

char..., short..., int..., long..., float, double

```
struct
{
    int alter;
    double einkommen;
    ...
} v1, v2;

v1.einkommen = 2000.0;
v1.alter      = 99;
v2 = v1;
```

```
struct pers
{
    int alter;
    char geschlecht;
    double einkommen;
} v2;
```

C, C++

```
struct pers v1;
v1.alter = 99;
...
int pers = 12;
v2 = v1;
```

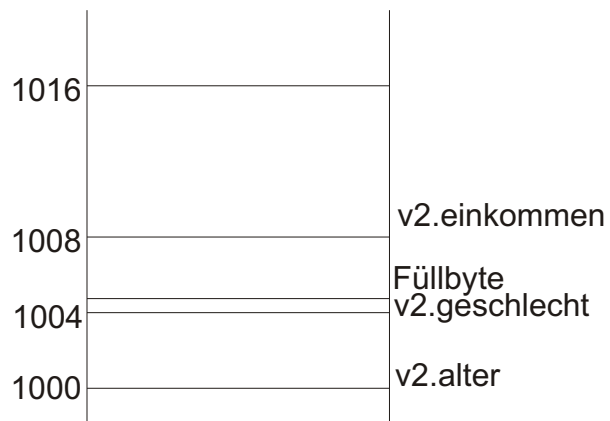
C++

```
pers v1;
v1.alter = 99;
...
int pers = 12;
v2 = v1;
```

```
struct firma
{
    struct pers chef;
    struct firma f;
};

struct
{
    unsigned char alter;
    double einkommen;
    char geschlecht;
}

struct
{
    double einkommen;
    unsigned int alter:7;
    unsigned int geschlecht:1;
}
```



Der Vorteil an dieser Variante: alter und geschlecht teilen sich gemeinsam einen 8-Bit-Bereich.

```
union u
{
    int i;
    float f;
};

union u v1;
v1.i
v1.f
```

```

struct pers
{
    unsigned int alter:7;
    unsigned int geschlecht:1;

    union
    {
        int bh;
        double einkommen;
    } bh3;} v1;    //v1 ist männl. mit Einkommen oder weibl. mit bh.
if (v1.geschlecht) {v1.bhe.bh=2;}
else {v1.bhe.einkommen=2000.0;}

```



```

struct
{
    unsigned int s:1;
    unsigned int m:20;
    unsigned int exp:11;
};

union
{
    float f;
    struct fl sfl;
} flvariable;
flvariable.f = 3.14;
... flvariable.sfl.exp ...

```

Adressen, Zeiger, "Pointer"

```
int main()
{
    int i;
    i=12;
}
```

Um die Adresse einer Variablen zu erhalten, stellt man der Variable das "&" voran (z.B. &i). Sie wird festgehalten in einer Variablen, die vorher mit dem gleichen Datentyp und folgendem * definiert wurde. Andernfalls muss gecastet werden

Beispiele:

```
int* adresse;           int adresse;
int i;                  int i;
i=12;                   i=12;
adresse=&i;              adresse=(int)&i;
```

Um von der Adresse auf den Wert zugreifen zu können, wird das * benötigt:

```
int* pi = &i;
*pi = 25; // i erhält den Wert 25
```

Man kann auch einen Zeiger auf einen Zeiger setzen:

```
int** ppi = &pi;
```

**ppi ist gleichzusetzen mit i, enthält aber die Adresse von *pi.

Felder, Arrays

```
int main()
{
    int arr[10];
    int *parr=arr; // Konstanter Zeiger
    int i;
    for (i=0; i<10; i++)
    {
        arr[i]=i+i;
    }
    arr[0]=12; // oder *arr=12;
    arr[4]=19; // oder *(arr+4)=19; oder 4[arr]=19;
}
```

"Zeigerarithmetik"

```
(&arr[4]) - (&arr[1]) -> 3
(&arr[1]) - (&arr[4]) -> -3
```

```
parr=&parr[1];
```

```
parr=&(* (parr+1)); // * und & heben sich gegenseitig auf
parr=parr+1;       // oder parr++;
parr+=5;           // Fünfmal den Wert für ein int addieren
```

```

int main()
{
    int arr[10];
    int *parr=arr; // Konstanter Zeiger
    int i;

    for (i=0; i<10; i++)
    {
        arr[i]=i+i;
    }

    for (parr=&arr[0] arr; parr < arr+10; parr++)
    {
        *parr=0;
    }

    int *p;
    int i;
*p = 25;
    p = &i;
    *p = 25;

    struct student
    {
        int matr;
    }

    void m1 (int *i)
    {
        (*i) += 25;
    }

    void m2 (struct student *s)
    {
        (*s).matr = 25;
    }

    int i=0;
    struct student s;
    s.matr=0;
    m1(&i);
    m2(&s);

```

Zeiger:

- CBR (call by reference, siehe oben)
- Felder, schneller Zugriff auf ..[]
- Strings, Zeichenketten -> nullterminierte Strings (zero terminated strings), char[...]

(*zeiger).element ist identisch mit zeiger->element

```
char s1[20];
s1[0]='K';
s1[1]='a';
s1[2]='i';
s1[3]="0";
printf(s1);
```

Abgekürzte Version

```
s1[20]={'K','a','i',0}
```

Ausgabe: Kai

```
s1[3]='_';
s1[4]='X';
s1[5]='.';
s1[6]="0";
printf(s1);
```

Ausgabe: Kai_X.

```
printf(s1+4);
```

Ausgabe: X.

```
s1[3]="0";
printf(s1);
```

Ausgabe: Kai

```
#include <string.h>
char s1[20]="Kai";
printf(s1);
strcpy(s1, "Sina");
strcat(s1, " ?");
printf(s1);
```

Ausgabe: Sina ?

```
int i=13;
char s1[6];
char s2[5];
int j=25;
```

```
strcpy( s1, "Kai");
strcpy( s2, "Sina");
```

```
strcpy( s1, "Kai ist doof");
strcpy( s1, "Kai ...", 20);
s1[19]=0;
```

klassischer Fehler, Speicher wird "über"-geschrieben
ist besser

```
strncat(..., ..., 16);
strlen(s1);
```

Länge des Strings ausgeben

```

char s1[20];
char *p;
p=strcpy(s1, "Kai");
strcat(strcat(strcpy(s1, "Kai"), " ist"), " doof");

```

```

memcpy(s1, "Kai", 4);
memcpy(s1, "Kai", 40);

```

```

strcmp(s1, "Kai");
memcmp(s1, s2, n);

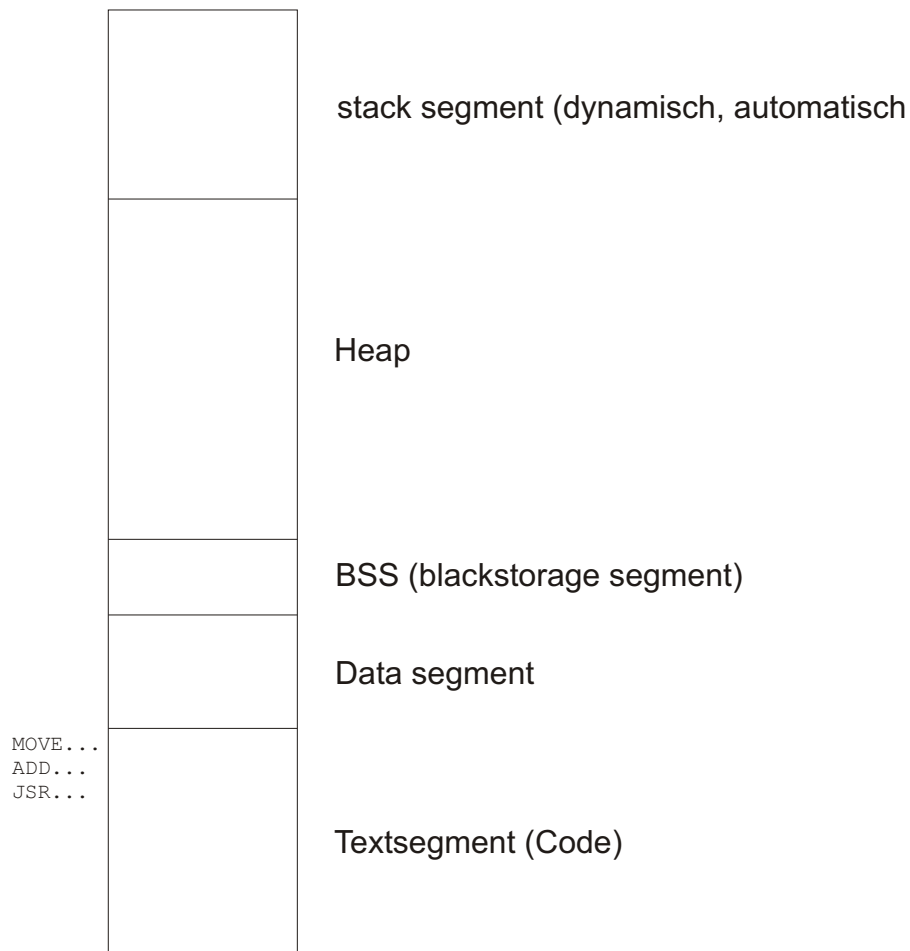
```

```

strcmp("Lai", "Kai");           -1   memcmp(s1, s2, n);
strcmp("Kai", "Lai");           +1
strcmp("Kai", "Kao");           +6
strcmp("Kai", "KaiA");          65
strcmp("Ab", "Ab");              0

```

memset(s1, 12, 20); auf Wert 12 werden 20 Elemente gesetzt



```

s1=malloc(Anzahl);
free(s1);
(int *) malloc(Anzahl * sizeof(int));

void * malloc(...);

```

```

if ((ip=(int *) malloc(...)) ==NULL)
{
    // Fehler...
}
else
{
    ip[0] ...
    free(ip);
}

int *p_arr=NULL;
int anzahl= ...
p_arr=(int *) malloc(anzahl*sizeof(int));
if (p_arr)
{
    p_arr[0]=...;
    p_arr[anzahl-1]=...;

    double *p_d;
    p_d=(double *) malloc(anzahl*sizeof(double));
    free(p_arr);
    p_arr=NULL;
}
else
{
    ...
}

```

Initialisierung eines Zeigerbereichs mit vorgegebenen Nullwerten:

```
memset(p_arr,0,anzahl*sizeof(int));
```

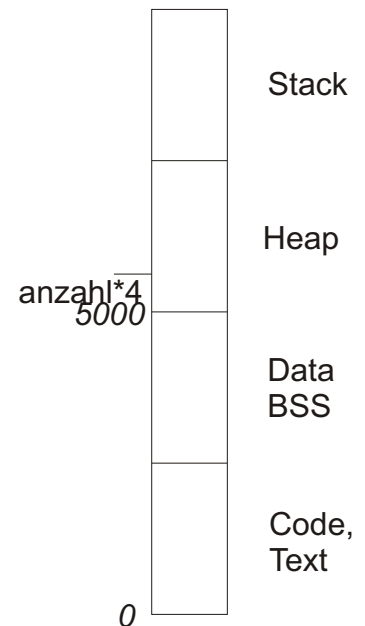
oder (statt des Befehls malloc):

```
calloc(anzahl, sizeof(int));
```

```
p=(...) malloc(100);
```

·
·
·

```
p=(int*) realloc(p,200); // zum erneuten Festlegen des reservierten Speicherbereichs
```



Weil bei einem misslungenen realloc (bei zu wenig Speicherplatz) der Speicherbereich verloren geht, sollte man realloc immer erst über eine weitere Variable aufrufen:

```
ptmp = (...) realloc(p2, 5000);
if (ptmp)
{
    p2=ptmp;
}

```

oder abgekürzt:

```
if (ptmp=(...) realloc(p2,500)) { p2=ptmp;}
```

```

p=(...) malloc(100);

realloc(p, 0); ist identisch mit free(p);
p=(...) realloc(NULL, 100); ist identisch mit ...=malloc(100);

int l=0;

char *p=NULL;
while(...)
{
    l=...;
    ptmp=(...) realloc(p, l);
    ...
}

```

Arrays mit verschobenen Indizes

```

int ARR_tmp[16];
int *arr = &ARR_tmp[5];
arr[-5] = ...;
arr[10] = ...;

```

```

int ARR_tmp[3];
int *arr=&ARR_tmp[-3];
arr[3] = ...;
arr[5] = ...;

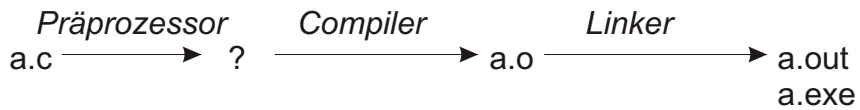
```

andere Art:

```

int *ARR_tmp = (...) malloc(3*sizeof(int));
if (ARR_tmp) ...
{
    int *arr=&ARR_tmp[-3];
    ...
}

```



```
gcc -E a.c -o a.pp
```

<code>#include <stdio.h></code>	<i>Präprozessor</i>	<code>(Inhalt von stdio.h)</code>
<code>#define N 10</code>		<code>/</code>
<code>int arr[N]</code>		<code>int arr[10];</code>
<code>int main(...)</code>		<code>int main(...)</code>
<code>{</code>		<code>{</code>
<code>for (i=0; i<N; i++)</code>		<code>for (i=0; i<10; i++)</code>
<code>{...}</code>		<code>{...}</code>
<code>}</code>		<code>}</code>
 <code>#define QUADRAT(x) x*x</code>		
 <code>i = QUADRAT(5);</code>		<code>i = 5*5;</code>
<code>y = QUADRAT(i);</code>		<code>y = i*i;</code>