Die Programmiersprache C

Abgeleitete Datentypen und Funktionen

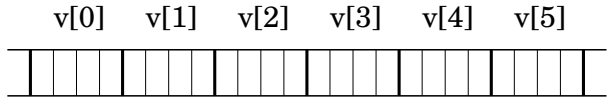
Feldtypen

Zu jedem Typ T kann man ein **Feld von** T erzeugen (man spricht auch von einem **Vektor** oder **Array**).

Bei Feldern handelt es sich um eine Ansammlung von Objekten gleichen Typs, wobei die Objekte in aufeinanderfolgenden Speicherbereichen abgelegt werden.

Ein Element eines Feldes wird über einen Index angesprochen, der in eckigen Klammern angegeben wird.

Beispiel:
`int v[6]` definiert



Feldtypen (2)

Beispiel: Im Folgenden wird ein Feld vom Typ `int` der Größe 10 deklariert. Der Name des Feldes ist `vektor`.

```
int vektor[10];
vektor[0] = 100;
vektor[1] = 2 * vektor[0];
vektor[2] = vektor[0] + vektor[1];
```

Das Array `int a[N]` besitzt die Elemente `a[0]` bis `a[N-1]`.
Vorsicht: Es erfolgt keine Bereichsüberprüfung.

Initialisierung bei der Deklaration:

```
float x1[4] = {1.0, 2.0, 3.0, 4.0};
int   x2[]  = {1, 2, 3, 4};
char  x3[]  = "Hello, world!";
```


Feldtypen (3)

Anmerkungen:

- **Arrays lassen sich nicht mit = zuweisen.**

  Die Anweisung `int a[20];` legt `a` als Name des Arrays fest: Die Inhalte von `a` können verändert werden, nicht `a` selbst.

- **Arrays lassen sich nicht mit == vergleichen.**

  Der Vergleich zweier Arrays ist zwar syntaktisch korrekt, liefert aber in der Regel nicht das gewünschte Ergebnis (siehe Abschnitt „Zeiger und Referenzen“).

Mehrdimensionale Felder

Mehrdimensionale Felder: Anhängen weiterer Indizes.

Beispiel: zweidimensionales Feld `matrix` vom Typ `int`

```
int matrix[10][20];
matrix[0][0] = 42;
matrix[1][0] = matrix[0][0] + matrix[0][1];
```

Mehrdimensionale Arrays und ihre Initialisierung:

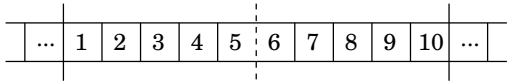
```
float x1[2][4] = { {1.0, 2.0, 3.0, 4.0},
                   {5.0, 6.0, 7.0, 8.0} };
int   x2[][4]   = { {1, 2, 3, 4}, {5, 6, 7, 8} };
char  x3[][7]   = { "Hello,", "world!" };
```


Mehrdimensionale Felder: Anmerkungen

Felder werden zeilenweise abgespeichert. Der letzte Index ändert sich schneller als der erste: `int a[#zeilen][#spalten]`

Beispiel:

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



Initialisierung: Nur die erste (äußerste) Größenangabe darf weggelassen werden. Der Rest nicht, da sonst der Index-Operator `[]` die Position nicht berechnen kann.

Beispiel: Position von `a[i][j]`: `i * #spalten + j`

Aufzählungstyp

Der `enum`-**Typ**: Konstanten werden Integer-Werte zugewiesen, Konstanten sind ansprechbar über Bezeichner.

Beispiel:

```
enum Tag {Mo, Di, Mi, Do, Fr, Sa, So};
```

Die Werte werden intern auf fortlaufende, nicht-negative ganze Zahlen abgebildet, beginnend bei 0. Im Beispiel: `Mo == 0`, `Di == 1`, `Mi == 2` usw.

Eine solche Abbildung kann auch direkt definiert werden:

```
enum Tag {Mo=1, Di=2, Mi=4, Do=8, Fr=16,
       Sa=32, So=64};
```


Aufzählungstyp (2)

Jeder Name darf nur einmal verwendet werden, d.h. Namen in verschiedenen Aufzählungen müssen sich unterscheiden.

Die Werte in einer Aufzählung können gleich sein.

Variablen können mit Aufzählungstypen deklariert werden.

- Sie unterliegen nicht notwendigerweise der Typprüfung.
- Eine Aufzählungskonstante des Typs kann als Wert zugewiesen werden.
- Sie können in logischen Ausdrücken verglichen und in arithmetischen Ausdrücken verknüpft werden.

Aufzählungstyp (3)

Beispiel:

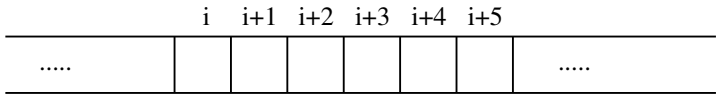
```
enum Tag {Mo, Di, Mi, Do, Fr, Sa, So};
enum Monat {Jan, Feb, Mar, Apr, May, Jun,
        Jul, Aug, Sep, Oct, Nov, Dec};

enum Tag t = Mo;
enum Monat m = Jan;

if (t == Di) ...
if (t == m) ...      /* ist erfüllt: Mo == 0 == Jan */
t = Jan + 3;         /* kein Fehler, keine Warnung  */
```


Zeiger und Adressen

Für jeden Typ T kann man einen **Zeiger auf** T erzeugen. Der Wert eines Zeigertyps ist die Adresse eines Objekts.

Spezieller Wert: der leere Zeiger (NULL-Zeiger) $\rightarrow$ die Konstante 0. Besser: logische Konstante NULL (hat den Wert 0 und ist in `<stddef.h>` definiert).

Vereinfachtes Bild der Speicherorganization:



Speicherzellen sind fortlaufend nummeriert und adressierbar, sie können einzeln oder in zusammenhängenden Gruppen bearbeitet werden.

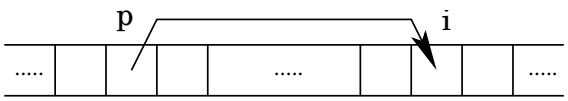
Zeiger und Adressen (2)

Der **Adressoperator** `&` liefert die Adresse eines Objekts.

Der **Inhaltsoperator** `*` liefert das Objekt, das unter einer Adresse abgelegt ist.

Beispiel:

```
int i;     /* Variable für Integer-Wert */
int *p;    /* Zeiger auf einen Integer-Wert */

p = &i;    /* p = Adresse von i: p zeigt auf i */
i = *p;    /* i = Inhalt der Adresse p */
```



Zeiger und Adressen (3)

Die Syntax der Variablenvereinbarung `int *p` imitiert die Syntax von Ausdrücken, in denen die Variable auftreten kann: **Der Ausdruck `*p` ist ein `int`-Wert.**

Daraus folgt:

- Ein Zeiger darf nur auf eine Art von Objekt zeigen.
- Jeder Zeiger zeigt auf einen festgelegten Datentyp.

Ausnahme: Ein **Zeiger auf** `void`

- nimmt einen Zeiger beliebigen Typs auf,
- darf aber nicht selbst zum Zugriff verwendet werden.

Zeiger und Adressen (4)

```
void *v;
int i = 1;
double d = 2.0;

v = &i;                     /* v zeigt auf i */
*(int *) v += 1;            /* cast notwendig: i += 1 */
printf("%d, %d\n", i, *(int *) v);

v = &d;                     /* v zeigt auf d */
*(double *) v += 1.0;       /* cast notwendig: d += 1 */
printf("%f, %f\n", d, *(double *) v);
```


Zeiger und Adressen (5)

Anmerkungen:

- Der Adressoperator `&` kann nur auf Objekte im Speicher angewendet werden, auf Variablen und Vektorelemente.

  Er kann nicht auf Ausdrücke, Konstanten oder `register`-Variablen angewandt werden.

- Die unären Operatoren `*` und `&` haben höheren Vorrang als die arithmetischen Operatoren.

  Beispiele:

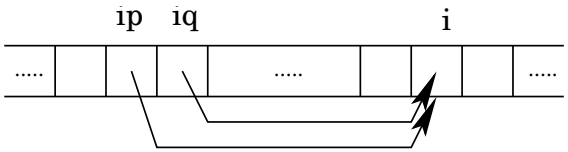
  * `*ip + 10` addiert 10 zu dem Wert, auf den `ip` zeigt.
  * `*ip += 1` inkrementiert den Wert, auf den `ip` zeigt.

Zeiger und Adressen (6)

Anmerkung: Zeiger sind Variablen und können folglich zugewiesen und verglichen werden.

Beispiel:

```
int i, *ip, *iq;

ip = &i;            /* ip zeigt auf i */
iq = ip;            /* iq zeigt auch auf i */
```

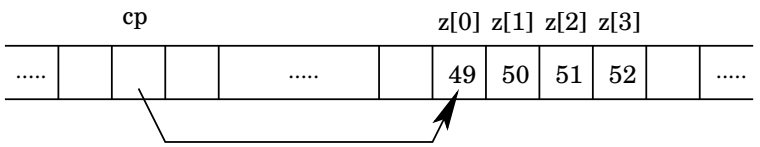


Arrays und Zeiger

Eine Array-Deklaration definiert einen Zeiger auf das erste Element des Arrays.

Beispiel:

```
char *cp;
char z[4] = {'1', '2', '3', '4'};

cp = z;             /* cp zeigt auf z[0] */
```



Arrays und Zeiger (2)

Per Sprachdefinition gilt: Zeigt `p` auf ein Element eines Vektors, dann zeigt `p + j` auf das `j`-te Element hinter `p`.

Vorsicht: Es sind auch negative Werte für `j` zulässig und es findet keine Bereichsüberprüfung statt.

Beispiel:

```
char *cp;
char z[4] = {'1', '2', '3', '4'};

cp = &z[1];       /* cp zeigt auf z[1] */
cp++;             /* cp zeigt auf z[2] */
printf("z[2] = %c, z[-8] = %c\n", *cp, *(cp-10));
```


Arrays und Zeiger (3)

Vorsicht: Arrays lassen sich nicht mit == vergleichen.

Beispiel:

```
int a[3] = {1, 2, 3};
int b[3] = {1, 2, 3};

if (a == b) ...
```

Beispiel:

```
char c1[10], c2[10];

scanf("%s", c1);
scanf("%s", c2);
if (c1 == c2) ...
```


Arrays und Zeiger (4)

Vorsicht: Eindimensionale Arrays können auch als Zeiger deklariert werden, aber

- Zeichenkonstanten liegen im **Programmsegment**,
- Arrays werden im **Datensegment** abgelegt.

Beispiel:

```
char ar[] = "Riker";
char *st = "Worf";

ar[0] = 'X';  /* o.k. */
st = "Troi";  /* o.k. */
st[0] = 'X';  /* nein! */
```

Beispiel:

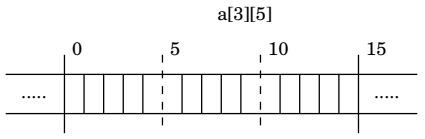
```
char *c1 = "Picard";
char *c2 = "Picard";

if (c1 == c2) ...
```

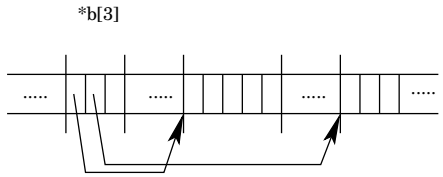

Zeiger vs. mehrdimensionale Vektoren

Unterscheide 2D-Vektor und Vektor von Zeigern:

```
int a[3][5];
int *b[3];
```

- Legitim: `a[2][4]` und `b[2][4]`.
- Aber `a` ist ein echter 2D-Vektor:
  * `a` stellt für 15 `int`-Werte Speicher bereit.
  * `b` reserviert nur Speicherplatz für 3 Zeiger.

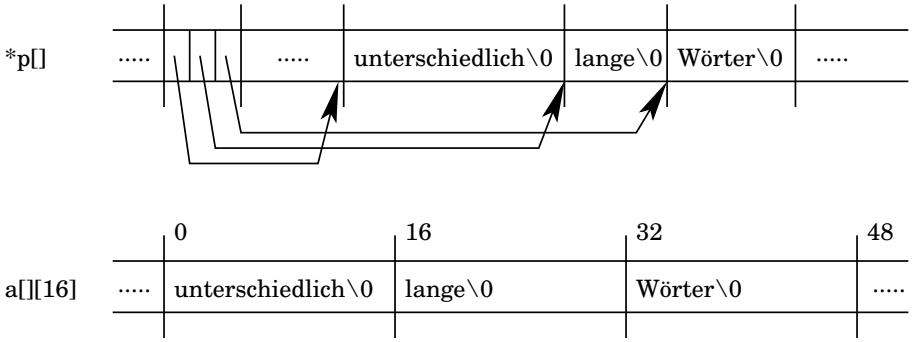




Zeiger vs. mehrdimensionale Vektoren (2)

Oft werden Zeigervektoren benutzt, um Zeichenketten unterschiedlicher Länge zu speichern.

```
char *p[] = {"Unterschiedlich", "lange", "Wörter"};
char a[][16] = {"Unterschiedlich", "lange", "Wörter"};
```



Strukturen

Zusammenfassung von Daten unterschiedlichen Typs.

Syntax:

```
struct <name> {
    <member_1>;
    ....
    <member_n>;
}
```

Beispiel:

```
struct adresse {
    char *str, *ort;
    int hnr, plz;
    long tel;
}
```

Erklärung: (Semantik)

- `<name>` ist der **Typname der Struktur** (kann entfallen)
- `<member_i>` ist eine Variablendeklaration
- die Variablen der Struktur heißen **Komponenten**
- ok: gleiche Komponente in verschiedenen Strukturen

Strukturen (2)

Strukturen ermöglichen die **Organisation von Daten**.

Eine `struct`-Vereinbarung definiert einen Datentyp.

Hinter der Komponentenliste kann eine Liste von Variablen stehen, genau wie bei einem elementaren Datentyp:

Beispiel:

```
int a, b, c;
```

```
struct complex {
    double re, im;
} x, y, z;
```

Beide Definitionen vereinbaren Variablen des angegebenen Typs und reservieren Speicherplatz.

Strukturen (3)

Die Komponenten einer Struktur werden mit dem Punktoperator . angesprochen.

- linker Operand: **Strukturvariable**
- rechter Operand: **Komponentenname**

Beispiel:

```
struct complex {
    double re, im;
} x, y, z;

x.re = 1.0;
x.im = 2.0;
```

Strukturen (4)

Strukturen dürfen andere Strukturen enthalten:

```
struct mitarbeiter {
    char *name, *vorname, *persnr;
    struct adresse adresse;
    int gehalt;
}
```

Strukturen können in Vektoren gespeichert werden:

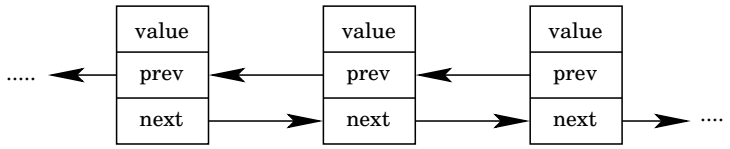
```
struct mitarbeiter personal[20];
...
for (i = 0; i < 20; i++)
    printf("Name[%d] = %s\n", i, personal[i].name);
```


Strukturen (5)

Vorsicht: Strukturen dürfen sich nicht selbst enthalten!
→ wieviel Speicherplatz soll bereitgestellt werden?

Rekursive Strukturen: In der Strukturdeklaration wird ein Zeiger auf die Struktur selbst vereinbart. **Beispiel:**

```
struct liste {
    struct mitarbeiter value;
    struct liste *prev, *next;
} fischer, meier, schulze;
```



Strukturen (6)

Abkürzende Schreibweise für Zeiger auf Strukturen:
`(*name).komponente` entspricht `name->komponente`

Beispiel:

```
fischer.next = &meier;
fischer.prev = 0;
fischer.value.name = "Fischer";
...
schulze.next = 0;
schulze.prev = &meier;
schulze.value.name = "Schulze";
for (l = &fischer; l != NULL; l = l->next)
    printf("%s\n", l->value.name);
```


Strukturen (7)

`typedef`: weise einer Struktur ein **Synonym** zu
→ das Synonym kann genauso genutzt werden, wie ein einfacher Datentyp.

Beispiele:

```
typedef struct {
    char *str, *ort;
    int hnr, plz;
    long tel;
} adresse_t;

adresse_t adr;
adr.str = "Waldweg";
...
```

```
typedef struct {
    char *name, *vname, *pnr;
    adresse_t adresse;
    int gehalt;
} mitarbeiter_t;

mitarbeiter_t arb;
arb.name = "Müller";
...
```

Strukturen (8)

Soll einer rekursiven Struktur mittels `typedef` ein Synonym zugewiesen werden, darf der Strukturname **nicht entfallen**.

Beispiel:

```
typedef struct liste {
    mitarbeiter_t value;
    struct liste *prev, *next;
} liste_t;

liste_t l;
mitarbeiter_t fischer, meier, schulze;
```


Strukturen (9)

Initialisierung: Der Struktur-Definition folgt eine Liste von konstanten Ausdrücken für die Komponenten.

Beispiele:

```
adresse_t a = {"Am Bach", "Aldrup", 4, 12345, 4711};

mitarbeiter_t m = {"Fischer", "Anna", "08F42W",
   {"Am Bach", "Aldrup", 4, 12345}, 2500};

liste_t l = {{"Fischer", "Anna", "08F42W",
   {"Am Bach", "Aldrup", 4, 12345, 4711}}, 0, 0};
```

Anmerkungen:

- Die Reihenfolge der Komponenten ist zu beachten!
- Es müssen nicht alle Komponenten initialisiert werden.

Dynamische Speicheranforderung

Arrays: zur Übersetzungszeit muss bekannt sein, wie viele Elemente gespeichert werden sollen (nicht in C99)

In der Regel ist diese Größe nicht vorhersagbar.
→ Speicher wird zur Laufzeit (dynamisch) angefordert

`void * malloc(size_t size)` liefert Zeiger auf Speicherbereich der Größe `size`, oder `NULL`, wenn die Anfrage nicht erfüllt werden kann. Der Bereich ist nicht initialisiert.

Beispiel:

```
int *pa, i, len;
...
pa = (int *) malloc(len * 4);
if (pa != NULL) ...
```


Dynamische Speicheranforderung (2)

Der unäre Operator `sizeof` liefert die Anzahl der Bytes für ein Objekt oder einen Typ.

Syntax:

```
sizeof <object>
sizeof(type)
```

Beispiel:

```
int i, j;
j = sizeof i;
j = sizeof(int);
```

`sizeof` darf nicht auf Operanden vom Typ Funktion, unvollständige Typen (Felder ohne Größenangabe) oder `void`-Typen angewendet werden.

Beispiel:

```
pa = (int *) malloc(len * sizeof(int));
v = malloc(len * sizeof(int []));   /* falsch! */
```

Dynamische Speicheranforderung (3)

Die Funktion `void free(void *p)` gibt den Speicherbereich, auf den `p` zeigt, wieder frei. Der Speicherbereich muss über einen Aufruf von `malloc` allokiert worden sein!

Beispiel:

```
adresse_t *adr;
...
adr = (adresse_t *) malloc(len * sizeof(adresse_t));
if (adr != 0) {
    adr[0].str = "Waldweg";
    adr[0].hnr = 13;
...
free(adr);
```


Vereinigungstyp

Der **Typ** `union` hat die gleiche Syntax wie der Strukturtyp.

Die einzelnen Komponenten werden **nicht hintereinander**, sondern beginnend an derselben Speicherstelle abgelegt.

Beispiel:

```
union zahlendarstellung {
    unsigned int zahl;
    unsigned char c[4];
} test = {123456};

main() {  .....
    for (i = 0; i < 4; i++)
        printf("Byte %d = %d\n", i, test.c[i]);
}
```


Vereinigungstyp (2)

Die Speicherbereiche überschneiden sich: In der Regel besitzt nur eine Komponente einen sinnvollen Wert.

Beispiel: In einer Symboltabelle sollen die Werte von Programmkonstanten verwaltet werden. Mögliche Typen: `int`, `float` und `char *`.

```
union Value {
    int ival;
    float fval;
    char *cval;
} u;
```

Problem: Der Datentyp, der entnommen wird, muss der Typ sein, der als letzter zuvor gespeichert wurde.

Vereinigungstyp (3)

Der Programmierer muss verfolgen, welcher Typ im `union` gespeichert ist.

Vereinigungstypen können innerhalb von Strukturen und Vektoren auftreten und umgekehrt.

Beispiel:

```
enum SymbolType = {INT, FLOAT, STRING};
struct Symbol {
   char *name;
   enum SymbolType type;
   union Value value;
};
struct Symbol tab[20];
```

Vereinigungstyp (4)

Sicherstellen der Konsistenz ist zum Teil mit erheblichem Aufwand verbunden:

```
for (i = 0; i < 20; i++) {
   printf("%2d: Name = %s, Typ = %d, ",
          i, tab[i].name, tab[i].type);
   if (tab[i].type == INT)
      printf("Wert = %d\n", tab[i].value.ival);
   else if (tab[i].type == FLOAT)
      printf("Wert = %f\n", tab[i].value.fval);
   else printf("Wert = %s\n", tab[i].value.cval);
}
```


Vereinigungstyp (5)

Operatoren:

- zuweisen oder kopieren als Ganzes
- berechnen der Adresse
- Zugriff auf eine Alternative

Syntax: Die Alternativen eines Vereinigungstyps werden angesprochen wie die Komponenten einer Struktur:

```
varname.alternative
varzeiger->alternative
```

Initialisierung: Ein Vereinigungstyp kann nur mit einem Wert initialisiert werden, der zum Typ der ersten Alternative passt (siehe Beispiel Zahlendarstellung).

Funktionen

Funktionen erledigen eine abgeschlossene Teilaufgabe.

Funktionen zerlegen Programme in kleinere Einheiten und erhöhen so die Übersichtlichkeit und die Wartbarkeit von Programmen! → strukturierte Programmierung

Vergleich:

- Funktionen: Programme organisieren
- Strukturen: Daten organisieren

Oft gebrauchte Funktionen und Daten sind in Bibliotheken (Libraries) bereitgestellt: `stdio`, `stdlib`, `string`, `math`, ...

Bei gut entworfenen Funktionen reicht es zu wissen **was** getan wird, gleichgültig **wie** eine Aufgabe gelöst wird.

Funktionen (2)

Syntax:

```
Rückgabetyp Funktionsname( Parameterliste ) {
    Vereinbarungen
    Anweisungen
}
```

Beispiel:

```
int max(int a, int b) {
    if (a > b)
        return a;
    return b;
}
```

Eine Funktion kann mit `return` einen Wert zurückgeben.

Funktionen: Struktogramm

Definition:

Prozedur ABC(IN x, OUT y)
Function ABC(IN x, OUT y)
RETURNS int: Erläuterungen

Zweck: Erläuterungen

Parameter:
IN x: Erläuterungen
OUT y: Erläuterungen

benutzte Prozeduren: Erläuterungen
benutzte Funktionen: Erläuterungen
benutzte Variablen: Erläuterungen

y := x * (y + x)

for i := 1, (1), x
    y = y + i * i

Aufruf:

for i := 1, (1), n
    a := i * 2 + 1
    x := ABC(IN a, OUT b)

Funktionen (3)

Gültigkeitsbereich von Bezeichnern:

Parameternamen und **deklarierte lokale Variablen** sind nur innerhalb der Funktion bekannt und für andere Funktionen nicht sichtbar! → andere Funktionen können dieselben Namen ohne Konflikte nutzen!

Rückgabewerte:

- Eine Funktion muss keinen Rückgabewert liefern.
- An der aufrufenden Stelle darf der Rückgabewert einer Funktion ignoriert werden.
- Fehlendes `return`: undefinierter Rückgabewert

Funktionen (4)

Funktionen müssen (wie Variablen) deklariert sein, bevor sie benutzt werden können.

Funktionsdeklaration:

```
int square(int x);
```

Funktionsdefinition:

```
int square(int x) {
    return x*x;
}
```

Funktionen der Standard-Bibliothek werden in **Definitionsdateien** (**header file**) wie `stdio.h` und `math.h` deklariert.

Definitionsdateien werden mittels `#include`-Anweisung am Anfang einer Quelldatei bereitgestellt.

Die Angabe der Parameternamen im Prototyp ist optional.

Funktionen (5)

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

int max(int, int);                  /* Prototyp */

main() {
    int n;
    n = printf("max(%d,%d) = %d\n", 5, 7, max(5, 7));
    printf("#Zeichen in vorheriger Ausgabe: %d\n", n);
    printf("sin(%f) = %f\n", M_PI / 4, sin(M_PI / 4));
}

int max(int a, int b) {             /* Definition */
    return (a > b) ? a : b;
}
```

Funktionen (6)

Rufen sich zwei Funktionen gegenseitig auf, **müssen** zunächst **Funktionsprototypen** definiert werden:

```
int a(int y);    /* Prototyp */
int b(int z);    /* Prototyp */

int a(int x) {
    return b(x - 1) * b(x - 2);
}

int b(int x) {
    return (x <= 0) ? x * 2 : a(x - 10);
}
```

Parameternamen in Prototyp und Funktionskopf müssen **nicht** übereinstimmen.

Funktionen (7)

Compiler-Fehler, wenn Prototyp und Funktionskopf unterschiedliche Rückgabewerte oder Parameter haben.

Beispiel:

```
int sqr(int x);
.....
float sqr(float x) {
    return x*x;
}
```

Beispiel:

```
int fkt(int x, int y);
.....
int fkt(int x) {
    return x*x;
}
```

Aufruf einer Funktion, die vorher **nicht deklariert** wurde:

- Als Rückgabetyp wird `int` angenommen.
- Es werden keine Annahmen über Parameter getroffen.
- Compiler-Verhalten abhängig von der Implementierung.

Funktionen: `main`

In C ist das Hauptprogramm eine Funktion, für die der Name `main` vorgeschrieben ist.

Für jedes ausführbare Programm muss die Funktion `main` existieren, die den Einstiegspunkt bezeichnet.

Damit der Einstiegspunkt eindeutig ist, darf maximal eine Funktion `main` im gesamten Programm existieren.

Im einfachsten Fall definiert man `main` als:

```
main() {
    ....
}
```


Funktionen: `main` (2)

Da für `main` kein Prototyp angegeben wurde, wird automatisch als Rückgabe der Typ `int` angenommen.

Übersetzen des letzten Beispiels liefert Warnungen:

```
warning: return-type defaults to ‘int’
warning: control reaches end of non-void function
```

Soll kein Wert zurückgegeben werden, ist explizit der Rückgabetyp `void` zu definieren.

Struktogramm: analog zu Funktionen und Prozeduren

Funktionen: `main` (3)

Auch `main` kann Parameter von außen übernehmen und Werte zurückgeben.

Definition nach ANSI C:

```
int main(int argc, char *argv[]) {
    ....
    return <ausdruck>;
}
```

Zeichenfolgen als Parameter übergeben:

- `argc` (argument count): Anzahl der beim Aufruf angegebenen Zeichenfolgen (inkl. Programmname).
- `argv` (argument vector): Vektor von Zeigern auf die angegebenen Zeichenfolgen. `argv[0]` = Programmname

Funktionen: `main` (4)

```
#include <stdio.h>

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

Wird das Programm als `arg.c` gespeichert und mit `arg eins 2 drei` ausgeführt, so wird folgende Ausgabe erzeugt:

```
Parameter 0 = arg
Parameter 1 = eins
Parameter 2 = 2
Parameter 3 = drei
```


Funktionen: `main` (5)

Üblich: String-Array zur Abfrage von Umgebungsvariablen.

```
#include <stdio.h>
void main(int argc, char *argv[], char *env[]) {
    int i;
    for (i = 0; env[i] != NULL; i++)
        printf("Variable %d = %s\n", i, env[i]);
}
```

Erzeugt beim Aufruf unter Linux bspw. die Ausgabe:

```
Variable 0 = PWD=/home/jochen
Variable 1 = VENDOR=suse
Variable 2 = PAGER=/usr/bin/less
...
```


Funktionen: `main` (6)

Umwandeln der Programmparameter:
`int sscanf(char *s, ...)`: äquivalent zu `scanf`, aber Eingabezeichen stammen aus Zeichenkette `s`. Rückgabewert: Anzahl der umgewandelten Eingaben.

Beispiel:

```
int i, n, p;
for (i = 0; i < argc; errno = 0, i++) {
   p = sscanf(argv[i], "%d", &n);
   if (p == 0)
      printf("failed to convert %s\n", argv[i]);
   else if (errno == ERANGE)
      printf("value %s out of range\n", argv[i]);
   else printf("%d: value %d\n", i, n);
}
```

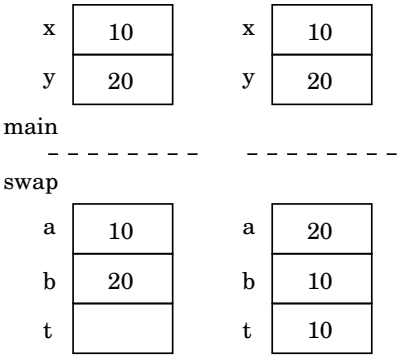
Funktionen: Parameterübergabe

Alle Parameter werden als Wert übergeben (**call by value**).

Innerhalb der Funktionen werden private Kopien der übergebenen Parameter angelegt, die während der Ausführung benutzt werden.

Beispiel:

```
void swap(int a, int b) {
   int t = a;
   a = b;
   b = t;
}
.....
swap(x, y);
```



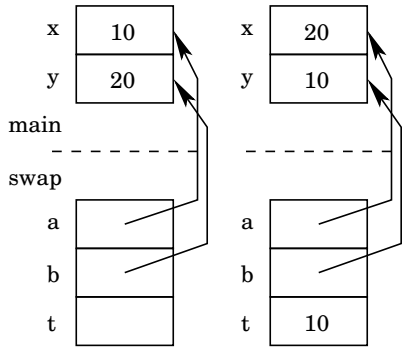
Funktionen: Parameterübergabe (2)

`swap` kann die Werte beim Aufrufer nicht beeinflussen, da nur Werte (Kopien) übergeben werden.

Lösung: Der Aufrufer muss Zeiger auf die Werte, die geändert werden sollen, übergeben (**call by reference**).

Beispiel:

```
void swap(int *a, int *b) {
   int t = *a;
   *a = *b;
   *b = t;
}
.....
swap(&x, &y);
```



Funktionen: Parameterübergabe (3)

Vektoren (Arrays) als Parameter in Funktionen:

- Nur die Adresse des ersten Elements wird übergeben.
- Die Elemente des Vektors werden **nicht** kopiert.
- Es erfolgt **keine** Bereichsüberprüfung.

Beispiel:

```
void main() {
  int i, u[10];
  init(u, 10);
  for (i = 0; i < n; i++)
    printf("%d\n", u[i]);
}
```

```
void init(int *a, int n) {
  for (n--; n >= 0; n--)
    a[n] = n+1;
}
```


Funktionen: Parameterübergabe (4)

Für Zeichenfolgen gilt dasselbe wie für Vektoren:

```
void substitute(char *a, char x, char y) {
   for (; *a != '\0'; a++)
      if (*a == x)
         *a = y;
}
void main() {
   char *s;
   s = (char *) malloc(sizeof("Hallo, Welt!"));
   strcpy(s, "Hallo, Welt!");

   substitute(s, 'l', 'n');
   printf("%s\n", s);
}
```

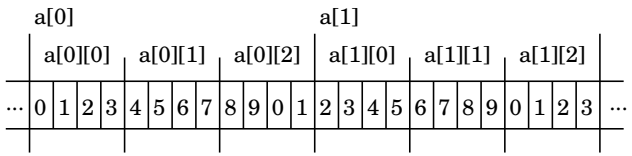
Funktionen: Parameterübergabe (5)

Bei 1D-Vektoren kann die Elementanzahl entfallen.

Bei mehrdimensionalen Vektoren darf nur die Elementanzahl der ersten Dimension weggelassen werden, da sonst der Index-Operator `[]` die Adresse nicht berechnen kann.

Beispiel:

```
type a[2][3][4];
...
b = a[x][y][z];
```



Element `a[x][y][z]` steht an Adresse `a + x*3*4 +y*4 + z`. Für diese Berechnung ist die Elementanzahl der ersten Dimension nicht wichtig.

Funktionen: Parameterübergabe (6)

Zugriff auf 2D-Vektoren mittels Index-Operator:

```
void main() {
    int a[2][4];
    init(a, 2, 4);
    ausgabe(a, 2, 4);
}
void init(int b[][4], int lx, int ly) {
    int i, j, n;
    for (i = 0, n = 0; i < lx; i++)
        for (j = 0; j < ly; j++, n++)
            b[i][j] = n;
}
void ausgabe(int b[][4], int lx, int ly) ....
```


Funktionen: Parameterübergabe (7)

Zugriff auf 2D-Vektoren mittels Zeiger-Arithmetik:

```
void main() {
    int a[2][4];
    init((int *) a, 2, 4);
    ausgabe((int *) a, 2, 4);
}
void init(int *b, int lx, int ly) {
    int i, j, n;
    for (i = 0, n = 0; i < lx; i++)
        for (j = 0; j < ly; j++, n++)
            *(b+i*ly+j) = n;
}
void ausgabe(int *b, int lx, int ly) .....
```


Speicherverwaltung

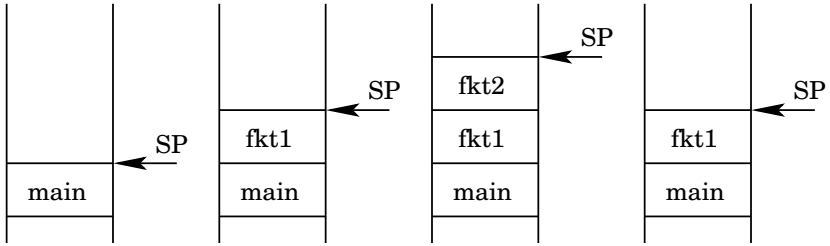
Der von einem C-Programm während seiner Ausführung belegte Speicher ist in vier Bereiche aufgeteilt:

- **Code-Segment**: enthält das übersetzte Programm, also die auszuführenden Maschinenbefehle.
- **Statische Daten**: während des gesamten Lebenszyklus verfügbare Daten wie globale Variablen oder statische, lokale Variablen.
- **Heap:** dynamisch allozierter Speicher (`malloc`)
- **Stack:** Informationen über Funktionsaufrufe

Speicherverwaltung (2)

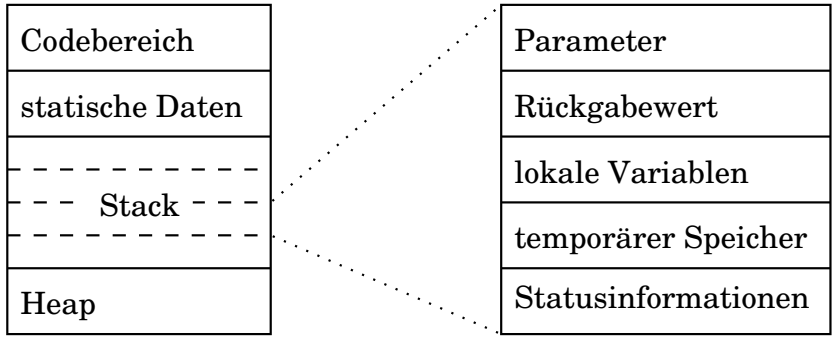
Für die lokalen Variablen und Argumente einer Funktion wird erst beim Aufruf der Funktion Speicherplatz reserviert.

Nach Beenden der Funktion wird der Speicherplatz wieder freigegeben.



Ein CPU-Register (SP: Stack Pointer) enthält die Adresse des nächsten freien Speicherplatzes.

Speicherverwaltung (3)



Alle Informationen, die zum Ausführen einer Funktion notwendig sind, werden in einem **Stack-Frame** abgelegt:

- **temporärer Speicher:** Evaluierung von Ausdrücken
- **Statusinformationen:** Programmzähler, ...

Speicherverwaltung (4)

Vorsicht: Zeiger auf lokale Variablen als Rückgabewert einer Funktion liefern **keinen** definierten Wert!

```
char* intToString(int a) {
    char s[10];
    int i, t;
    for (t = 1; t < a; t *= 10)
        ;
    for (t /= 10, i = 0; t > 0; t/= 10, i++)
        s[i] = (a / t) % 10 + '0';
    s[i] = '\0';
    return s;
}
```


Rekursive Funktionen

Rekursion in der Mathematik:

- **Binomialkoeffizienten:**
$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \text{ mit } \binom{n}{n} = 1 \text{ und } \binom{n}{0} = 1$$
- **Fibonacci-Zahlen:**
$$F_n = F_{n-1} + F_{n-2} \text{ mit } F_0 = 0 \text{ und } F_1 = 1$$
- **Determinante:** Die Adjunkte A_{ik} ist die Determinante $(n-1)$ter Ordnung, die durch Streichen der i-ten Zeile und k-ten Spalte, multipliziert mit $(-1)^{i+k}$ entsteht.

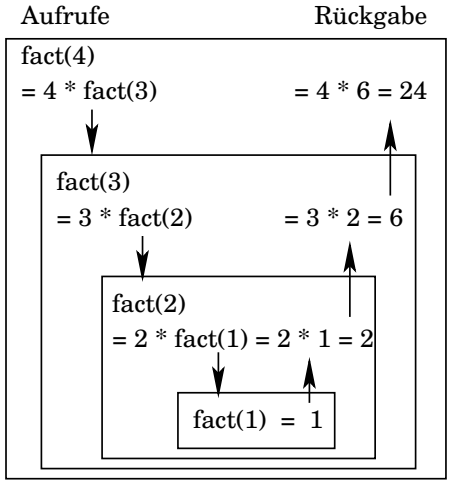
  Entwickeln nach der i-ten Zeile und k-ten Spalte:
$$D(a_{ij}) = \sum_{k=1}^{n} a_{ik} A_{ik}$$

Rekursive Funktionen: Fakultäten

C-Code:

```
#include <stdio.h>
int fact(int n) {
   if (n <= 1)
      return 1;
   return n * fact(n-1);
}
void main() {
   int n;
   for (n = 0; n < 10; n++)
      printf("%d! = %d\n",
             n, fact(n));
}
```

Programmablauf:

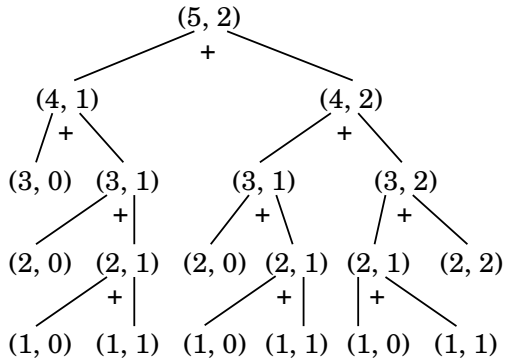


Rekursive Funktionen: Binomialkoeffizienten

C-Code:

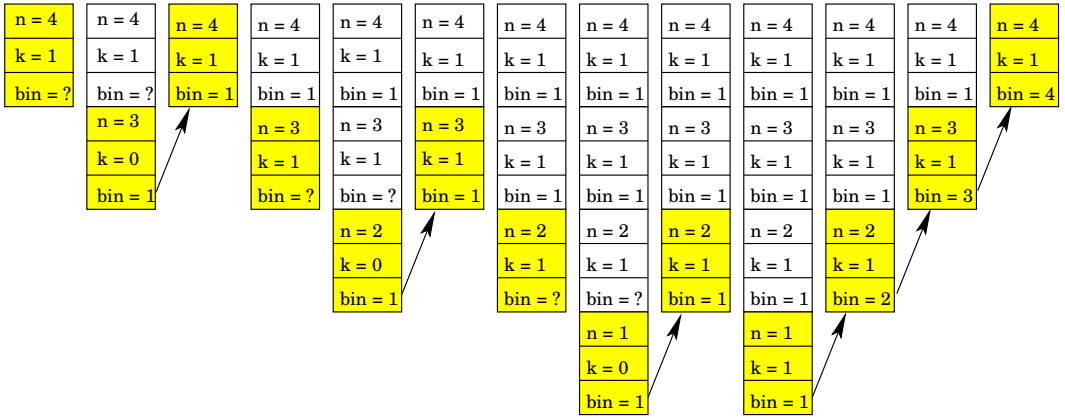
```
#include <stdio.h>
int bin(int n, int k) {
   if (n < k || k < 0)
      return -1;
   if (k == 0 || k == n)
      return 1;
   return bin(n-1, k-1)
      + bin(n-1, k);
}
void main() {
   printf("bin(%d, %d) = %d\n", 7, 3, bin(7, 3));
}
```

Programmablauf:



Rekursive Funktionen: Speicherverwaltung

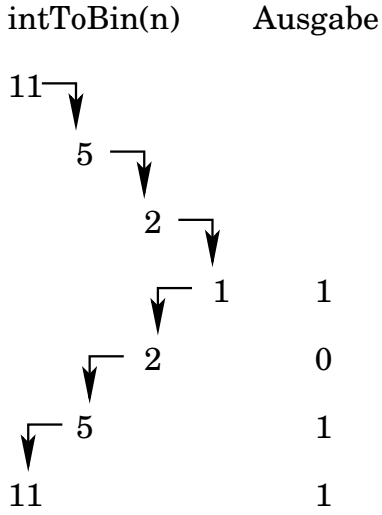
am Beispiel der Binomialkoeffizienten:



Rekursive Funktionen: Binärdarstellung

```
#include <stdio.h>
void intToBin(int n) {
    if (n < 2) {
        printf("%1d", n);
        return;
    }
    intToBin(n / 2);
    printf("%1d", n % 2);
}

void main(void) {
    intToBin(11);
}
```

Ablauf:



Gültigkeitsbereich von Bezeichnern

Gültigkeitsbereich (scope): Der Teil des Programms, wo ein Bezeichner benutzt werden kann.

Die Bedeutung einer Deklaration ist abhängig von deren Stelle im C-Code. Mögliche Stellen für Deklarationen:

- Außerhalb von Funktionen: **externe/globale Variablen**
- Im Funktionskopf: **formale Parameter einer Funktion**
- Innerhalb eines Blocks: **lokale Variablen**
- Funktionen können nur außerhalb von Funktionen deklariert werden (parallele Blöcke).

Gültigkeitsbereich von Bezeichnern (2)

Module: C-Quellcode auf mehrere Dateien aufteilen:

- Datei enthält Vereinbarungen und/oder Funktionen.
- Nur eine Datei darf die Funktion `main` enthalten.

Beispiel: Stack

- `stack.h`: enthält die Datenstruktur und die Deklaration aller Zugriffsmethoden
- `stack.c`: Implementierung aller Zugriffsmethoden
- `rechner.c`: Rechner für Postfix-Eingabe (wird auch Umgekehrte Polnische Notation genannt), enthält `main`

Gültigkeitsbereich von Bezeichnern (3)

Es gibt vier mögliche Gültigkeitsbereiche:

- Der Compiler kennt nur, was er bereits gelesen hat, aber keine anderen Dateien. → Externe Variablen und alle Funktionen gelten vom Deklarationspunkt bis zum Ende der Datei.
- Formale Funktionsparameter gelten vom Deklarationspunkt bis zum Ende der Funktion bzw. des Prototypen.
- Lokale Variablen am Beginn eines Blocks gelten bis zum Ende des Blocks.
- Anweisungsmarken gelten innerhalb der Funktion, in der die Deklaration erfolgt.

Gültigkeitsbereich von Bezeichnern (4)

```
/* global: gültig bis zum Ende der Datei */
int global;

/* x: nur innerhalb der Klammern gültig */
double sin(double x);

int main(int argc, char *argv[]) {
    /* lokal: nur innerhalb des Blocks gültig */
    int lokal;
    ...
    /* Label ende: innerhalb der Funktion main gültig */
    ende: ...
}
```

Mehrfachnutzung von Bezeichnern

Derselbe Bezeichner kann gleichzeitig für unterschiedliche Objekte benutzt werden.

Namensklassen: Ein Bezeichner kann auftreten

- als Marke,
- als Name eines Strukturtypen,
- als Komponente innerhalb einer Struktur
- oder als anderer Bezeichner.

Verschiedene Namensklassen können sich überschneiden.

Mehrfachnutzung von Bezeichnern (2)

Beispiel: Namensklassen, die sich überschneiden.

```
int main(int argc, char *argv[]) {
    struct x {
        int x, y;
    } y;
    int x = 5;
    if (x >= 5)
        goto x;
    ...
x:  ...
}
```

Sehr schlechter Programmierstil!

Sichtbarkeit von Bezeichnern

Mehrfachnutzung desselben Bezeichners in einer Namensklasse möglich.

Voraussetzung: Bezeichner tritt in verschiedenen Gültigkeitsbereichen auf.

Beispiel:

```
int x, f;              /* globale Variable */

int main(int cnt, char *args[]) {
    double f;          /* lokale Variable */
    f = sin(1.0);      /* Zugriff auf lokale Var */
    scanf("%d", &x);   /* Zugriff auf globale Var */
    ...
}
```


Sichtbarkeit von Bezeichnern (2)

Die Variablen eines Blocks sind in weiter innen liegenden Blöcken verfügbar, falls keine Variablen mit demselben Bezeichner definiert werden.

Es gelten folgende Sichtbarkeitsregeln:

- Funktionsparameter überschreiben globale Variablen.
- Deklarationen am Anfang eines Blocks überschreiben Deklarationen außerhalb des Blocks.

Lebensdauer von Objekten

Wie lange bleibt ein Objekt im Speicher und kann über seinen Bezeichner angesprochen werden?

Es sind drei Fälle zu unterscheiden:

- **statische Lebensdauer:** Objekte sind stets verfügbar (Funktionen, globale Variablen, statische Variablen).
- **automatische Lebensdauer:** Objekte, die zu Beginn eines Blocks/einer Funktion angelegt werden, sind nach dem Verlassen des Blocks/der Funktion nicht mehr verfügbar, sofern nicht anders definiert.
- **dynamische Lebensdauer:** Der Speicherbereich wird mittels Bibliotheksfunktionen angelegt und freigegeben (`malloc`, `free`).

Lebensdauer von Objekten: Beispiel

```
void fkt(int x) {
    static int s = 1;   /* statische Lebensdauer */
    auto int a = 1;     /* automatische Lebensdauer */
    printf("x = %2d, s = %2d, a = %2d\n", x, s++, a++);
}
void main(void) {
    int i, *p;
    p = (int *) malloc(10 * sizeof(int));
    for (i = 0; i < 10; i++) {
        p[i] = i + 1;   /* dynamische Lebensdauer */
        fkt(p[i]);
    }
    free(p);
}
```


Speicherklasse

Die Speicherklasse eines Objekts hat Einfluss auf

- die Lebensdauer und
- den Gültigkeitsbereich des Objekts und
- erlaubt Einschränkungen für den Zugriff auf Daten.

In C gibt es vier Speicherklassen:

- `auto`
- `register`
- `static`
- `extern`

Speicherklasse: `auto`

Ist nur am Anfang von Blöcken/Funktionen erlaubt.

Definierter Standard:

- alle innerhalb eines Blocks/einer Funktion deklarierten Variablen sind implizit von der Speicherklasse `auto`
- wird in Deklarationen oft weggelassen

Lebensdauer: wird bei jedem Blockeintritt erzeugt und initialisiert, wird nach Verlassen des Blocks wieder gelöscht

Gültigkeit: bis zum Ende des umschließenden Blocks `{ }`

Speicherklasse: `register`

Hinweis an den Compiler, dass auf dieses Objekt oft zugegriffen wird. Soll daher in einem schnellen CPU-Register gespeichert werden. Anzahl prinzipiell unbegrenzt.

Hinweis: Der Compiler ist in keiner Weise daran gebunden.

Verwendung: lokale Variablen und Parameterdeklarationen.

Beispiel:

```
{
    register int i;
    for (i = 0; i < MAX; i++)
        ...
} /* hier wird das Register wieder freigegeben */
```


Speicherklasse: `register` (2)

Lebensdauer: wird bei jedem Blockeintritt erzeugt und nach Verlassen des Blocks wieder gelöscht.

Gültigkeit: bis zum Ende des umschließenden Blocks `{ }`.

Anmerkungen:

- Der Adressoperator `&` ist nicht anwendbar.
- Das Attribut hat keine Bedeutung mehr, da moderne Compiler sehr gute Optimierungsstufen haben.
- In der Praxis: Einschränkungen bzgl. Anzahl und Datentypen, maschinenabhängig.

Speicherklasse: `extern`

Variablen, die außerhalb von Funktionen deklariert werden, sind **implizit** vom Typ `extern`.

Lebensdauer: Für die gesamte Laufzeit des Programms wird Speicherplatz reserviert.

Gültigkeit: Globale Variablen gelten von dem Punkt, an dem sie vereinbart werden, bis zum Ende der Datei.

Ein Objekt einer externen Deklaration ist beim Binden auch Programmteilen aus anderen Dateien bekannt.

Speicherklasse: `extern` (2)

Externe Variable einer Datei in Funktion einer anderen Datei nutzen: Verweis auf Variable angeben.

Beispiel: **schlechter Programmierstil!**

Datei `file1.c`:

```
#include <stdio.h>
int a = 1, b = 3;
int fkt(void);
void main(void) {
    int c = fkt();
    printf("%d, %d, %d\n",
           a, b, c);
}
```

Datei `file2.c`:

```
int fkt(void) {
    extern int a;
    int b = 2;
    return a+b;
}
```

Speicherklasse: `extern` (3)

In der Regel ist Datenaustausch über eine Parameterliste gegenüber globalen Variablen vorzuziehen:

- Global gültige Daten führen oft zu Programmen mit vielen, schwer durchschaubaren Datenpfaden zwischen Funktionen (mit unerwünschten Nebenwirkungen).
- Der Aufbau von Bibliotheken mit allgemein gültigen Funktionen ist nicht möglich.

Weniger Fehleranfällig: Prinzip der Datenkapselung.

- Daten sind nur innerhalb eines Moduls sichtbar.
- Zugriff auf Daten nur über definierte Zugriffsmethoden.
- Beispiel Stack: `push(elem)`, `pop()`, `top()`, `isEmpty()`, ...

Speicherklasse: `static`

Anwendung bei

- **Funktionen:** Die Funktion ist nur innerhalb der Quelldatei und **nicht** dem Linker bekannt (siehe Modulare Programmierung, Datenkapselung).
- **lokale Variablen:** Objekte behalten ihren Wert auch nach Verlassen des Blocks, in dem sie definiert sind.
- **globale Variablen:** Einschränkung des Gültigkeitsbereichs auf die Datei.

Gültigkeit: innerhalb des umschließenden Blocks bzw. bei Funktionen und globalen Variablen innerhalb der Datei.

Speicherklasse: `static` (Beispiel)

Datei `dat1.c`:

```
void f(void) {
    extern int i;
    ...
}

static int i;
int j;
void g(void) {
    i = 42;
    j = 4711;
    ...
}
```

Datei `dat2.c`:

```
void h(void) {
    extern int j;  /* ok */
    extern int i;  /* Fehler! */

    i += 42;
    j += 4711
}
```


Speicherklasse: Beispiel

```
extern int ext, g();      /* global, extern definiert */

static int f(register i) {/* dieser Datei bekannt */
    auto int a;           /* lokale Variable */
    static int x;         /* Wert bleibt erhalten */
    extern int e();       /* lokal, extern definiert */

    for (a = 0; a < 2; a++, i++) x += i * i;
    return e(x);
}
void main(void) {
    register r = ext;     /* lokale Register-Variable */
    printf("f(%d) = %d\n", r, f(r));
    printf("g(%d) = %d\n", r, g(r));
}
```

Pointer Fun

www.cs.stanford.edu/cslibrary/PointerFunCppBig.avi

Testen

Aufgabe: Einlesen einer Datei und umbenennen aller darin enthalten Dateinamen durch Anhängen einer Extension. Der Dateiname sowie die Extension werden als Kommandozeilenparameter an das Programm übergeben.

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

#define LEN 65

int main(int argc, char *argv[]) {
    FILE *file;
    char *filename, *extension, *listfile;
```


Testen (2)

```
    listfile = argv[1];
    extension = argv[2];
    file = fopen(listfile, "r");
    while (fgets(filename, LEN, file) != NULL) {
        char *newFilename;

        strncpy(newFilename, filename, LEN);
        strncat(newFilename, ".", LEN);
        strncat(newFilename, extension, LEN);
        rename(filename, newFilename);
    }

    return 0;
}
```


Testen (3)

Speicherzugriffsfehler! → Fehlerbehandlung einbauen

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

#define LEN 65

int main(int argc, char *argv[]) {
    FILE *file;
    char *filename, *extension, *listfile;

    if (argc != 3) {
        printf("try: %s listfile extension\n", argv[0]);
        return 1;
    }
```

Testen (4)

```
    listfile = argv[1];
    extension = argv[2];
    file = fopen(listfile, "r");
    if (file == NULL) {                          // !!!
        perror(listfile);                        // !!!
        return 1;                                // !!!
    } else printf("file %s opened\n", listfile); // !!!

    while (fgets(filename, LEN, file) != NULL) {
        ...
        if (rename(filename, newFilename) < 0)   // !!!
            perror(filename);                    // !!!
    }
    return 0;
}
```


Testen (5)

Speicherzugriffsfehler → Debug-Ausgaben einbauen

```
    while (fgets(filename, LEN, file) != NULL) {
        char *newFilename;

    #ifdef DEBUG
        printf("filename = %s", filename);
    #endif
        strncpy(newFilename, filename, LEN);
        strncat(newFilename, ".", LEN);
        strncat(newFilename, extension, LEN);
    #ifdef DEBUG
        printf("new filename = %s", newFilename);
    #endif
        ...
```


Testen (6)

Speicherplatz für `filename` und `newFilename` bereitstellen!

```
    char filename[LEN];
    char newFilename[LEN];
```

keine Datei wurde umbenannt
→ Zeilentrenner `\n` aus Dateinamen entfernen

```
void chomp(char *str) {
    int i = 0;

    while (str[i] != '\n')
        i += 1;
    str[i] = '\0';
}
```


Testen (7)

```
int main(int argc, char *argv[]) {
    ...
    while (fgets(filename, LEN, file) != NULL) {
        char *newFilename;

        chomp(filename);
    #ifdef DEBUG
        printf("filename = %s", filename);
    #endif
        ...
```

einige Dateien wurden nicht umbenannt
→ führende Leerzeichen entfernen

Testen (8)

```
void removeLeadingBlanks(char *str) {
    int i, p = 0;
    int len = strlen(str);

    while (isblank(str[p]))
        p += 1;

    for (i = 0; p < len; i++, p++)
        str[i] = str[p];
    str[i] = '\0';
}
```


Testen (9)

```
int main(int argc, char *argv[]) {
    ...
    while (fgets(filename, LEN, file) != NULL) {
        char *newFilename;

        chomp(filename);
        removeLeadingBlanks(filename);
    #ifdef DEBUG
        printf("filename = %s", filename);
    #endif
        ...
```