5. Unterprogramme

5.1 Prozeduren und Funktionen

5.2 Deklaration und Aufruf

5.3 Call-by-Value / Call-by-Reference

5.4 Standardfunktionen in C

5.5 Gültigkeitsbereich von Namen

5.6 Rekursion

5.1 Prozeduren und Funktionen

Unterprogramme dienen zur **Modularisierung** von Programmen.

Unterprogramme werden **deklariert** und **aufgerufen**.

Generell unterschiedet man in prozeduralen Programmiersprachen zwei Arten von Unterprogrammen:

a) Prozeduren

- führen einen Teil der Arbeit des aufrufenden Programms durch
- Ergebnisse können in Parametern an das aufrufende Programm zurück gegeben werden.

b) Funktionen

- führen die Berechnung von Funktionswerten (meist im mathematischen Sinn) durch
- Ein Ergebnis wird (gegebenenfalls zusätzlich zu anderen Parametern) an der Stelle des Funktionsnamens als Wert zurück geliefert.

In C gibt es nur einen Typ von Unterprogrammen: **Funktionen**. Ergebniswerte von Funktionen können in C jedoch auch ignoriert werden, so dass eine C-Funktion sich dann wie eine Prozedur verhält.

5.2 Deklaration und Aufruf

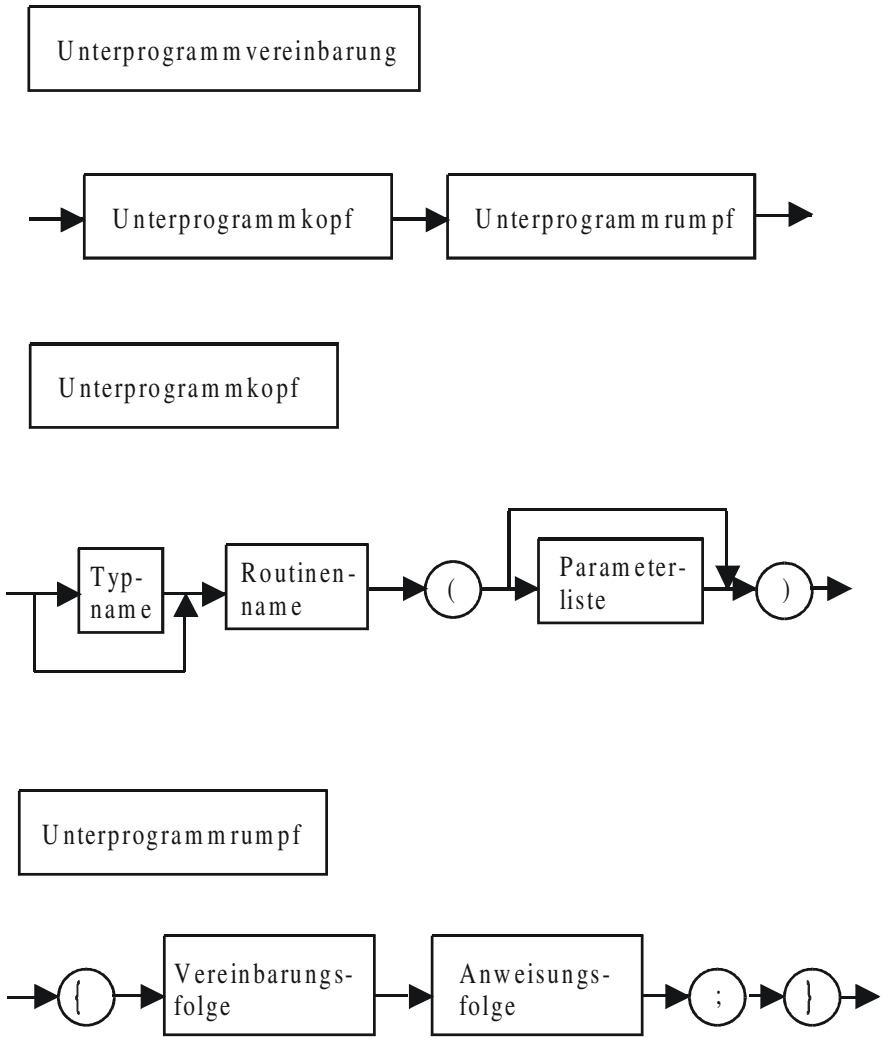
In der **Deklaration** eines Unterprogramms wird festgelegt, wie das Unterprogramm heißt, welche Parameter es hat, welche lokalen Variablen im Unterprogramm benutzt werden werden und wie die Funktionsweise ist (ausführbare Anweisungen).

Der **Kopf** der Deklaration enthält den Namen und die Parameter des Unterprogramms, der **Rumpf** („body“) enthält den auszuführenden C-Code.

Beim **Aufruf** eines Unterprogramms wird zur Laufzeit auf das Unterpogramm verzweigt. Dabei werden die aktuellen Parameter an das Unterprogramm übergeben. Nach Ausführung des Unterprogramms kehrt der Rechner an die auf den Aufruf folgende nächste Anweisung im Hauptprogramm zurück bzw. fährt mit der Berechnung das Ausdruck im Hauptprogramm fort, in dem der Unterprogrammaufruf steht.

Der Aufruf erfolgt in C stets **synchron**, das heißt, dass die Ausführung des Hauptprogramms ruht, solange das Unterprogramm ausgeführt wird.

Deklaration von Unterprogrammen (1)

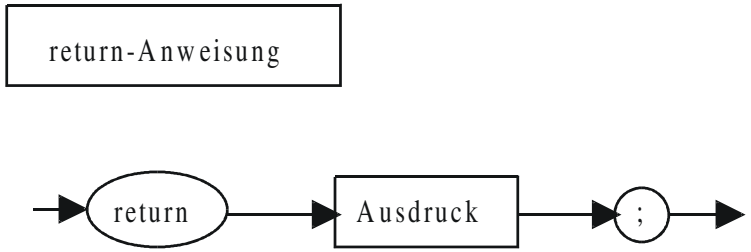


Deklaration von Unterprogrammen (2)

Unterprogramme sollten im Deklarationsteil eines C-Programms, im Anschluss an die Variablen-Deklarationen, deklariert werden. Es sollte immer ein Typ für das Ergebnis angegeben werden, auch wenn der Compiler dies nicht verlangt. Wird keiner angegeben, geht der Compiler vom Typ "int" für die Routine aus.

Die `return`-Anweisung

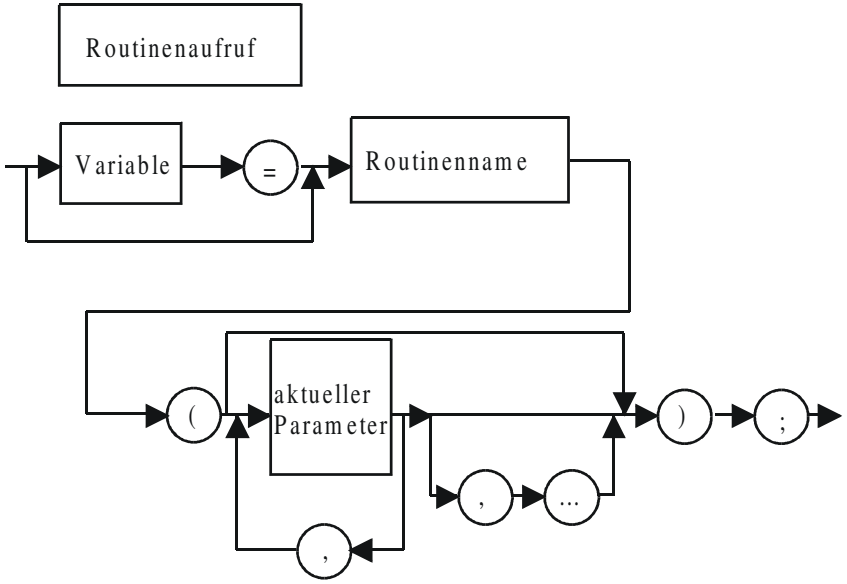
Die Rückgabe von Ergebnissen einer Funktion erfolgt in der Funktion durch die Anweisung **`return`**:



Nach dem Schlüsselwort kann ein beliebiger Ausdruck stehen. Das Ergebnis der Funktion ist das Ergebnis der Berechnung dieses Ausdrucks.

Merke: Eine Funktion muss keinen Resultatwert liefern. Ein leeres **`return`** bzw. die abschließende geschweifte Klammer beenden die Funktion und geben 0 zurück (normales Ende). Eine Funktion sollte jedoch immer eine Return-Anweisung haben, um ein definiertes Ende explizit anzuzeigen.

Aufruf von Unterprogrammen



Der Aufruf ohne Zuweisung an eine Variable ist, syntaktisch gesehen, eine **Anweisung** (statement) und entspricht damit der oben erwähnten Prozedur. Solche Prozeduren sollten mit dem Typ **`void`** vereinbart werden.

Der Aufruf mit Zuweisung an eine Variable ist, syntaktisch gesehen, ein **Ausdruck** (expression) und entspricht damit einer Funktion, bei der ein Wert zurückgegeben wird.

Beispiel für ein Unterprogramm

```
void mache_garnichts ()
{
}
double square_root(double value)
{
return (sqrt(value));
}
```

Aufrufe:

```
double a, b;
        .
        .
mache_garnichts ();
        .
        .
b = 4.0;
a = square_root (b);
```

Korrespondenz zwischen formalen und aktuellen Parametern

- Formale und aktuelle Parameter entsprechen einander in der Reihenfolge, in der sie in der Vereinbarung und im Aufruf auftreten (Stellungsparameter).
- Aktuelle und formale Parameter sollten in der Anzahl übereinstimmen.
- Aktuelle und formale Parameter sollten im Typ übereinstimmen.

Variable Parameterlisten

Es dürfen beim Aufruf eines Unterprogramms auch mehr Parameter übergeben werden als vereinbart sind. Dazu ist in der Deklaration ' ...' als letzter Parameter notwendig. Dies ist sinnvoll, wenn vorab nicht bekannt ist, wie viele Parameter übergeben werden sollen.

Beispiel

Die später noch einzuführende Funktion printf ist folgendermaßen deklariert:

```
int printf (const char *format , ... )
```

Sie gibt u.a. den Inhalt beliebig vieler Variablen auf dem Bildschirm aus.

Achtung:

Es dürfen jedoch nie **weniger** Parameter als vereinbart übergeben werden, da ansonsten der Effekt des Aufrufs undefiniert ist.

5.3 Call-by-Value / Call-by-Reference

Beispiel für schwierige Semantik

```
int i;
void test(int k; int j) {
     k = k+1;
     j = 3*i;
     return;
} /* test */

main () {
     int a[3];
     a[0] = 1; a[1] = 2; a[2] = 3;
     i = 1;
     test(i,a[i]);
     return;
}
```

Fragen bezüglich i und a[i]:

Wie wird auf den aktuellen Parameter zugegriffen?

- indem innerhalb der Funktion Speicherplatz angelegt und der Wert dorthin kopiert wird?
- indem direkt auf den Speicherplatz der Variablen im Hauptprogramm zugegriffen wird?
- indem der Parameter-Ausdruck bei jeder Benutzung neu berechnet wird?

Call-by-Value

- Für jeden Parameter wird ein dem Typ entsprechender Speicherplatz lokal im Inneren der Funktion reserviert.
- Zum Zeitpunkt des Aufrufs wird der Wert des Ausdrucks berechnet, der als aktueller Parameter angegeben ist. Dieser Wert wird in den lokalen Speicherplatz im Inneren der Funktion hinein geschrieben.
- Alle Operationen innerhalb der Funktion werden auf den lokalen Speicherplätzen ausgeführt.

Also:

Die Berechnungen im Inneren der Funktion haben keine Auswirkungen auf die Parameterwerte außerhalb der Funktion! Parameter sind nur **Eingangsparameter**.

Beispiel für den Call-by-Value

```
int i, a[3];

void test (int k, int j) {
    k = k+1;
    j = 3*a[i];
    return;
} /* test */

main () {
    a[0] = 1;
    a[1] = 2;
    a[2] = 3;
    i = 1;
    test (i, a[i]);
    return;
}
```

i	a[0]	a[1]	a[2]
1	1	2	3

Call-by-Reference

- Beim Eintritt in die Funktion wird die **Speicheradresse** des Parameters berechnet.
- Alle Operationen innerhalb der Funktion werden direkt auf den so berechneten Speicheradressen ausgeführt.

Also:

Änderungen des Parameterwertes wirken sich auch auf die Umgebung aus. Geeignet für **Eingangs- und Ausgangsparameter**, d.h. mit Call-by-Reference können Daten des aufrufenden Hauptprogramms geändert werden.

Beispiel für den Call-by-Reference

```
int i, a[3];
/* Deklar. der Parameter als Zeiger */
void test (int *k, int *j) {
    *k = *k+1;
    *j = 3*a[i];
    return;
} /* test */

main () {
    a[0] = 1;
    a[1] = 2;
    a[2] = 3;
    i = 1;
    test (&i, &a[i]);
    return;
}
```

i	a[0]	a[1]	a[2]
2	1	9	3

Call-by-Name

- Betroffen sind Ausdrücke als aktuelle Parameter
- Operationen werden auf den Original-Speicherplätzen außerhalb der Funktion ausgeführt
- Dabei erfolgt aber jetzt eine erneute Auswertung des Ausdrucks bei jeder Verwendung innerhalb der Funktion!

Also:

Der Call-by-Name führt dann zu anderen Werten als der Call-by-Reference, wenn mehrere Parameter übergeben werden, die voneinander abhängen.

Nicht empfehlenswert, da schwer verständlich.

In der Sprache C ist ein Call-by-Name nicht möglich!

Beispiel für den Call-by-Name (1)

Beispiel-Programm in C-Notation:

```
int i, a[3];

void test (int *k, int *j) {
 *k = *k+1;
 *j = 3*a[i];
 return;
} /* test */


main () {
 a[0] = 1;
 a[1] = 2;
 a[2] = 3;
 i = 1;
 test (&i, &a[i]);
 return;
}
```


Beispiel für den Call-by-Name (2)

Call-by-Name ist in C nicht möglich. Dazu müsste **`*j`** im Unterprogramm **`test`** nach der Anweisung **`*k=*k+1`** erneut ausgewertet werden, d.h., da **`*j`** für **`a[i]`** steht und **`i=*k`** ist, wäre **`j=a[2]`**. Damit würde das Programm folgende Ergebnisse liefern:

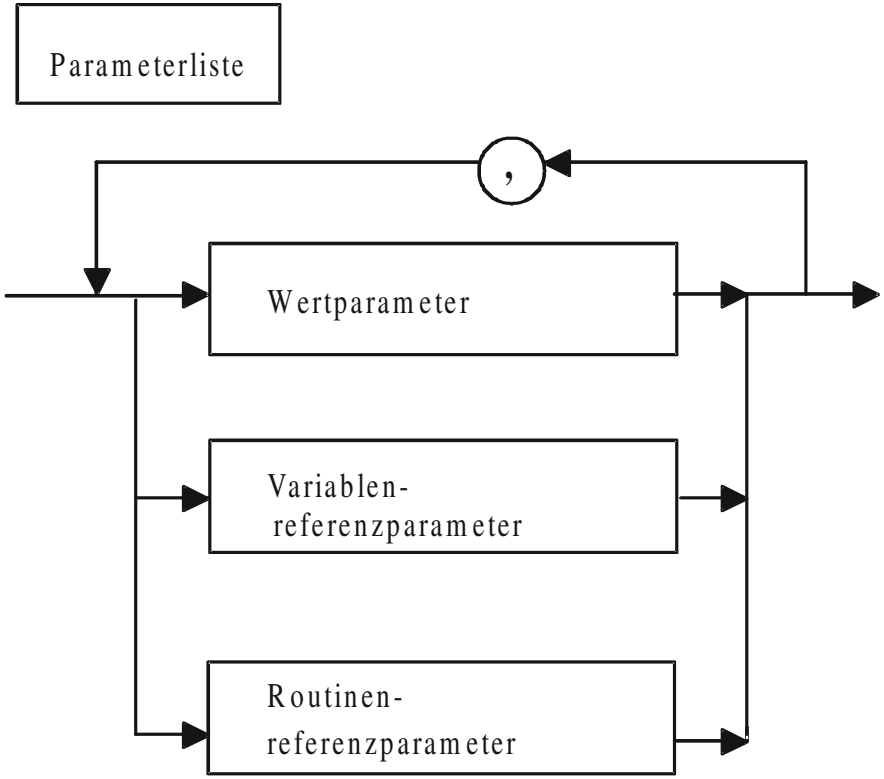
	a[0]	a[1]	a[2]
2	1	2	9

Binderegeln in C (1)

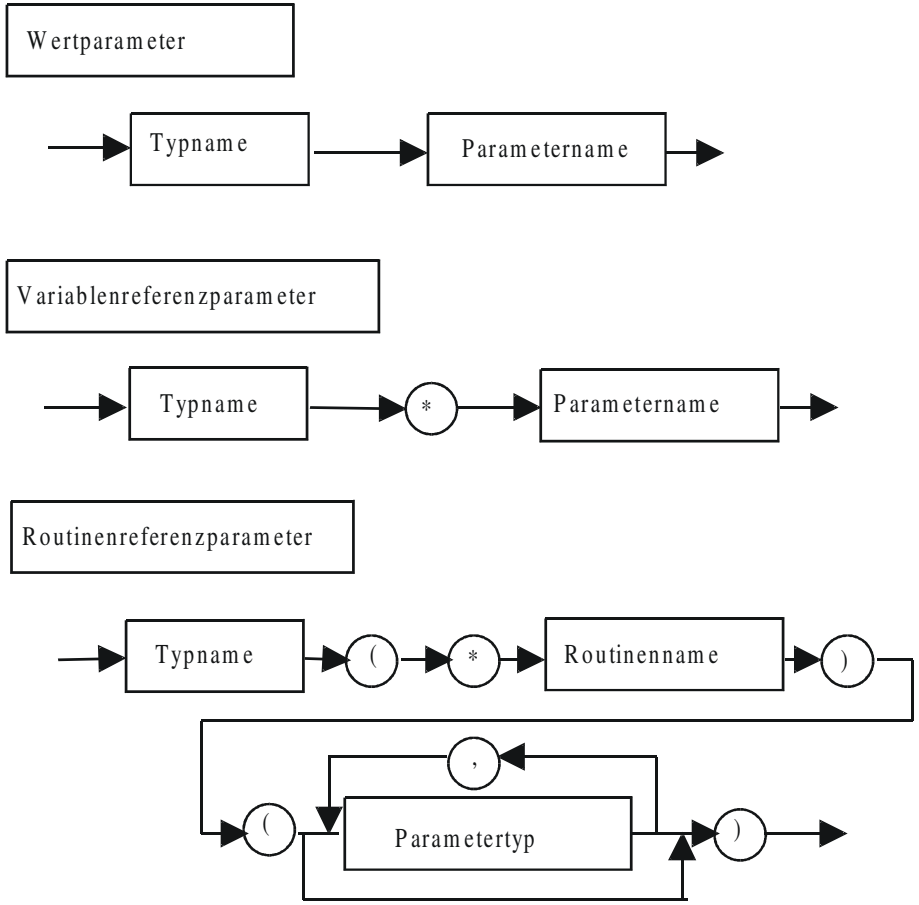
Es gibt grundsätzlich in C nur den Call-by-Value.

Ein Call-by-Reference kann durch die Übergabe von Zeigern (Speicheradressen erreicht werden!

Binderegeln in C (2)



Binderegeln in C (3)



Beispiele: Binderegeln in C (1)

a) Call-by-Value

```
float betrag (float x)
{
if (x >= 0)
   return(x);
else
   return(-x);
}
```

Aufgerufen wird die Funktion zum Beispiel durch

```
y = betrag(3.14*z);
```

Beispiele: Binderegeln in C (2)

b) Call-by-Reference

```
void tausch (int *a, int *b)
{
  int hilf;

  hilf = *a;
  *a = *b;
  *b = hilf;
  return;
}
```

Aufgerufen wird die Funktion zum Beispiel durch

```
tausch (&x,&y);
```

(wobei x und y integer sind)

Beispiele: Binderegeln in C (3)

c) Funktionen als Parameter

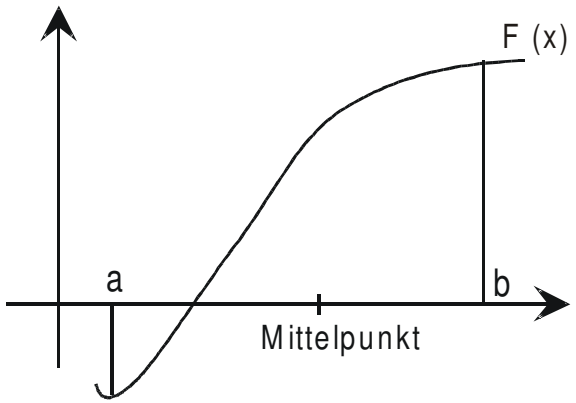
Annahme:

(*F)(float) ist eine beliebige reell-wertige Funktion mit
(*F)(a) < 0 und (*F)(b) > 0.

„**`nullstelle`**" berechnet nun nach der Methode der Intervallhalbierung eine Nullstelle von (*F).

```
float nullstelle(float(*F)(float), float a,
float b) {
   float mittelpunkt = 0.0;
   while (fabs((*F)(mittelpunkt) > (1e-10))) {
      mittelpunkt = (a+b)/2;
      if ( (*F)(mittelpunkt) < 0)
         a = mittelpunkt;
      else
         b = mittelpunkt;
      }
   return (mittelpunkt);
}
```

Beispiele: Binderegeln in C (4)



Der Aufruf geschieht z.B. durch: z = Nullstelle (cos( ), x, y);

Achtung:

Bei der Angabe der Routine **`(*F)`** als Parameter muss auf die Klammern geachtet werden, da man sonst eine Funktion angibt, die einen Zeiger auf ein **`float`** zurückgibt und nicht einen Funktionszeiger!

5.4 Standardfunktionen in C

Beispiel für Standardfunktionen

In der Standard-include-Datei <math.h> vereinbarte mathematische Standardfunktionen:

Aufruf	Parameter-typ	Ergebnistyp	Bedeutung
abs(x)	integer	integer	Betrag eines Integers
labs(x)	long	long	Betrag eines Longs
fabs(x)	float	float	Betrag eines Floats
ceil(x)	double	double	kleinster ganzzahliger Wert, der nicht kleiner als x ist
floor(x)	double	double	größter ganzzahliger Wert, der nicht größer als x ist
sin(x)	double	double	Sinusfunktion
cos(x)	double	double	Cosinusfunktion
exp(x)	double	double	Exponentialfunktion
log(x)	double	double	10er- Logarithmus
sqrt(x)	double	double	Wurzel einer Zahl
atan(x)	double	double	Arcustangens
pow(x,y)	double	double	x hoch y

5.5 Gültigkeitsbereich von Namen

Wenn Variable, Konstante usw. innerhalb einer Prozedur oder Funktion denselben Namen haben wie Variable, Konstante usw. außerhalb, muss klar sein, welches Objekt jeweils gemeint ist. Es gilt:

Jede Vereinbarung eines Namens hat nur in dem Block Gültigkeit, in dem sie vorgenommen wird.

Blöcke können geschachtelt werden.

Also: Ein Name bezieht sich immer auf die am nächsten darüber liegende Deklaration.

Namen müssen innerhalb eines Blockes eindeutig sein.

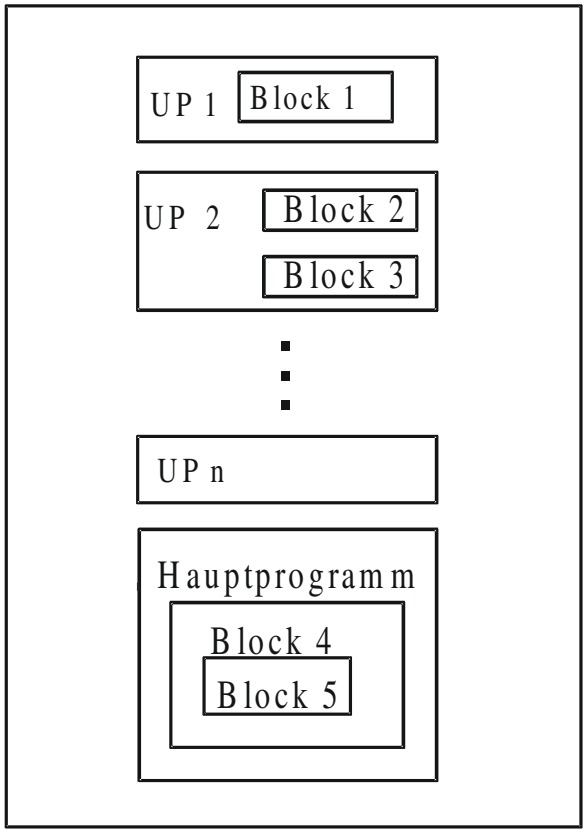
Die Deklaration muss der Verwendung vorangehen.

Ein innerhalb eines Blockes vereinbarter Name heißt **lokal**.

Ein außerhalb des Blockes vereinbarter Name heißt **global**.

Blockstruktur (1)

In C kann jeder Anweisungsblock am Beginn eigene Deklarationen enthalten.



Blockstruktur (2)

Achtung:

In ANSI-C können **Routinen nicht geschachtelt** werden.

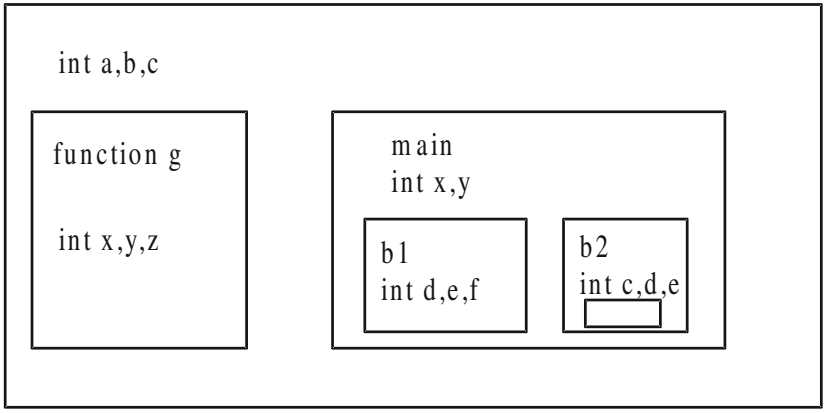
Insbesondere darf auch das Hauptprogramm keine Unterprogramme enthalten.

Es können jedoch **Anweisungsblöcke geschachtelt** werden.

Regeln für die Sichtbarkeit von Vereinbarungen

- Sichtbarkeit von äußeren Blöcken nach innen
- keine Sichtbarkeit von innen nach außen
- keine gegenseitige Sichtbarkeit für Blöcke derselben Schachtelungstiefe

Beispiel:



Gültigkeitsbereich vs. Lebensdauer

Der **Gültigkeitsbereich** eines Namens umfasst den Block, in dem der Name deklariert ist.

Die **Lebensdauer** eines Objekts umfasst den Block, in dem es definiert ist. Es existiert nur so lange, wie Anweisungen des zugehörigen Blocks ausgeführt werden. Das Laufzeitsystem von C legt beim Eintritt in einen Block den Speicherplatz für die dort lokal benötigten Objekte an und gibt ihn beim Verlassen des Blocks wieder frei.

Eine Ausnahme hierzu bildet die static-Deklaration:
Wird eine Variable innerhalb einer Funktion als **`static`** deklariert, so behält sie ihren Wert auch nach Beendigung der Funktion. Bei erneutem Aufruf hat sie den Wert vom vorigen Verlassen der Funktion.

Beispiel: static int alert;

Merke:
Objekte, die nur innerhalb eines Blocks benötigt werden, sollten innerhalb dieses Blocks vereinbart werden.

+ Übersichtlichkeit
+ Vermeidung von Seiteneffekten

5.6 Rekursion

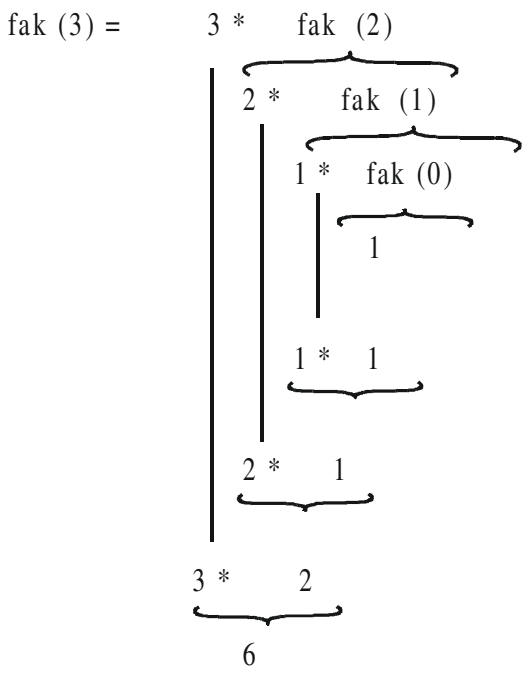
Da für alle Call-by-Value-Parameter und für alle lokalen Variablen Speicherplatz beim Prozedureintritt dynamisch angelegt wird, können Funktionen und Prozeduren in C rekursiv aufgerufen werden!

Die Anweisungen innerhalb eines Prozedurrumpfes beziehen sich dabei jeweils auf die lokalen Variablen und Parameter.

Bei der Rückkehr aus der Rekursion findet die Funktion dann jeweils wieder die alten Werte vor.

Beispiel für eine Rekursion in C

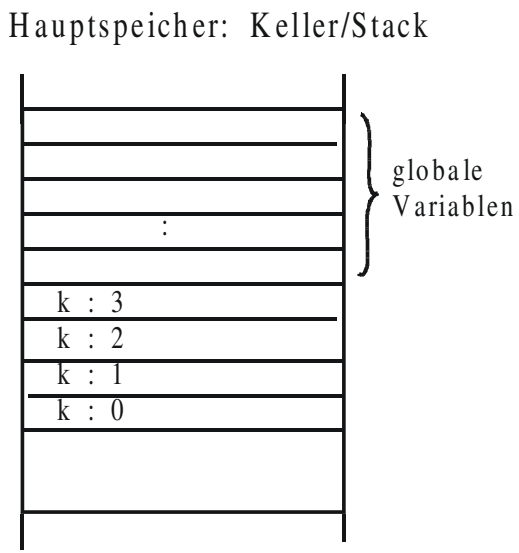
```
int fak (int k)
{
if (k==0)
   return (1);
else
   return (k*fak(k-1));
}
```



Rekursion und Kellerspeicher (Stack)

Bei jedem rekursiven Aufruf wird ein neuer Speicherbereich für die lokalen Variablen und Parameter angelegt. Zugleich werden die lokalen Variablen und Parameter aus der nächst höheren Rekursionsstufe unzugänglich. Daher lässt sich der Speicher für Unterprogrammdaten als Kellerspeicher (Stack) organisieren. Dies geschieht auch in den C-Laufzeitsystemen.

Beim Umsetzen einer Rekursion in eine Iteration muss der Kellerspeicher oft vom Programmierer „von Hand“ angelegt und verwaltet werden.



Seiteneffekte

Auswirkungen von Funktionen und Prozeduren, die nicht unmittelbar aus der intendierten Semantik hervorgehen, heißen **Seiteneffekte**. Sie treten oft auf im Zusammenhang mit

- Funktionsaufrufen und globalen Variablen
- verschachtelten Zuweisungen
  Beispiel: x = 0;
  v = --x - (x=4);
- Makros

Merke:

Externe, global gültige Variablen sind nur für große, den Kern eines Programms bestimmende Datenstrukturen sinnvoll.

Grundsätzlich sollten alle in einer Routine verwendeten Variablen entweder lokal sein, oder als Parameter übergeben werden.

Beispiel für einen Seiteneffekt (1)

```
/* Globale Variable */
int r;
float s, wert;

float hoch (float x, int y) {
 float u, v;

 u = 1;
 v = x;
 r = y;
 while (r > 0) {
    if (r%2)   /* r ungerade? */
       u = u*v;
 v = v*v;      /* v-Quadrat   */
 r = r/2;
 }
 return (u);
}
```

Beispiel für einen Seiteneffekt (2)

```
main () {
  printf ("Bitte ein Zahlenpaar eingeben;
               mit <ENTER> beenden: \n");
  scanf ("%f",&s);
  scanf ("%d",&r);
  wert = hoch (s,r);
  printf ("%f hoch %d ist %f \n", s, r,
                                   wert);
  return;
}
```

Eine Eingabe von "2 7" erzeugt eine Ausgabe von "2 hoch 0 ist 49"!