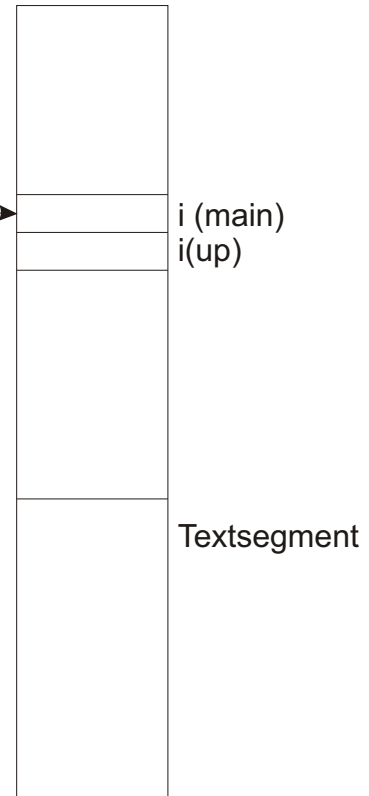


```

void up (int a, int b)
{
    int i;
    printf (...);
}

int main()
{
    int i; int
    up (12,5);
    ... &i int*
    ... &up void (*) (int, int)
}

```



```
#include <stdlib.h> // qsort, bsearch
```

```
// liefert <0, wenn *p1<p2, >0 wenn *p1>p2, 0 wenn gleich
```

```
int vergleich (const void * p1, const void * p2)
```

```
{
```

```
// Umständliche Art:
```

```
    if (* (int *) p1 < * (int *) p2) {return -1;}
    else if (* (int *) p1 > * (int *) p2) {return +1;}
    else {return 0;}

```

```
// Einfachere Art:
```

```
    return *(int *) p2 - * (int *) p1;
}

```

```
int arr[5] = {12, 1, -5, 10, 8};
```

```
    int (*) (void *a, void *b)
```

```
qsort(arr, 5, sizeof(int), &vergleich); das "&" kann bei "vergleich" auch  
weggelassen werden
```

```
int key=10;
```

```
p_gefunden = (int *) bsearch (&key , arr, 5, sizeof (int),  
&vergleich);
```

```
if (p_gefunden) {... *p_gefunden ...}
```

```
enum befehl
{
    print = 0;
    let = 1;
    call = 2;
    ...
}
```

Statt auf diese Art:

```
while (...)
{
    switch (akt_befehl)
    {
        case print: print();
                    break;
        case let:   ...
        case ...
        case ...
    }
}
```

kann man mit Zeigern auf Funktionen das Programm wie folgt beschleunigen:

```
void set() {...}
void print() {...}

void (* f[100]) () =
{
    &print;
    &let;
    ...
}

while (...)
{
    *(f[akt_befehl]) ();
}
```

```
void * bsearch (void * arr, size_t n; size_t s, ... vergl ...)
{
    size_t mid = n/2;
    if (vergl (arr, (void *) ((char *) arr)[mid * s]) < 0) ...
}

```

→ **vergleiche arr[0] mit arr[mid] ...**

C++

- **bool** true, false

```
bool b; b=true; // 1
if (b) ...
i > 10 int

```

- **//**-Kommentare

- **char** bleibt char

```
char c='A'+ 'B'+ 'c'+ 'd';
      ▼   ▼   ▼   ▼
C:      int  int  int  int
C++:    char char char char

```

- **(int)** ausdruck
zusätzlich int(ausdruck)

Default-Parameter

```
int sum (int a, int b, int c, int d)
{
    return a+b+c+d;
}

```

```
... = sum (12, 13, 14, 15); //4 Zahlen addieren
... = sum (10, 5, 0, 0);    //2 Zahlen addieren
... = sum (10, 5, 4, 0);    //3 Zahlen addieren

```

Mit der Erweiterung der Funktionszeile auf

```
int sum (int a, int b, int c=0, int d=0)

```

kann statt dessen auch geschrieben werden:

```
... = sum (10, 5);
... = sum (10, 5, 4);

```

Die Überladung von Funktionen ist nicht unter C möglich, dafür aber unter C++:

```
int sum (int a, int b) ←  
{  
    return a+b;  
}  
  
int sum (int a, int b, int c)  
{  
    return a+b+c;  
}  
  
double sum (double a, double b)  
{  
    return a+b;  
}  
  
double sum (int a, int b) {...} ————— kollidiert mit  
  
    = sum(12,13);
```

stddef.h:

```
#define NULL (void *) 0 // C  
const int  
#define NULL 0
```

```

class Stud
{
public:
    void init()
    {
        vn = NULL; nn = NULL;
    }

    void set (const char * vn, const char * nn, int m)
    {
        free();
        this->vn = (char *)malloc (strlen(vn)+1);
        if (! this->vn) {...}
        strcpy (this->vn, vn);
        this->m = m

        void free() {::free(vn); ::free(nn);}

private:
    char *vn;
    char *nn;
    int m;
};

...
Stud s;
s.init();
s.set("Karl", "Meier", 25);
s.print();
s.set("Otto", "Müller", 30);
s.print();

s.free();
}

```

In C++ gibt es einen Konstruktor, der automatisch aufgerufen wird. Dieser Konstruktor hat die gleiche Bezeichnung wie die Klasse, in der er steht, und keinen Rückgabewert (auch kein void). Die Methode `void init()` könnte man beispielsweise in `Stud()` umändern. Im Programmablauf kann man dann auf den einleitenden Aufruf von `init()` und den abschließenden Aufruf von `free()` verzichten.