschließlich den neuen Stil.

5.2 Funktionen und der Stack

Der Stapel eines Prozessors (Stack) ist ein Speicherbereich, auf dem dynamisch zu einer Funktion gehörige lokale Variablen und Informationen abgelegt werden. Die zu einer Funktion zugehörige Information wird als *Stapelrahmen* (oder besser engl. *Stackframe*) bezeichnet.

Lokale Variable sind die Variable, welche im Inneren des Funktionsrumpfs vereinbart werden (und nicht der Speicherklasse **static** zugeordnet sind).

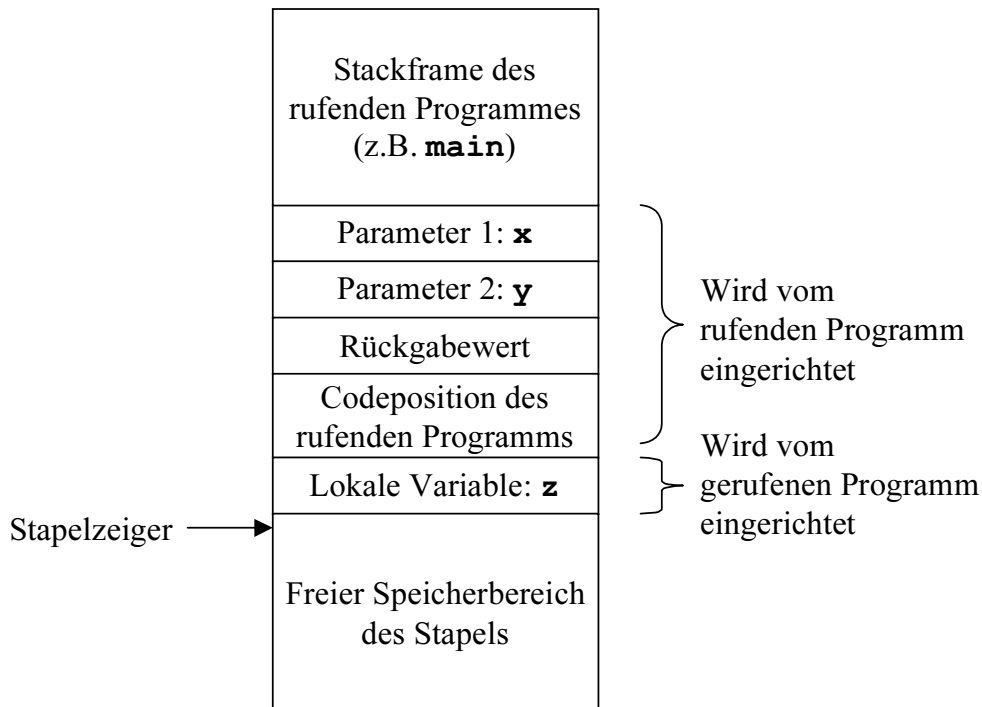
Informationen zu einer Funktion sind

- Position im Code des rufenden Programmes. Diese Position wird benötigt, damit nach Ausführung der gerufenen Funktion der Prozessor wieder zur rufenden Funktion zurückkehren kann.
- Übergabeparameter (aktuelle Parameter), die von der rufenden an die gerufene Funktion übergeben werden.
- Rückgabewert der gerufenen Funktion, der an die rufende Funktion übergeben wird.

Zu einem Stapel gehört ein **Stapelzeiger**, der auf den nächsten freien Eintrag des Stapels zeigt. Relativ zum Stapelzeiger erfolgt der Zugriff auf die lokalen Variablen der Funktion und auf die Informationen zur Funktion. Der Compiler berechnet den Offset vom Stapelspeicher zur Speicherzelle, in der die gewünschte Information liegt. Die Adresse der Speicherzelle wird aus dem Inhalt des Stapelspeichers zur Laufzeit und dem zur Kompilierungszeit bekannten Offset ermittelt.

Beispiel:

Die oben verwendete Funktion „**double poly(double, double)**“ benötigt beispielsweise den folgenden *Stackframe*:



Der Zugriff auf die Variable **z** (Größe 4 Bytes) erfolgt beispielsweise mit der Adresse (Wert des Stapelzeigers)+4. Benötigt man ebenfalls 4 Bytes zur Speicherung der Codeposition (Größe ist prozessorabhängig), so erfolgt beispielsweise der Zugriff auf den Parameter **x** mit der Adresse (Wert des Stapelzeigers)+20. Beim Beenden schreibt die Funktion **poly** ihren durch die **return**-Anweisung spezifizierten Ergebniswert in die Speicherzelle mit der Adresse (Wert des Stapelspeichers)+12.

Jedesmal wenn eine Funktion aufgerufen wird wird ein Stackframe für diese Funktion auf dem Stack angelegt. Nach Bearbeitung der Funktion wird der Stackframe durch Verändern des Wertes im Stapelzeiger wieder vom Stapel entfernt.

Werden aus einer Funktion heraus weitere Funktionen aufgerufen, so werden deren Stackframes zusätzlich auf den Stapel gepackt. Der oberste Rahmen gehört zur aktuellen Funktion, deren Anweisungen gerade vom Prozessor bearbeitet werden.

Die Stackframes liegen (beginnend mit dem Rahmen der **main**-Funktion) in der Reihenfolge auf dem Stapel, in der hierarchisch die Funktionen bis zur aktuellen Funktion aufgerufen wurden.

5.3 Parameterübergabe

Bei der Übergabe von Parametern an eine Funktion werden in der Sprache C *Werte* übergeben, die von der Funktion ausgewertet werden können. Diese Art der Parameterübergabe wird als **call by value** bezeichnet. Bei jedem Aufruf der

Funktion werden die Werte der aktuellen Parameter von der rufenden Funktion in die auf dem Stapel eingerichteten Speicherzellen der Parameter *kopiert*.

Zur Übergabe von Information von einer Funktion an die rufende Funktion ist nur der Rückgabewert vorgesehen. Die Werte der aktuellen Parameter können durch eine Wertveränderung der Funktionsparameter nicht beeinflusst werden.

- Vorteil dieses Vorgehens: sicher
- Nachteil: unflexibel

Andere Sprachen (z.B. Pascal) bieten die Möglichkeit, die Referenz einer Variablen an eine Funktion zu übergeben. Dann kann die Funktion über diese Referenz direkt auf eine Variablen des rufenden Programms zugreifen und auch deren Werte verändern. Diese Art der Parameterübergabe wird als **call by reference** bezeichnet.

In der Sprache C wird die *call by reference*-Parameterübergabe dadurch simuliert, dass man einer Funktion Adressen von Parametern übergibt. Bei *call by reference*-Parametern wird mit Hilfe des **Adress-Operators &** die Adresse des Speicherplatzes ermittelt, der einer Variablen zugeordnet ist.

Beispiel:

```
kreis(radius, &umfang, &flaeche);
```

Eine Funktion mit Namen **kreis** wird aufgerufen. Von den drei Parametern wird der erste Parameter **radius** als Wert und die beiden Parameter **umfang** und **flaeche** als Adresse übergeben.

Zum Zugriff auf den Wert einer Speicherzelle, von der die Adresse verfügbar ist, dient der **Inhalts-Operator ***. Er wird auch als **Dereferenzierungsoperator** bezeichnet.

Beispiel:

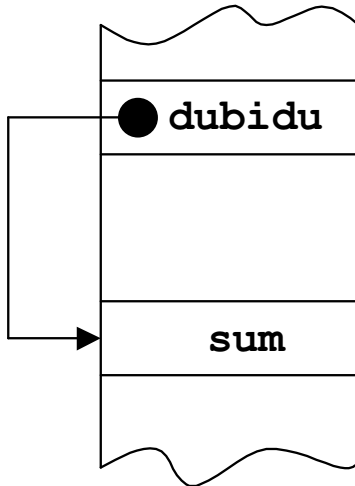
Der Ausdruck ***&summe** (Inhalt von der Adresse von **summe**) ist gleichbedeutend mit dem direkten Zugriff auf eine Variable mit Namen **summe**.

Adressen:

- sind **vorzeichenlose Dualzahlen** einer bestimmten vom Prozessor abhängigen Länge (z.B. 32 Bit).
- Adressen werden in C als **Zeiger** bezeichnet (da die Adresse immer auf den Anfang eines Speicherbereichs zeigt).
- Zeiger der verschiedenen Datentypen werden in C unterschieden. Grund ist, dass die Längen und Informationsdarstellungen der Datentypen unterschiedlich sind und bei Kenntnis des Datentyps immer korrekt auf den Inhalt der zugeordneten Speicherzelle zugegriffen werden kann. Zu jedem Datentyp gibt es somit einen zugehörigen **Zeigertyp** (Zeiger auf int, Zeiger auf float, usw.).
- Die **Definition von Zeigervariablen** erfolgt über den zugehörigen Datentyp, dem Variablennamen eines Zeigers wird jedoch das Zeichen ***** vorangestellt.

Beispiele:

```
float *z_float; /* Zeiger auf float. */
int a, *b, *c, d; /* b,c Zeiger auf int */
float dubidu,*sum; /* sum Zeiger auf float */
...
sum=&dubidu;
```



- Ein **dereferenzierter Zeiger** wird wie eine Variable des Datentyps verwendet, auf den der Zeiger zeigt. (z.B. Wertzuweisung).

Beispiel:

```
double *a, b;
...
a=&b;
*a=5;
(*a)++; /* b=6 */
```

Beispiele:

- a) Vertauschen der Werte von zwei Variablen

```
int tausch(int *, int *);

int main(){
    int a=5,b=6;
    tausch (&a,&b);
    return 0;
}

int tausch(int *x, int *y){
    int hilf;
    hilf=*x;
    *x=*y;
    *y=hilf;
    return 0;
}
```

b) Berechnung der Fläche und des Umfangs eines Kreises

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

void kreis(double, double*, double*);

int main() {
    double radius=10, flaeche, umfang;
    kreis(radius, &flaeche, &umfang);
    printf("Radius=%f Flaeche=%f Umfang=%f",
           radius, flaeche, umfang);
}

void kreis(double r, double *f, double *u){
    *u=2*r*M_PI;
    *f=M_PI*r*r;
}
```

5.4 Rekursion

Eine C-Funktion darf direkt oder indirekt sich selber wieder aufrufen. Solch ein eigener Aufruf wird als **Rekursion** bezeichnet. (Unabhängig von der verwendeten Programmiersprache muss bei rekursiven Funktionsaufrufen durch den Algorithmus gewährleistet werden, dass die Rekursion terminiert.)

Beispiel: Berechnung der Fakultät n!

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

Die Funktion **fak** wird (n+1)-mal mit den folgenden Parameterwerten aufgerufen:

n, n-1, n-2, n-3, ..., 1, 0.

Bei jedem Aufruf wird die Abarbeitung durch den erneuten Aufruf von fak realisiert, bis bei n=0 der Rückgabewert 1 feststeht:

$fak(n) \longleftrightarrow fak(n-1) \longleftrightarrow \dots \longleftrightarrow fak(1) \longleftrightarrow fak(0)$

Iterative Lösung:

```
int fak(int n){
    int i, f=1;
    for (i=1; i<=n; i++)
        f*=i;
    return f;
}
```

Rekursion empfiehlt sich bei oft wiederholenden gleichen Operationen oder bei rekursiv definierten Datenstrukturen (Bäume etc.).

Jede rekursive Lösung kann umgeformt und in eine iterative Lösung überführt werden. Diese führt jedoch in den meisten Fällen zu einem komplizierteren Programmcode. Die rekursive Realisierung eines Algorithmus benötigt hingegen mehr Speicherplatz als die iterative Lösung, ist aber übersichtlicher.

Beispiel: Zahlen einlesen bis eine 0 eingegeben wird. Dann die Zahlen in umgekehrter Reihenfolge wieder ausgeben.

```
void f(void) {
    int i;
    scanf("%d", &i);
    if(i)
        f();
    printf("%d\n", i);
}

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

Beispiel: Berechnung des ggT nach dem euklidischen Algorithmus
(ggT: größter gemeinsamer Teiler)

Definition: d heißt ggT(a,b) falls

1. d/a und d/b
2. $c/a, c/b \Rightarrow c/d$

Beispiele:

$$\text{ggT}(12,30)=6$$

Wichtige Eigenschaften des ggT:

$$\text{ggT}(0,a)=\text{ggT}(a,0)=a$$

$$\text{ggT}(a,b)=\text{ggT}(a-qb,b) \text{ mit } a-qb>0.$$

Beispiele:

$$\text{ggT}(114,18) = \text{ggT}(114-18,18) = \text{ggT}(96,18)$$

$$= \text{ggT}(96-18,18)=\dots$$

$$= \text{ggT}(96\%18,18)$$

$$= \text{ggT}(6,18)= \text{ggT}(18,6)$$

$$= \text{ggT}(6,0)=6$$

Allgemein gilt für den ggT:

$$\text{ggT}(a,b)=\text{ggT}(b,r) \text{ mit } r=a\%b$$

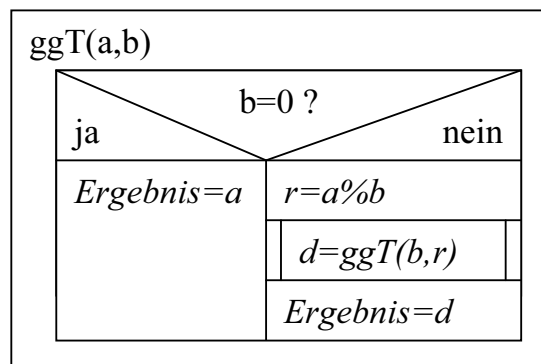
Algorithmus in Pseudocode:

```

ggT(a,b):
  Falls b=0 {
    /* weitere Rekursion nicht notwendig */
    Ergebnis=a
  } sonst {
    Ermittle Rest r der Division a/b:  $r = a \% b$ 
    Berechne  $d = \text{ggT}(b,r)$  /* rekursiver Aufruf */
    Ergebnis ist d
  }

```

Algorithmus als Struktogramm:



Beispiel:

a	B	r
114	18	$114 - 6 * 18 = 6$
18	6	$18 - 3 * 6 = 0$
6	0	

Beispiel: Türme von Hanoi

Problem: In der Stadt Hanoi stehen drei Türme. Auf einem dieser Türme befinden sich 64 Scheiben mit wachsendem Durchmesser. Buddhistische Mönche befassen sich einer Legende nach mit der Aufgabe, diese Scheiben auf eine andere Säule zu tragen unter den Nebenbedingungen

1. Niemals darf mehr als eine Scheibe bewegt werden
2. Nie darf eine größere Scheibe auf einer kleineren liegen.

Die Mönche glauben, das Ende der Welt würde durch die Lösung angekündigt. Dieses scheinbar komplexe Problem besitzt eine einfache rekursive Lösung.

Angenommen das Problem ist für N-1 Scheiben (N: Anzahl der Scheiben) bereits gelöst, dann kann die Lösung für das Problem N Scheiben von Turm A nach C zu bringen wie folgt gelöst werden:

1. Ist $N=1$, bringe die eine Scheibe von A nach C. (STOP)
2. Bringe die oberen N-1 Scheiben mit Hilfe von Turm C von A nach B.
3. Bringe die letzte Scheibe von A nach C.
4. Bringe die N-1 Scheiben von B mit Hilfe von Turm A nach C.

Allgemein lässt sich zeigen, dass für N Scheiben $2^N - 1$ Umlegungen nötig sind. Im Fall N=64 ergeben sich somit $2^{64} - 1$ Bewegungen. Benötigt man pro Bewegung eine Sekunde, so sind die Mönche 584,9 Milliarden Jahre beschäftigt.

```
#include <stdio.h>
void transport(int anzahl, int von, int hilfe,
               int nach, int *p_umleg);
int main(){
    int n;          /* Anzahl der Scheiben          */
    int umleg=0;   /* Anzahl der Umlegaktionen          */
    void transport(int,int,int,int, int *);
    printf("Tuerme von Hanoi\n");
    printf("Anzahl der Scheiben eingeben:\n");
    scanf("%d", &n);
    transport(n, 1, 2, 3, &umleg);
    printf("%d Umlegaktionen\n", umleg);
    return 0;
}
void transport(int anzahl, int von, int hilfe,
               int nach, int *p_umleg){
    if(anzahl>1)
        transport(anzahl-1, von, nach, hilfe, p_umleg);
    printf("Bringe Scheibe %2d von %2d nach %2d\n",
           anzahl, von, nach);
    (*p_umleg)++;
    if(anzahl>1)
        transport(anzahl-1, hilfe, von, nach, p_umleg);
}
```

Für n=3 ergibt sich folgende Ausgabe:

```
Anzahl der Scheiben eingeben:
3
Bringe Scheibe 1 von 1 nach Turm 3
Bringe Scheibe 2 von 1 nach Turm 2
Bringe Scheibe 1 von 3 nach Turm 2
Bringe Scheibe 3 von 1 nach Turm 3
Bringe Scheibe 1 von 2 nach Turm 1
Bringe Scheibe 2 von 2 nach Turm 3
Bringe Scheibe 1 von 1 nach Turm 3
7 Umlegaktionen
```

6 C-Präprozessor

Vor dem eigentlichen Übersetzen werden vom C-Präprozessor einige Vorfunktionen ausgeführt. Diese werden im C-Programm mit dem Zeichen # markiert.

6.1 Einfügen von Dateien

Enthält der Quellcode eines C-Programms eine **#include**-Anweisung in der folgenden Form mit spitzen Klammern ('<', '>'):

```
#include <Dateiname>
```

dann wird vom Präprozessor die Datei mit Namen *Dateiname* im Standard-Include-Verzeichnis des Compilers (z.B. */usr/include*) gesucht und in den Quellcode eingefügt. Weitere Suchverzeichnisse können durch die Compileroption *-I* eingegeben werden; z.B.:

```
cc -I /usr/westerka/include test.c
```

Alternativ kann der Dateiname in doppelte Hochkomma eingeschlossen werden:

```
#include "Dateiname"
```

Dann wird die Datei mit Namen *Dateiname* zunächst im aktuellen Arbeitsverzeichnis gesucht. Wird sie dort nicht gefunden, werden die Include-Verzeichnisse wie oben beschrieben nach der Datei durchsucht.

Durch die **#include**-Anweisung referenzierte Dateien werden als Include-Dateien oder auch als Header-Dateien bezeichnet.

Include-Dateien können in beliebiger Schachtelungstiefe über **#include**-Anweisungen weitere Include-Dateien referenzieren.

Vereinbarungsgemäß besitzen Include-Dateien die Endung „.h“ (Header-Dateien). Bekannte Headerdateien sind z.B.: *stdio.h*, *math.h*, *limits.h*.

In Include-Dateien werden allgemeine Deklarationen (keine Anweisungen, kein ausführbarer Code) eines Programmpaketes zusammengefasst und mit der **#include**-Anweisung in alle betroffenen C-Dateien eingefügt. So ist sichergestellt, dass alle C-Dateien auf dieselben Deklarationen zugreifen.

6.2 Textersatz, Makros

C kennt einen einfachen Mechanismus, um Konstanten, d.h. festen Zeichenketten, Namen zu geben.

Syntax:

```
#define NAME zeichenkette
```

In der gesamten Quelldatei wird dann *NAME* durch *zeichenkette* ersetzt.

Beispiel:

```
#define LENGTH 13
int main() {
    printf("Laenge: %d\n", LENGTH) ;
}
```

```

    return 0;
}

```

Der Textersatz wird nur für vollständige, abgeschlossene Namen durchgeführt. Findet sich *NAME* innerhalb einer längeren Zeichenkette, findet keine Ersetzung statt.

Die Definition *NAME* wird als **Makro** bezeichnet.

Konvention: Namen der mit **#define** vereinbarten Konstanten sollten vereinbarungsgemäß nur mit GROSSBUCHSTABEN bezeichnet werden.

Textersatz ist besonders bei Konstanten nützlich:

```

#define EOF          -1
#define NULL         0
#define BUFFERSIZE 1000

```

Ein Makro kann auch parametrisiert werden. Das entsprechende Präprozessor-Makro hat dann folgende Form:

```

#define NAME(Arg1,...,ArgN) zeichenkette

```

Beim Aufruf des Makros *NAME* ersetzen die aktuellen Makro-Parameter jedes Vorkommen der formalen Makro-Parameter in der Zeichenkette *zeichenkette*.

Beispiel:

```

#define ISLOWER(c)  ((c)>='a' && (c) <='z')
#define TOUPPER(c)  ((c)+'A' - 'a')
#define TOLOWER(c)  ((c)+'a' - 'A')
#define max(A,B)    ((A)>(B) ? (A) : (B))

TOUPPER('x') wird zu: (('x')+'A' - 'a')
max(x,y)      wird zu: ((x)>(y) ? (x) : (y))

```

Ein mit **#define** definiertes Makro *NAME* kann mit:

```

#undef NAME

```

wieder gelöscht werden.

6.3 Bedingte Übersetzung

Der Präprozessor kann mit Bedingungen gesteuert werden, die bestimmen, welche Anweisungen in den Quellcode eingefügt werden.

Struktur:

```

#if Bedingung
    Anweisungen (C-Anweisungen oder Präprozessor-
Direktiven)
#elif Bedingung
    Anweisungen
...
#else
    Anweisungen
#endif

```

Es werden nur die C-Anweisungen übersetzt bzw. die Präprozessor-Direktiven aus dem Abschnitt ausgeführt, bei dem die *Bedingung* erfüllt ist.

Beispiel:

```
#if SYSTEM == SYSV
    #define HD "sysv.h"
#elif SYSTEM == MSDOS
    #define HD "msdos.h"
#else
    #define HD "default.h"
#endif
```

Präprozessor-Bedingungen lassen sich auch mit `#ifdef` und `#ifndef` formulieren. Diese Präprozessor-Anweisungen dienen zur Überprüfung, ob ein Name definiert ist.

Beispiel:

```
#ifndef HD
    #define HD "default.h"
#endif
```

Beispiel:

```
#define DEBUG 1
...
#ifdef DEBUG
    printf("%d\n", n);
#endif
```

Mit der Compiler-Option `-DDEBUG=1` kann das Makro auch beim Compilieren definiert werden.

```
cc -DDEBUG=1 test.c
```

Beispiel:

```
#include <stdio.h>
main() {
    #if DEBUG==1
        printf("Jetzt debuggen mit Option 1\n");
    #elif DEBUG==2
        printf("Jetzt debuggen mit Option 2\n");
    #endif
}
```