

prog.c:

```
int main()
{
    int erg;
    up(12);
    erg=up2(10,5);
    return 0;
}
```

up.c:

```
void up(int i)
{
    ...
}

int up2(int a, int b)
{
    ...
    return 12;
}
```

Aufruf mittels:

```
gcc -Wall prog.c up.c -o prog && ./prog
cl prog.c up.c -o prog.exe && ./prog
```

C unter Linux
C++ unter win32

oder:

```
gcc -Wall -c prog.c
gcc -Wall -c up.c
gcc -Wall prog.o up.o -o prog && ./prog
```

C unter Linux

```
cl /c prog.c
cl /c up.c
cl prog.obj up.obj -o prog.exe && ./prog
```

Der Compiler wird beim Ausführen die Funktionen `up()` und `up2()` nicht erkennen. Daher muss man im Kopf der `prog.c` bekannt machen:

```
void up(int i);
int up2(int, int);
```

Sinnvoller ist es, eine Header-Datei anzulegen. Diese kann beispielsweise wie folgt aussehen:

up.h:

```
// Deklaration:
void up(int i);
int up2(int a, int b);
```

Die Dateien `prog.c` und `up.c` erhält dann eine `include`-Zeile (in `prog.c` stehen nicht mehr die bekanntmachenden obigen zwei Zeilen):

```
#include "up.h"
```

bibliothek.c:

```
#include "bibliothek.h"
int bibint()
{
    ...
}

int tuwas()
{
    ...
}
```

bibliothek.h:

```
#define BIBMAX 100
int bibinit();

int tuwas();
```

up.c:

```
#include "up.h"
#include "bibliothek.h"

void up(int i)
{
    bibinit();
}

int up2(int a, int b)
{
    return a+b;
}
```

up.h:

```
#include "bibliothek.h"
void up(int i);
int up2(int a, int b);
#define UPMAX ((BIBMAX)/2);
```

prog.c:

```
#include "up.h"
#include "bibliothek.h"
int main()
{
    int erg;
    up(UPMAX);
    erg = up2(10, 5);
    bibinit();
    return 0;
}
```

Bei dieser Lösung kommt es zu Kollisionen. Das liegt daran, dass nicht nur `up.c` ein Include auf `bibliothek.h` hat, sondern auch `prog.c`

bibliothek.h:

```
#ifndef _BIBLIOTHEK_H_
#define _BIBLIOTHEK_H_
#define BIBMAX 100
int bibinit();

int tuwas();
#endif
```

up.c:

```
#include "up.h"
#include "bibliothek.h"

void up(int i)
{
    bibinit();
}

int up2(int a, int b)
{
    return a+b;
}
```

In dieser Lösung kommt es zu keinen Kollisionen mehr.

up.h:

```
#ifndef _UP_H_
#define _UP_H_
#include "bibliothek.h"
void up(int i);
int up2(int a, int b);
#define UPMAX ((BIBMAX)/2);
#endif
```

prog.c:

```
#include "up.h"
#include "bibliothek.h"
int main()
{
    int erg;
    up(UPMAX);
    erg = up2(10, 5);
    bibinit();
    return 0;
}
```

bibliothek.h:

```
#ifndef _BIBLIOTHEK_H_
#define _BIBLIOTHEK_H_
#define BIBMAX 100
int bibinit();

int tuwas();
#endif
```

up.c:

```
#include "up.h"
```

```
int global=0;
```

definiert man mit `static`, ist `global` nicht mehr von aussen lesbar

```
int up2(int a, int b)
{
    global ++;
    return a+b;
}
```

prog.c:

```
#include "up.h"
```

```
extern int global = 5 ;
```

```
int main()
{
    int erg; global ++;
    up(UPMAX);
    erg = up2(10, 5);
    bibinit();
    return 0;
}
```

```

int up(int i)
{
    int erg = ...;
    if (i>0)
    {
        return up(i-1);
    }
    return 0;
}

int main()
{
    up(5);
}

```

in unserem Beispiel 0...5x
 die Variable `erg` wird 5x definiert, wenn sie nicht mit `static` definiert wird.

extern, static, auto, const

```

int i;
const int i2 = 12;

i = 12;
i2 = 25;

```

der Inhalt von `i2` kann nicht mehr verändert werden

```

void up (char * s)
{
    s[0]='a';
}

```

```

char puffer[100]="xyz";
up (puffer);
up ("xyz");

```

sorgt bei modernen Systemen für einen Absturz

Wenn man auch Konstanten wie `"xyz"` der Funktion `up()` übergeben will, muss man die Funktion abändern auf:

```

void up (const char * s)

```

```

const char * s = "xyz"
s++

```

funktioniert (`s` steht jetzt auf `"y"`), weil sich `const` auf den String `"xyz"` bezieht, nicht auf den Zeiger `s`

```
volatile int i;  
i = 10;  
...  
    pthread_create(f, &i ...);  
printf("%d\n", i);
```

```
f (void * p)  
{  
    (* (int *) p)++;  
}
```