**Problem:**

Vererbung erlaubt die Wiederverwendung von Programmcode, virtuelle Funktionen ermöglichen dabei den Austausch gewisser Funktionalität

Vererbung erlaubt NICHT die Wiederverwendung durch Parametrisierung von Klassen, z.B.

```
class Stack_of_int{...};
```
und
```
class Stack_of_double{...};
```

NICHT (sinnvoll) realisierbar als:

```
class Stack { ... }; // abstract ??
class Stack_of_int: public Stack { ... };
class Stack_of_double: public Stack { ... };
```

**Lösung:**

C++ erlaubt die Definition von generischen Klassen:
solche, die von (noch nicht spezifizierten) Typen oder Werten abhängen !

```
// GenericStack.h :
template <class T, int D> // variabel in T und D
class Stack {
protected:
   T *data;
   int top, max;
public:
   Stack(int dim = D): data(new T[dim]), top(0), max(dim) {}
   ~Stack() { delete [] data; } // beide hier inline
   void push (T i); // outline
   T pop();         // outline
};
```

Die Definition von Memberfunktionen außerhalb von generischen Klassen hängt damit selbst von diesen Typen oder Werten ab !

```
// GenericStack.cpp ???
template <class T, int D>
void Stack<T,D>::push (T i) {
      data[top++]=i;
}

template <class T, int D>
T Stack<T,D>::pop () {
      return data[--top];
}
```

eine Template-Klasse definiert ein allgemeines Muster für beliebig viele konkrete Klassen,

aus dem Template selbst lässt sich jedoch (i. allg.) kein Code generieren,

dies geschieht erst bei der sog. Instantiierung der Template-Klasse

```
#include "GenericStack.h"
Stack<int, 10> s1, s2;
Stack<double, 20> s3;
Stack<Stack<int, 10>, 10> ss;
void foo() {
  s1.push(1);  s2.push(2);
  ss.push(s1); ss.push(s2);
  s3.push(3.7);
}
```

Daher muss bei der Instantiierung einer Template-Klasse der Quelltext aller (benutzten) (auch der nicht-inline) Funktionen zur Verfügung stehen:

übliche Verfahrensweise:

```
// GenericStack.h :
template <class T, int D>
class Stack { /* wie oben */ };
#include "GenericStack.cpp"
```

die Instantiierung einer konkreten Klasse geschieht auf explizite Anforderung (erste Verwendung), eine solche kann jedoch weitere Instantiierungen nach sich ziehen

Instantiierung 'auf Vorrat' (ohne sofortige Verwendung) ist möglich:

```
template class Stack<int, 10>;
template class Stack<double, 20>;
template class Stack<Stack<int, 10>, 10>;
```

`<class XYZ>` bedeutet nicht, dass nur mit Klassentypen instantiiert werden darf, vielmehr kündigt `class` einen *'Meta-(Typ-)Parameter'* an, alternativ kann hier auch das neue Schlüsselwort `typename` verwendet werden

Wozu dient das neue Schlüsselwort **typename** ?

```
template <typename T> class Beispiel { ...
   T::X * x;
   // ist nach C++-Grammatik mehrdeutig
};

class T1 { public: static int X; };
```

**Beispiel<T1>**-Instantiierung: * ist Multiplikation

```
class T2 { public: typedef int X; };
```

**Beispiel<T2>**-Instantiierung: * ist Zeigertyp-Konstrukt

```
template <typename T> class Beispiel { ...
   typename T::X * x; // X muss in T ein Typname sein
}
```

# Voreinstellungen für Template-Parameter sind möglich

```
template <class T = int, int D = 20>
class Stack {....};
Stack<> s; // Stack<int,20> !
```

**ACHTUNG Lexikfalle:**

```
template <typename T> class X { ... };
X<X<T>> xxt;  // ERROR: operator >> ???
// demnächst erlaubt in C++0x :-)
X<X<T> > xxt; // OK
```

# Individuelle Implementationen für spezielle Template-Parameter sind möglich:

```
template <> // alle Parameter gebunden!
class Stack<std::string, 100> {
// Stacks von strings kann man u.U.
// anders/spezieller/effizienter/cleverer
// implementieren
... };
template<>
void Stack<std::string, 100>::push
          (const std::string& s) { .... }
```

# Individuelle (partielle) Implementationen für spezielle Template-Parameter sind möglich:

```
template <typename T1, typename T2>
class MyClass { // allgemeinste Form
.... };
template<typename T>
class MyClass<T, T> { // T1 und T2 sind gleich
.... };
template<typename T>
class MyClass<T, int> { // T2 ist int
.... };
template <typename T1, typename T2>
class MyClass<T1*, T2*> { // beide sind Zeigertypen
.... };
          // ACHTUNG MyClass<int, int> nicht eindeutig!
```

Einzelne Memberfunktionen können eigene Template-Parameter (über die Klasse hinaus) besitzen: *member templates*

```
template <typename T>
class Stack {
      // Stack<S> und Stack<T> sollen zuweisbar
      // sein, wenn S und T zuweisbar sind
      template <typename T2>
      Stack<T>& operator=(const Stack<T2>&);
.... };                  // default op= bleibt erhalten !
template <typename T>
  template <typename T2>
    Stack<T>& Stack<T>::operator=(const Stack<T2>& o)
              { .... }
```

# Template-Parameter können selbst wieder Template-Typen sein: *template template parameters*

```
template <typename T>
class Stack { ... };

template <typename T, typename Container>
class LIFO {  Container elems; };

LIFO<int, Stack<int> > l; // int redundant
//besser:
template <typename T,
    template <typename E> class Container = Stack>
class LIFO {  Container<T> elems; }
LIFO<int> l1; LIFO<int, otherStack> l2;
```

kann fehlen, weil nicht benutzt

NICHT typename

Template-Code muss zunächst nur syntaktisch korrekt sein. Erst bei der Instantiierung wird semantisch Korrektheit geprüft.

Für Klassentemplates werden nur die Funktionen instantiiert, die auch benötigt werden, nur diese müssen fehlerfrei übersetzbar sein !

SFINAE: ***S**ubstitution **f**ailure **i**s **n**ot **a**n **e**rror*

*generic programming* eröffnet völlig neue Möglichkeiten

Beispiel: Zuweisbarkeit von Typen statisch entscheiden
Ist ein Typ T in einen Typ U konvertierbar ?

```
// s. Andrei Alexandrescu: Modern C++ Design

template <class T, class U>
class Conversion {
   typedef char Small;
   class Big {char dummy[2];}; // guaranteed bigger !
   static Small Test(U);       // not implemented !
   static Big   Test(...);     // not implemented !
   static T MakeT();           // not implemented !
public:
   enum {
     exists = sizeof(Test(MakeT())) == sizeof(Small),
     same = false };
};
```

### 3. Generische Programmierung in C++

```
int main() {
  using namespace std;
  cout
   << Conversion<double, int>::exists << ' '
   << Conversion<char, char*>::exists << ' '
   << Conversion<size_t, vector<int> >::exists << ' ';
}
// double ---> int ?
// Test(MakeT()) overloading zwischen int und ...
// int passt besser: Test liefert ein Small
// sizeof-Vergleich liefert true, exists ist 1
```

```
1 0 0
```

```
// even more is possible:
template <class T>
class Conversion<T,T> { // same Type
public:
   enum { exists = 1, same = 1 };
};

#define SUPERSUBCLASS (T, U) \
  (Conversion<const U*, const T*>::exists &&\
  !Conversion<const T*, const void*>::same)
// true if U inherits publicly from T
// or T and U are identically
```

Auch Funktionen können als Templates implementiert werden *function templates*

```
template <typename T>
const T& max (const T& a, const T& b) {
      return a > b ? a : b;
}
const int&  max (const int& a, const int& b) {
      return a > b ? a : b;
}
```

'universell' überladene Funktion, Überladung wird bevorzugt in non-template Code aufgelöst (SFINAE bei template Code), wenn eine template-Funktion exakt passt, ansonsten Typumwandlungen nötig sind, wird die Schablone benutzt!
Funktionskörper muss für Compiler zugänglich sein
---> *inlining* en passant

Aufruf auch ohne explizite Instantiierung möglich (wenn vorliegende Parameter diese eindeutig machen):

```
max ( 3, 5 ); // non-template max
max<> ( 3, 5 ); // max<int>
max ('a', 23.45); // non-template max
max ( 3.1241, 5.0 ); // max<double>
     // better match als non-template max !
// max ( 7, "sieben" ); ERROR
max ( 7, 23L ); // non-template max
max<long> ( 7, 23L ); // ok
```

Spezialisierungen sind natürlich möglich (weitere explizite Überladung)

```
inline const char*
max (const char* a, const char* b) {
      return (std::strcmp(a,b) > 0 ? a : b;
}
```

Der Template-Typ darf auch für lokale Variablen benutzt werden

```
template <typename T>
void swap (T& a, T& b) {
      T tmp(a); a = b; b = tmp;
}
```

keine Default-Template Argumente, keine Template Template Argumente und keine partielle Spezialisierung ist für Funktionstemplates erlaubt

# Dadurch können auch Algorithmen generisch werden

```
// swap kann beliebige Objekte 'austauschen'
// Benutzung z.B.:
template <class T>
void simple_sort (T* vec, int size) {
  for (int m=0; m<size-1; m++)
    for (int n=m+1; n<size; n++)
      if (vec[m]>vec[n]) swap(vec[m], vec[n]);
}
int v [100]; ....
simple_sort (v, 100);
```

der entstehende Programmcode muss semantisch korrekt sein:

```
class Y {};
class X {
      // operator > not defined
public:
      operator int () {return 1; }
};

X x [20];  Y y [40];
simple_sort (x, 20);
simple_sort (y, 40);

/* swap.cc: In function `void simple_sort(T*, int)
[with T = Y]':
swap.cc:34:   instantiated from here
swap.cc:15: no match for `Y& > Y&' operator
*/
```