Statische Komponenten in Templates

`static`-Template-Komponenten sind für jede erzeugte Klasse von einander verschieden.

Bsp.: Linke Nullteiler in endlichen Ringen

```
#include <iostream>
#include <iomanip>

using namespace std;

template<int n> class Z
{
  private:
    int k;
  public:
    Z<n>(int kk=0) { k=kk%n; if (k<0) k+=n; }
    Z<n> operator+(Z<n> z) const
      { Z<n> temp(*this); temp.k=(temp.k+z.k)%n; return temp; }
    Z<n> operator*(Z<n> z) const
      { Z<n> temp(*this); temp.k=(temp.k*z.k)%n; return temp; }
    friend ostream& operator<< (ostream& stream, Z<n> z)
      { stream << z.k; return stream; }
    // praefix++: Nachfolger
    Z<n>& operator++() { ++k; return *this; }
    bool operator==(Z<n> z) const { return k==z.k; }
    bool operator!=(Z<n> z) const { return k!=z.k; }
    static Z<n> null()  { return Z<n>(0); }
    static Z<n> start() { return Z<n>(0); }
    static Z<n> stop()  { Z<n> z; z.k=n; return z; }
};

template<class R, class S> class Prod
{
  private:
    R r;
    S s;
  public:
    Prod(R rr=R::null(),S ss=S::null()) : r(rr),s(ss) {}
    Prod operator+(Prod p) const
      { Prod temp(*this); temp.r=temp.r+p.r; temp.s=temp.s+p.s; return temp; }
    Prod operator*(Prod p) const
      { Prod temp(*this); temp.r=temp.r*p.r; temp.s=temp.s*p.s; return temp; }
    friend ostream& operator<< (ostream& stream, Prod p)
      { stream << "(" << p.r << "," << p.s << ")"; return stream; }
    // praefix++: Nachfolger, lexikographische Ordnung
    Prod& operator++()
      { if (++s==S::stop()) { s=S::start(); ++r;
                              if (r==R::stop()) s=S::stop(); }
        return *this; }
    bool operator==(Prod p) const { return p.r==r && p.s==s; }
    bool operator!=(Prod p) const { return p.r!=r || p.s!=s; }
```

```
    static Prod null()  { return Prod(R::null(),S::null()); }
    static Prod start() { return Prod(R::start(),S::start()); }
    static Prod stop()  { return Prod(R::stop(),S::stop()); }
};

template<class R> bool ist_linkerNullteiler(R a)
{
  if (a == R::null()) return false;
  for (R b=R::start(); b!=R::stop(); ++b)
    if (a*b == R::null() && b != R::null()) return true;
  return false;
};

int main()
{
  Z<10> z;
  for (z=Z<10>::start(); z!=Z<10>::stop(); ++z)
    if (ist_linkerNullteiler(z))
      cout << "Linker Nullteiler: " << z << endl;
  cout << endl;

  Prod<Z<3>,Z<4>> p;
  for (p=Prod<Z<3>,Z<4>>::start(); p!=Prod<Z<3>,Z<4>>::stop(); ++p) {
    if (ist_linkerNullteiler(p))
      cout << "Linker Nullteiler: " << p << endl;
  }
  return 0;
}
```

Templates in Klassen

Innerhalb von Klassen oder Klassentemplates können wiederum Klassentemplates oder Funktionstemplates vereinbart werden (member templates).

Bsp.: Template-Deklarationen in `<vector>`

```
template <class T, Class Allocator = allocator<T>> class vector {
  public:
                          :
    template <class InputIterator>
      vector (InputIterator first, InputIterator last,
              const Allocator& = Allocator());
                          :
    template <class InputIterator>
      void insert(iterator position, InputIterator first, InputIterator last);
                          :
};
```

Funktionsobjekte in STL-Behälterklassen (`<functional>`)

Funktionsobjekte kommen in der Standard Template Library (STL) sehr häufig zum Einsatz, weil die entsprechenden Funktorklassen in einfacher Weise als Template-Parameter übergeben werden können. In Behälterklassen stellen sie einen Ersatz für schwieriger zu handhabende Funktionsparameter dar.

Bsp.: Selbstdefinierte Funktorklasse für map

```
            :
class Less {
  public:
    bool operator()(string s,string t) {
      return s<t;
    }
};

int main()
{
  map<string,int,Less> a;
            :
  map<string,int,Less>::iterator pos;
            :
}
```

Bem.: Die Zusammenarbeit mit der STL funktioniert im allg. besser wenn folgende Ableitung vom Klassentemplate `binary_function` (aus `<functional>`) vorgenommen wird:

```
class Less : public binary_function<string,string,bool>
```

Statt der selbstdefinierten Klasse `Less` kann auch die Standardklasse `less<string>` benutzt werden.

Bsp.: map mit Standardfunktorklasse

```
#include <functional>
            :

int main()
{
  map<string,int,less<string>> a;
            :
  map<string,int,less<string>>::iterator pos;
            :
}
```

Letzteres entspricht der Voreinstellung, weil

`map<I,T>` $\equiv$ `map<I,T,less<T>>`

und für das Funktionsobjekt `less<T>()` gilt

`less<T>()(x,y)` $\equiv$ x<y

Weitere Funktorklassentemplates für Vergleiche in `<functional>`:
`greater`, `greater_equal`, `less_equal`, `equal_to`, `not_equal_to`.

[Entsprechende Funktorklassentemplates existieren auch für binäre arithmetische Operatoren (`plus`, `minus`, `multiplies`, `divides`, `modulus`), die Vorzeichenumkehr (`negate`) und logische Operatoren (`logical_and`, `logical_or` und `logical_not`).]

Manchmal ist es zweckmäßig (z.B. zur Vermeidung der Codeaufblähung durch unterschiedliche Templateinstanziierungen), zusätzliche Funktorklassen zu vermeiden.

Bsp.: map mit Vergleichsfunktion als Konstruktorargument

```
              :
bool lesstest(string s, string t)
{
  return s<t;
}

int main()
{
  map<string,int,bool (*)(string,string)> a(lesstest);
              :
  map<string,int,bool (*)(string,string)>::iterator pos;
              :
}
```

Das funktioniert, weil in der Vereinbarung

```
map<I,T,Comp> a(comp)
```

für Vergleiche das Funktionsobjekt *comp* aus der Funktorklasse *Comp* verwendet wird und an Stelle von Klassen auch andere Datentypen als Templateargumente verwendet werden dürfen.

Bsp.: Umformulierung des vorigen Beispiels mit STL-Funktionsadapter

```
#include <functional>
              :
int main()
{
  map<string,int,pointer_to_binary_function<string,string,bool>> a(ptr_fun(lesstest));
              :
  map<string,int,pointer_to_binary_function<string,string,bool>>::iterator pos;
              :
}
```

Es muss sorgfältig zwischen Funktorklassen und Funktionsobjekten unterschieden werden. Während die Behälterklassen in der Regel Funktorklassen als Templateargumente akzeptieren, kommen in den Funktionstemplates aus `<algorithm>` häufig Funktionen oder Funktionsobjekte als Argumente vor.

Bsp: sort mit Standardfunktionsobjekt als Argument

```
#include <functional>
            :
int main()
{
   vector<double> a;
            :
   sort(a.begin(),a.end(),greater<double>());  // okay
// sort(a.begin(),a.end(),greater<double>);    // falsch!!
            :
}
```