5. Unterprogramme

Dienen zur Modularisierung von Programmen.

Regel: Unterprogramme sollten nicht größer als 1 Druckseite sein.

Unterprogramme werden deklariert und aufgerufen.

Zwei Konzepte:

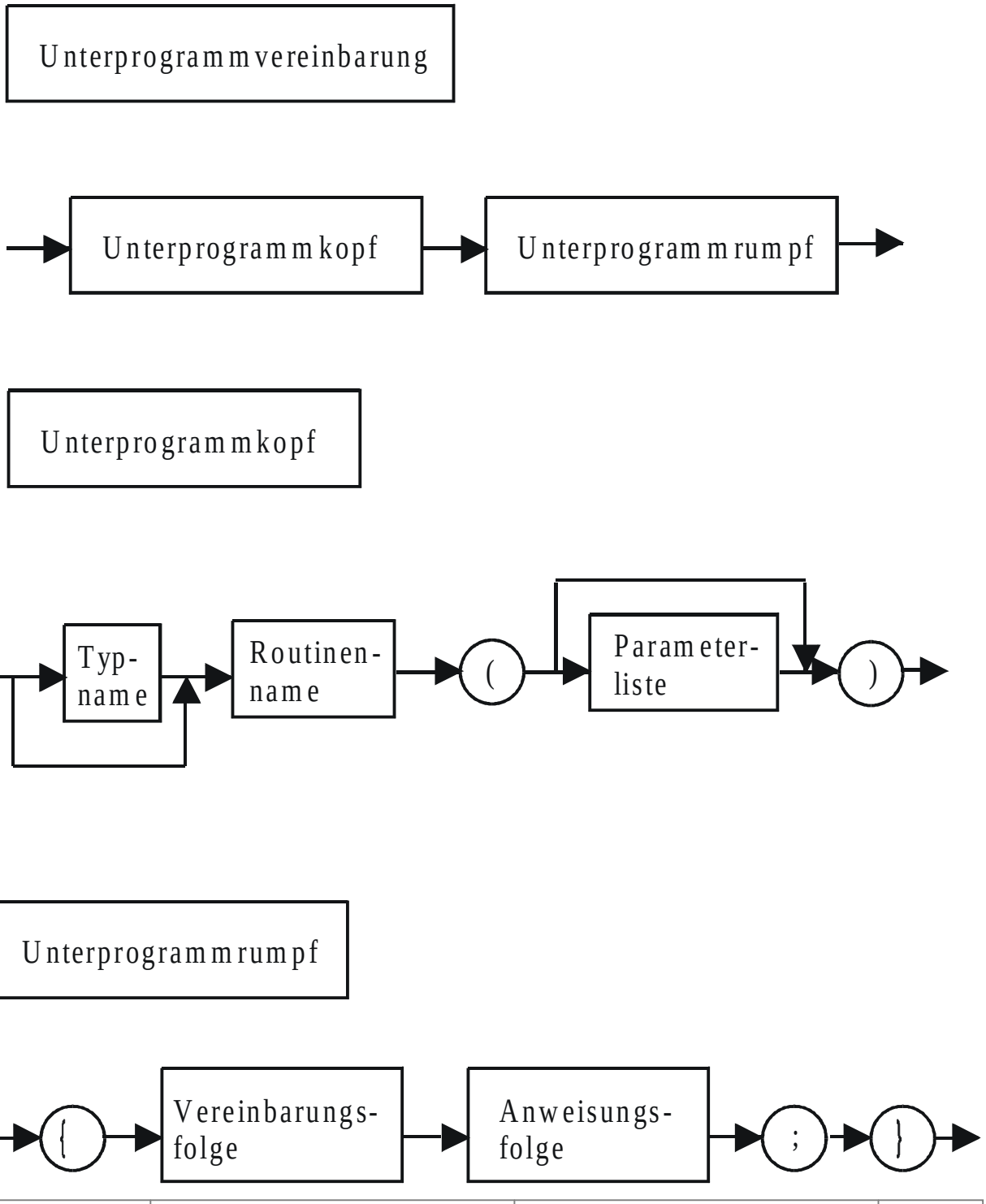
a) Prozeduren
- führen einen Teil der Arbeit des aufrufenden Programms durch
- Ergebnisse können in Parametern übergeben werden

b) Funktionen
- führen die Berechnung von Funktionswerten im mathematischen Sinn durch
- Ergebnisse werden zusätzlich zu Parametern durch den Funktionsaufruf zurückgeliefert

In C gibt es nur einen Typ von Unterprogrammen: Funktionen.

Ergebniswerte von Funktionen können in C jedoch ignoriert werden, so dass eine C-Funktion sich wie eine Prozedur verhält.

Deklaration von Unterprogrammen (1)

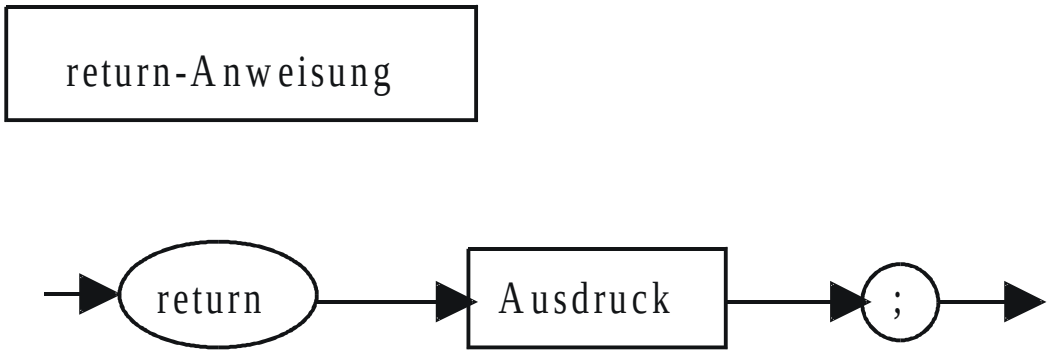


Deklaration von Unterprogrammen (2)

Unterprogramme sollten im Deklarationsteil eines Programms, im Anschluss an die Variablen-Deklaration, vereinbart werden. Es sollte immer ein Routinentyp 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

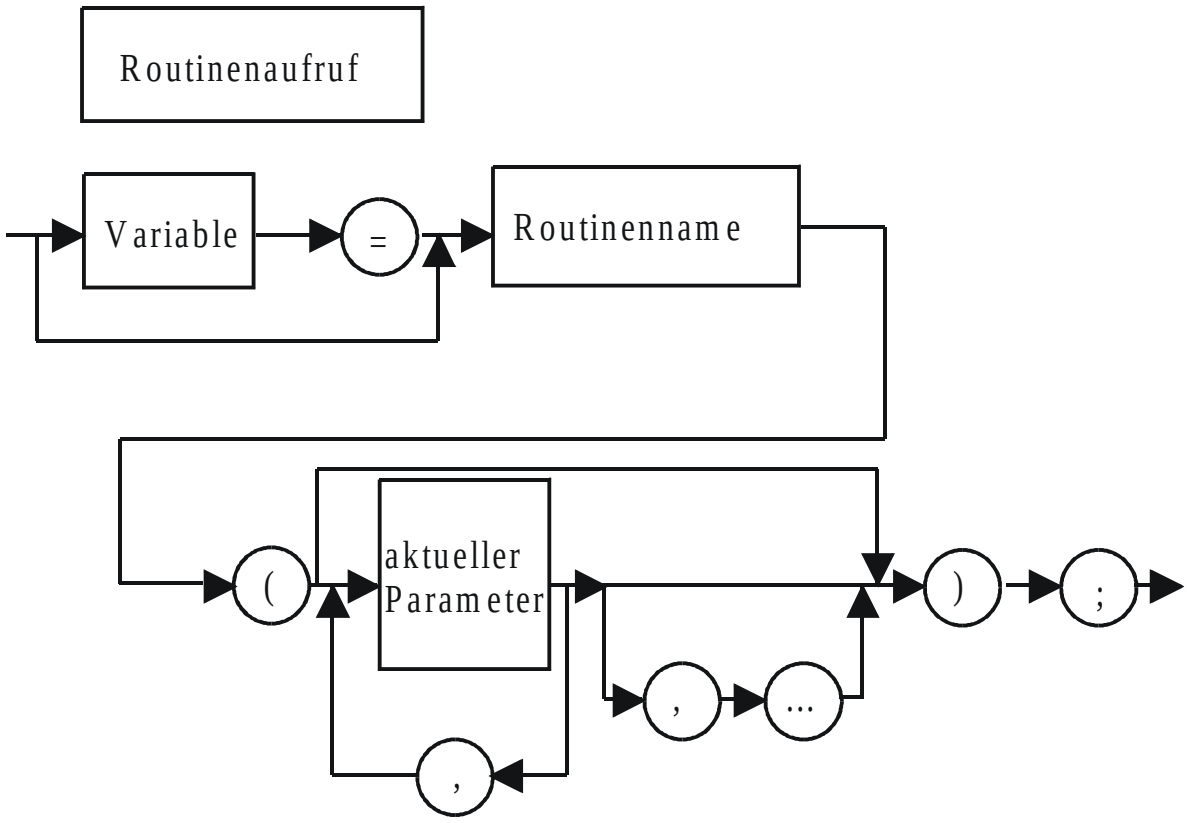
Die Rückgabe von Ergebnissen einer Funktion erfolgt in der Funktion durch die Anweisung return:



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

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

Aufruf von Unterprogrammen



Der Aufruf ohne Zuweisung an eine Variable ist, syntaktisch gesehen, eine Anweisung (statement) und fällt damit in die Prozedurklasse. Solche Routinen sollten mit dem Typ void vereinbart werden.

Der Aufruf mit Zuweisung an eine Variable ist, syntaktisch gesehen, ein Ausdruck (expression) und fällt damit in die Funktionsklasse, bei der ein Wert zurückgegeben wird. Dies wird durch Einsatz des Befehls "return" realisiert.

Beispiel: Aufruf von Unterprogrammen

```
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

- Aktuelle und formale Parameter sollten in der Anzahl übereinstimmen.

- Aktuelle und formale Parameter sollten im Typ übereinstimmen.

- Aktuelle und formale Parameter entsprechen einander in der Reihenfolge, in der sie in der Vereinbarung und im Aufruf auftreten (Stellungsparameter).

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 Argumente als vereinbart übergeben werden, da ansonsten der Effekt des Aufrufs undefiniert ist.

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 bzgl. i und a[i]:

Wie wird auf den aktuellen Parameter zugegriffen?

- indem innerhalb der Routine 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 (1)

- Argumente werden bei Routinenaufruf in die Routine auf lokalen Speicherplatz kopiert.
- Berechnung der Parameterwerte bei Aufruf der Routine.
- Alle Operationen innerhalb der Routine werden auf dem lokalen Speicherplatz ausgeführt.

Also: Keine Auswirkungen außerhalb der Prozedur.
Nur geeignet für Eingangsparameter.

Call - by – Value (2)

```
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 (1)

- bei Eintritt in die Routine wird die Speicheradresse des Parameters berechnet.

- alle Operationen innerhalb der Routine werden direkt auf den so berechneten Speicheradressen ausgeführt.

Also:

Änderungen des Parameterinhaltes ändern die Umgebung. Geeignet für Eingangs- und Ausgangsparameter, d.h. damit können Daten der aufrufenden Routine geändert werden!

Call - by - reference (2)

```
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]
2	1	9	3

Call - by – Name (1)

- Ausdrücke als aktueller Parameter erlaubt
- Operationen werden auf den Original-Speicherplätzen außerhalb der Routine ausgeführt
- erneute Auswertung des Ausdrucks bei jeder Verwendung innerhalb der Routine

Also:

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

Nicht empfehlenswert, da schwer verständlich.

Call - by – Name (2)

```
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;
}
```

Call - by – Name (3)

Bemerkung:

Call-by-name ist in C nicht möglich. Dazu müßte "* j" im Unterprogramm "test" erst 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

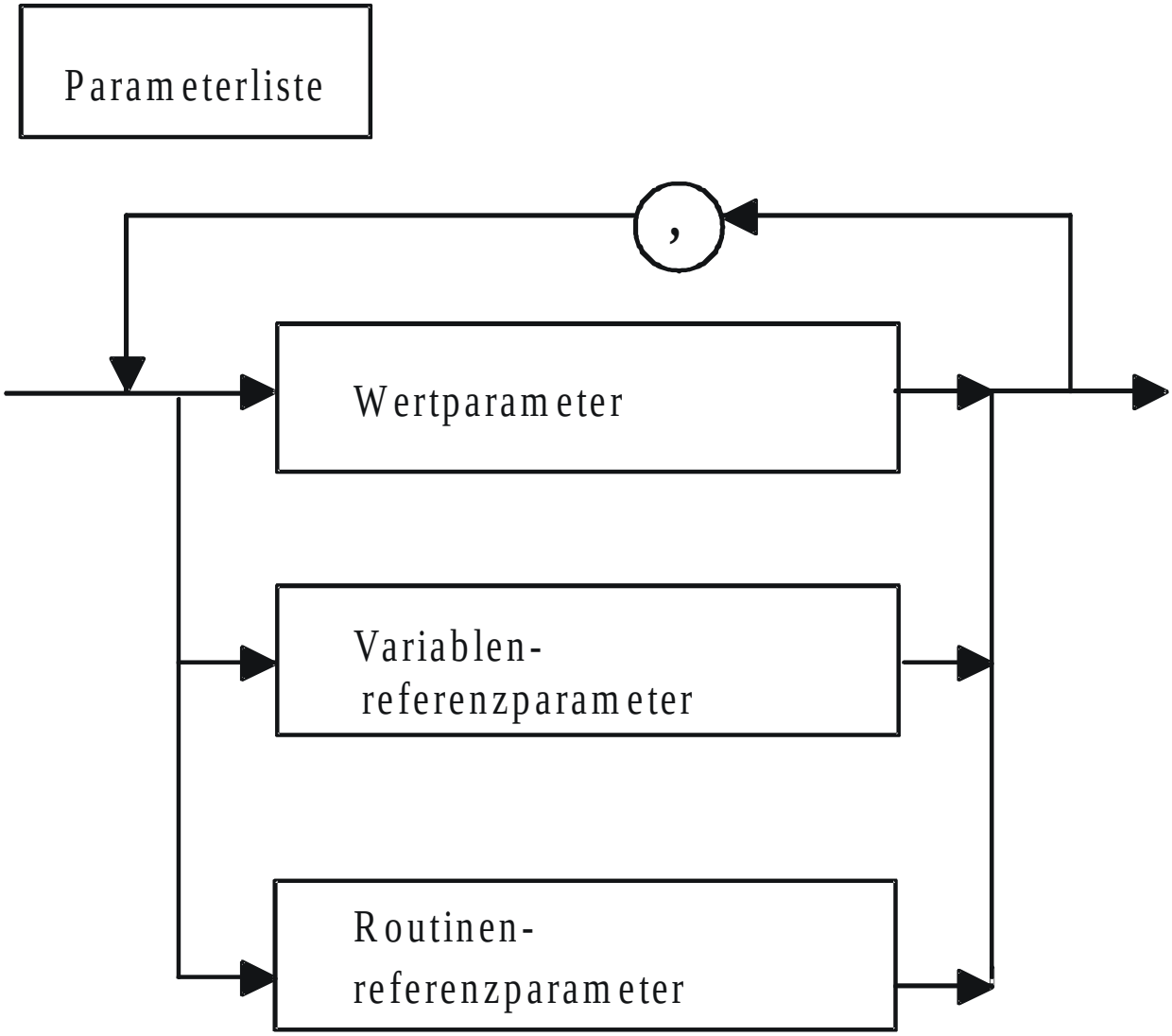
Binderegeln in C (1)

Es gibt nur Call-by-Value in C.

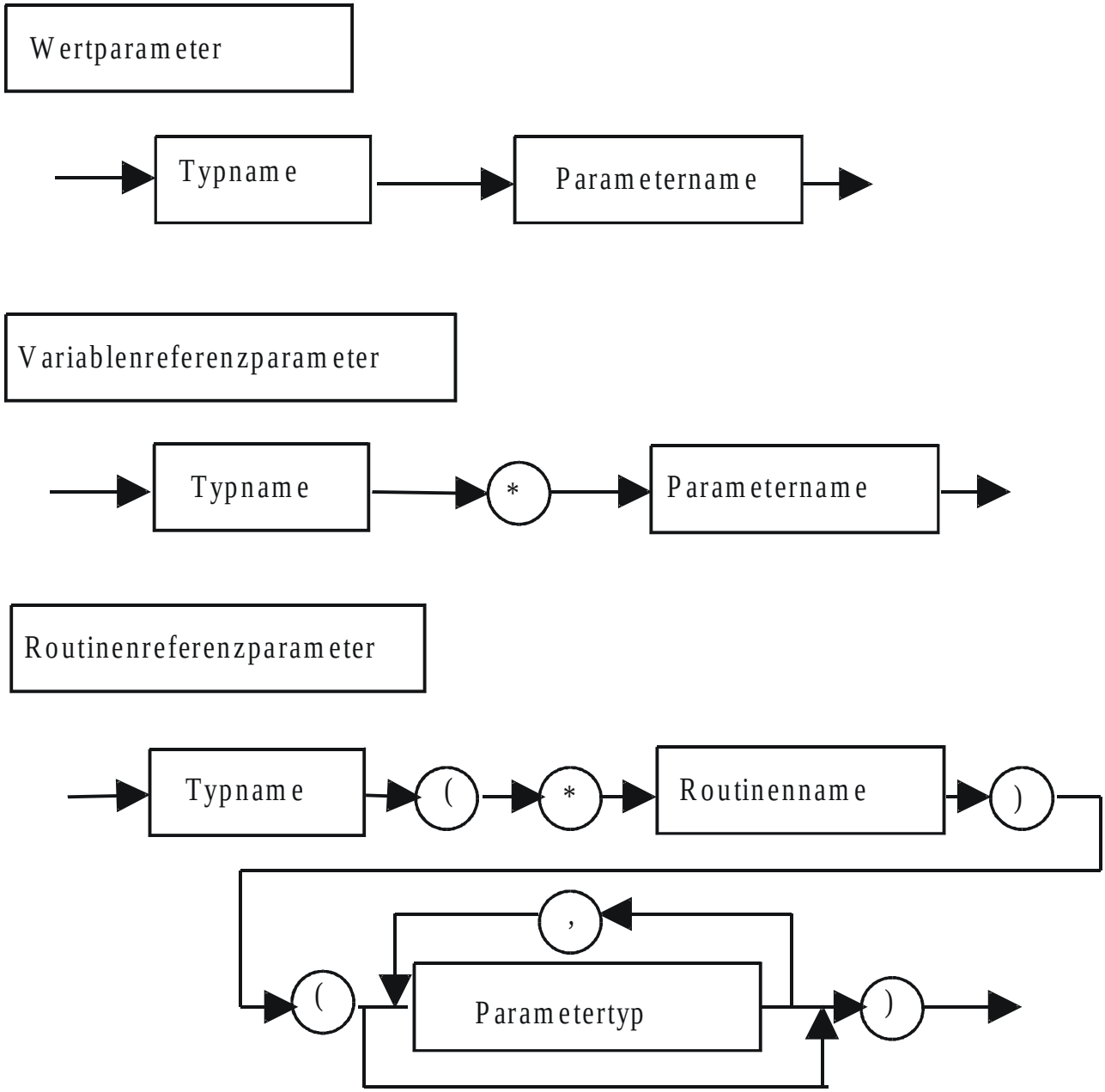
Call-by-Referenz kann durch den Referenzspeicher * dargestellt werden.

Parameterklasse	Übergaberegel	aktueller Parameter
Wertparameter	call-by-value	Ausdruck
Referenzparameter	call-by-reference	Variablenreferenz
Routinenparameter	call-by-reference	Routinenreferenz

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 z.B. 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 z.B. durch Tausch (&x, &y);
(wobei x und y integer sind)

Beispiele: Binderegeln in C (3)

c) Routinen als Parameter

Vor.:

(*F)(float) ist eine beliebige reellwertige Funktion bei der (*F)(a)<0 und (*F)(b)>0.

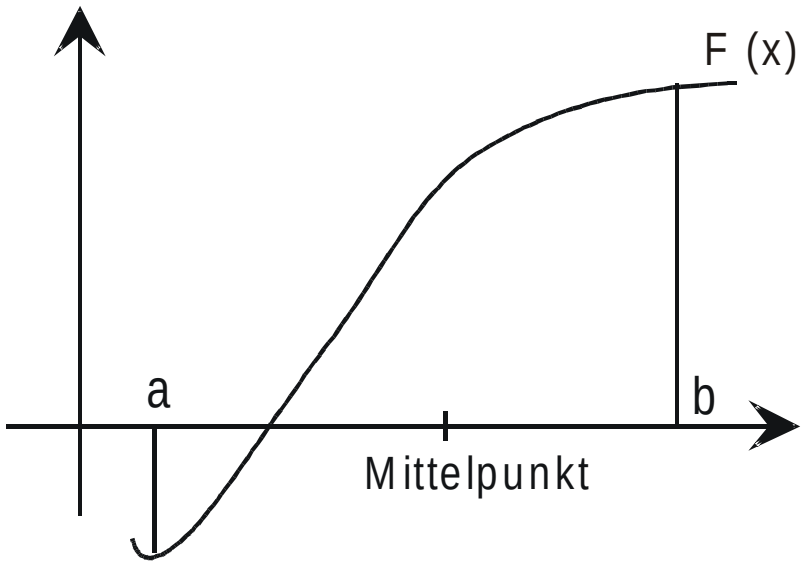
"Nullstelle" berechnet nun nach der Methode der Intervallhalbierung eine Nullstelle von (* F).

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.

Standardfunktionen in C (1)

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

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

Standardfunktionen in C (2)

In der Standard include-Datei <string. h> vereinbarte Standardfunktionen zur Handhabung von Zeichenketten:

Aufruf	Bedeutung
char *strcpy(s,ct)	Kopiert Zeichenkette ct in Vektor s
char *strncpy(ct,n)	Kopiert höchstens n Zeichen aus ct in s
char *strcat(s,ct)	Hängt Zeichenkette ct an s hinten an
char *strncat(s,ct,n)	Hängt höchstens n Zeichen an
int strcmp(cs,ct)	Vergleicht Zeichenketten cs und ct
int strncmp(cs,ct,n)	Vergleicht höchstens n Zeichen von cs und ct
char *strstr(cs,ct)	Liefert Zeiger auf erste Kopie von ct in cs
size_t strlen(cs)	Liefert Länge von cs

Gültigkeitsbereich von Namen

Wenn Variablen, Konstanten usw. innerhalb einer Prozedur oder Funktion denselben Namen haben wie Variablen, Konstanten usw. außerhalb, muß geklärt werden, welches Objekt jeweils gemeint ist. Es gilt:

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

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

Namen müssen innerhalb eines Blockes eindeutig sein.

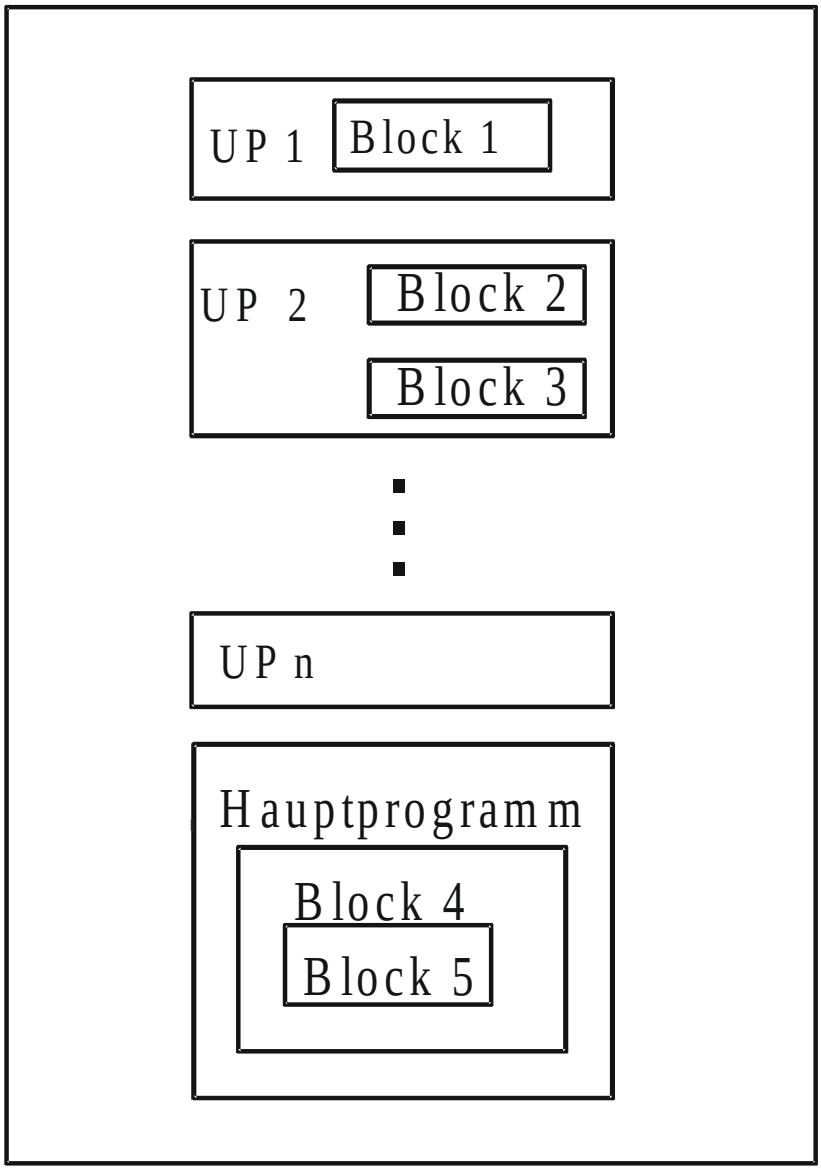
Die Deklaration muss der Verwendung vorangehen.

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. Ansonsten gibt es Vereinbarungen außerhalb von Routinen ("globale") und zu Beginn von Routinen.



Blockstruktur (2)

Achtung:

In ANSI-C können Routinen nicht geschachtelt werden.

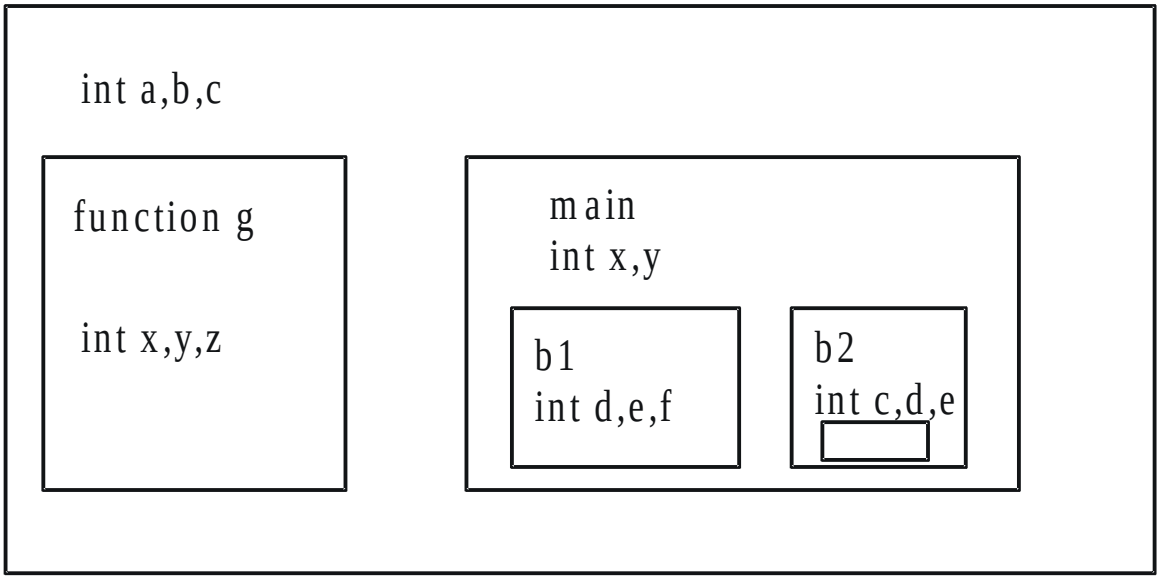
Insbesondere darf auch das Hauptprogramm keine Unterprogramme enthalten.

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 Prozedur.

Beispiel: static int alert;

Merke:

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

+ Übersichtlichkeit
+ Vermeidung von Seiteneffekten

Rekursion in C

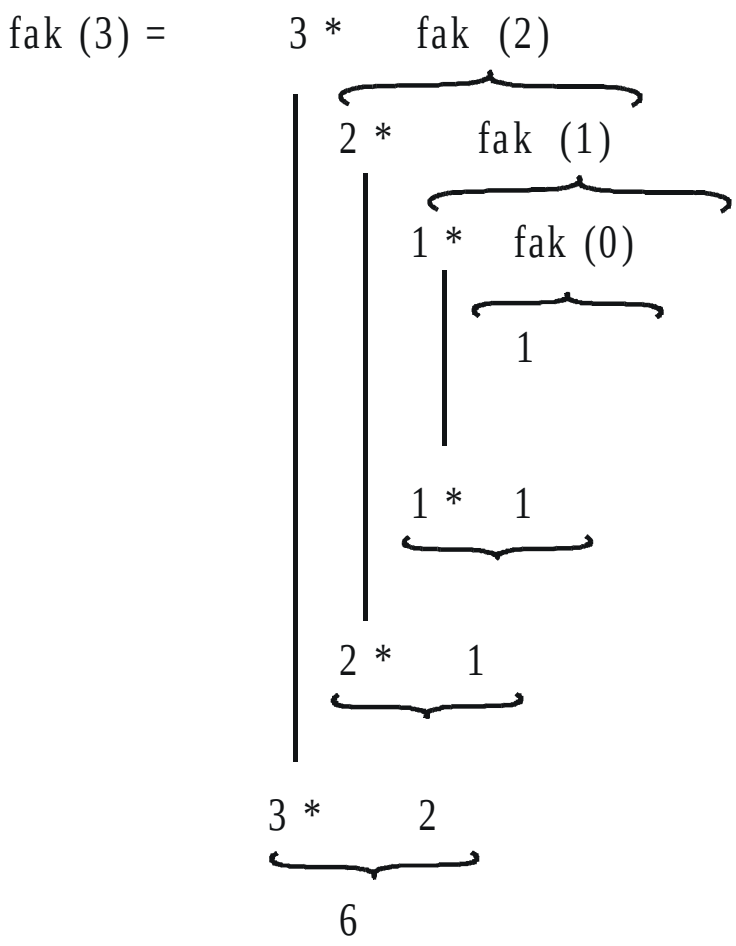
Da für alle Call-by-Value-Parameter und für alle lokalen Variablen Speicherplatz beim Prozedureintritt dynamisch angelegt wird, können Funktionen und Prozeduren in C rekursiv aufgerufen werden!

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

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

Beispiel: Rekursion in C

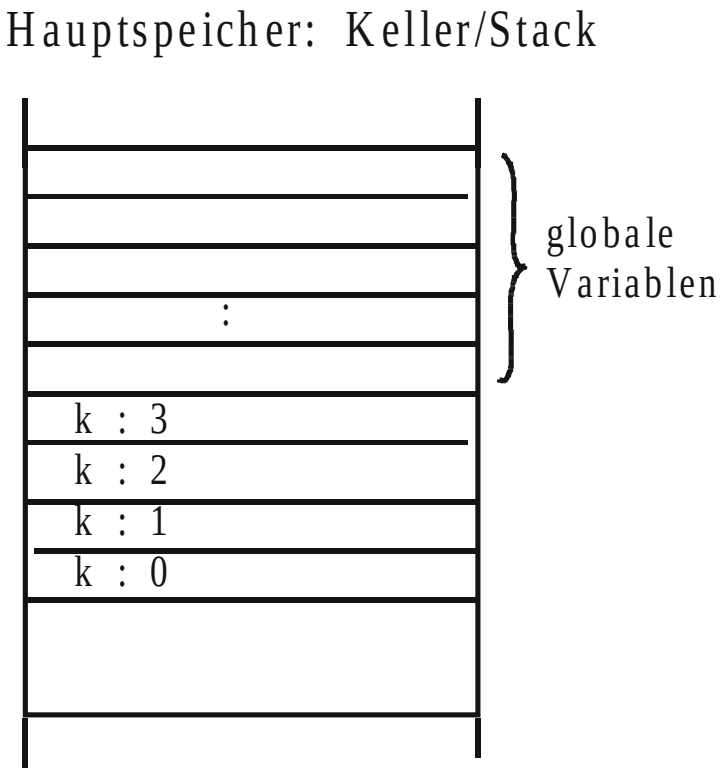
```
int fak (int k)
{
        if          (k==0)
                     return (1);
        else
                     return (k * fak (k-1));
}
```



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 nächsthöheren 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 angelegt und verwaltet werden.



Seiteneffekte

Auswirkungen von Funktionen und Prozeduren, die nicht unmittelbar aus der intendierten Semantik hervorgehen, heißen Seiteneffekte. Sie treten meist 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 Variablen */
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 ist 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"!