

```

//inline                                     #define MIN(i, j) ((i)<(j)?(i):(j))

int min (int i, int j)
{
    return a<b? a : b;
}

...

int main()
{
    int a=10;
    int b=12;
    ... = min (++a, b);
}

```

~~min.cpp.h~~

```

inline int min (int a, int b)
{
    return a<b? a : b;
}

```

~~min.h~~

```

inline int min (int a, int b);

```

~~test.cpp~~

```

#include "min.h"
...
    = min (a, b);

```

```

gcc -c min.cpp
gcc -c test.cpp
gcc test.o min.o

```

Kleine Funktionen sollten mit `inline` eingebaut werden, da sonst der Funktionsaufruf mehr Ressourcen verbraucht als die Funktion selbst.

```

template<class typename T>
T min (T a, T b)
{
    return a<b? a : b;
}

int min (int i, int j)
{
    return a<b? a : b;
}

double min (double a, double b)
{
    return a<b? a : b;
}

```

```

int main()
{
    int a=10;
    int b=12;
    ... = min (++a, b);
    e = min (pc, d);           oder
    ... = min<double> (c, d);
    x = min ((double) a, d);   oder
    ... = min<double> (a, d);
}

```

Vergleicht man mit der Methode `min()` zwei Strings, wird das Programm die Zeigeradressen der Strings vergleichen. Beispiel

```
... = min ("abc", "def");
```

vergleicht den Zeiger auf `a` mit dem Zeiger auf `d`. Deshalb muss man die Methode `min()` überschreiben. Templates lassen sich nicht überschreiben, deshalb muss in der Hauptdatei spezialisiert werden:

```

#include <string.h>
template<>
const char * min <const char * c.c.*a, c.c.*b)
{
    strcmp(a,b) < 0 ? a : b;
}

```

```

template <typename T, const int i>
T add(T a, T b)
{
    return a+b+i;
}

...
... = add<int, 5> (5,7);
... = add<int, 0> (...); kann man mithilfe des Default-Parameters vereinfachen.

```

Die Templatedefinition muss dann lauten:

```
template <typename T, const int i=0>
```

So kann der Aufruf wie folgt erfolgen:

```
... = add (5,7);
```

~~min.cpp.h~~

```
template <...>  
T min (T ...)  
{ ...  
}  
int min<int> (int,int);
```

~~min.h~~

```
template <...> T min (T, T);
```

test.cpp

```
#include "min.h"  
... = min (10, 15);  
up();
```

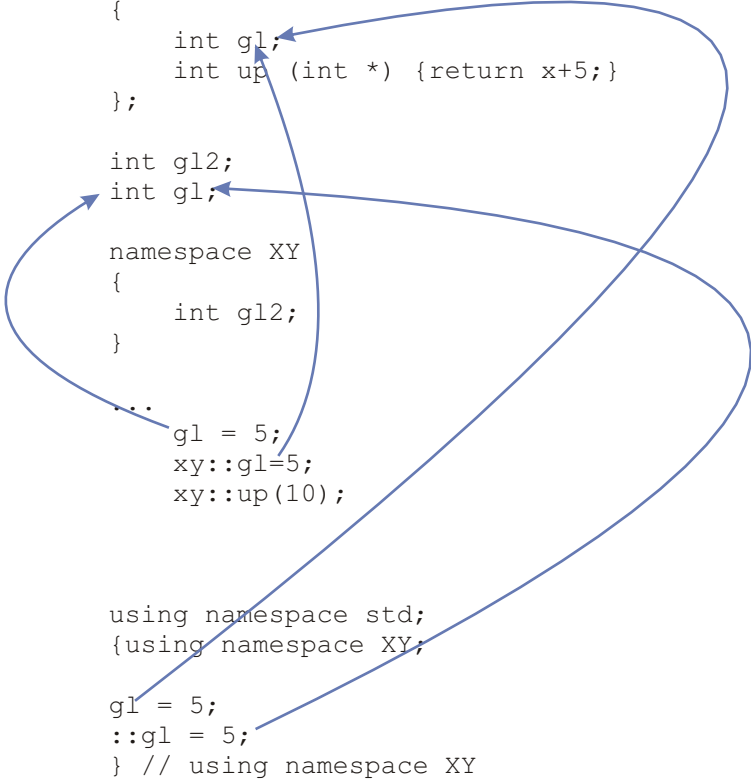
up.cpp

```
#include "min,h"  
void up()  
{  
    ... = min (10,12);  
}
```

```
namespace XY  
{  
    int gl;  
    int up (int *) {return x+5;}  
};
```

```
int gl2;  
int gl;  
  
namespace XY  
{  
    int gl2;  
}  
...  
gl = 5;  
xy::gl=5;  
xy::up(10);
```

```
using namespace std;  
{using namespace XY;  
  
gl = 5;  
::gl = 5;  
} // using namespace XY
```



```

#include <string>
...

std::String S1="Hallo",
             S2("auch Hallo"),
             S3;

S3 = S1;
S3 = "Ochse";
S2 = S1 + S3;

```

um Funktionen aus C-Bibliotheken unter C++ verwenden zu können, verwendet man:

```

#include <stdlib>
z.B. für std::malloc(...);
...

```

```

#include <cstring>
z.B. für std::strcpy(...);

```

```

#include <string>

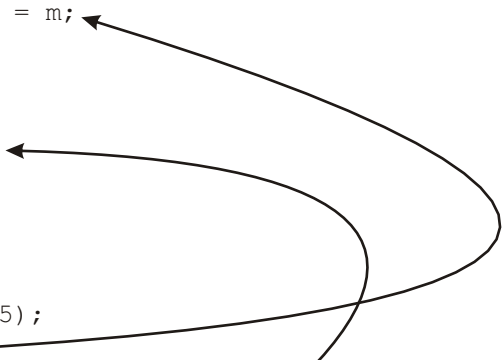
class Stud
{
public:
    Stud (c.c. * v, c.c. * n, int m)
    {
        vn = v; nn = n; matr = m;
    }

private:
    std::string vn, nn;
    int matr;
};

...

Stud s ("Karl", "Klammer", 25);

```



ist nicht effizient, weil vn, nn und matr erst im private-Bereich initialisiert werden. Besser ist:

```

public:
    Stud (c. c. * v, c.c. * n, int m)
        : vn(v), nn(n), matr(m)

```