Themen

- Templates
  - Templateklassen
  - Templatefunktionen
- numeric_limits

Was sind Templates?

Templates sind Schablonen, die es dem Programmierer erlauben, generischen Code einmal zu schreiben, um ihn anschließend auf verschiedene Datentypen anzuwenden.

Funktionstemplates

- Funktionstemplates stellen eine Funktion zur Verfügung, welche sich mit verschiedenen Datentypen nutzen läßt
- Die Deklaration und Definition ist der einer normalen Funktion sehr ähnlich, nur dass ein Teil der verwendeten Datentypen unbestimmt bleibt

max für int

```
// max Funktion für int

int max(int a, int b)
{
  return (a > b) ? a : b;
}
```

max für double

```
// max Funktion für double

double max(double a, double b)
{
  return (a > b) ? a : b;
}
```

Funktionstemplates

- beginnen mit dem Schlüsselwort **`template`**
- darauf folgt eine Komma getrennte Liste von Typnamen, jedem Typnamen wird das Schlüsselwort **`typename`** oder **`class`** vorangestellt
  `template<typename T,typename U,...>`
- Danach folgt die Deklaration mit Rückgabewert, Name und Argumenten, wobei einer oder mehrere der Datentypen durch generische Typnamen angegeben sind

Funktionstemplates

- für jeden Aufruf einer Templatefunktion prüft der Compiler die Argumente und generiert eine Instanz der Funktion mit den angegebenen Datentypen
- Die Definition der Templatefunktion muss an der Stelle an der sie benutzt wird bekannt sein, d.h. Templatefunktionen sind meist in Header Dateien implementiert

Funktionstemplate Beispiel

```
// max Funktion

template<typename TYPE>
TYPE max(TYPE a, TYPE b)
{
  return (a > b) ? a : b;
}
```

Funktionstemplate Beispiel

```
double a=3.5;
double b=4.6;
double maxAB=max(a,b);
int c=56;
int d=-34;
int maxCD=max(c,d);
string e=”Test”;
string f=”Auto”;
string maxFG=max(e,f);
```

Funktionstemplate Beispiel

```
double a=3.5;
int b=4;
double maxAB=max(a,b); // Fehler
```

Funktionstemplate Beispiel

```
double a=3.5;
int b=4;
double maxAB=max(a,(double)b);
```

Funktionstemplate Beispiel

```
double a=3.5;
int b=4;
double maxAB=max<double>(a,b);
```

Funktionen und Templates überladen

```
int max(int a,int b){...}   // 1
template<typename T>
T max(T a,T b){...}         // 2
template<typename T>
T max(T a,T b,T c){...}     // 3

max(7,42,68);               // -> 3
max(7.0,42.0);              // -> 2
max('a','b');               // -> 2
max(7,42);                  // -> 1
max<>(7,42);                // -> 2
max<double>(7,42);          // -> 2
max('a',42.7);              // -> 1
```

Benutzung von Funktionstemplates

- bei der Instanziierung von Funktions-templates werden die übergebenen Argumente nicht automatisch umgewandelt
- um ein Funktionstemplate zu benutzen, muss der Compiler in der Lage sein, die Datentypen eindeutig zu bestimmen, oder man muss sie explizit angeben.

Spezialisierung von Funktions-Templates

```
template<typename T>
T max(T a,T b){...}

const char* s1=”Apfel”;
const char* s2=”Birne”;
const char* maxS1S2=max(s1,s2);

// -> max<const char*>(const char*,const
 char*)
// -> Instanz vergleicht die Zeiger
 (Adressen) und nicht die C Strings!!!
```

Spezialisierung von Funktions-Templates

```
template<typename T>
T max(T a,T b){...}

template<>
const char* max(const char* a,const char* b)
{
    (strcmp(a,b)>0)?a:b;
}

const char* s1=”Apfel”;
const char* s1=”Birne”;
const char* maxS1S2=max(s1,s2);
```

Klassentemplates

- Klassentemplates stellen eine Familie von Klassen zur Verfügung, welche mit unterschiedlichen Datentypen funktionieren
- Deklaration und Definition sind der einer normalen Klasse sehr ähnlich, lediglich ein Teil der Datentypen bleibt unbestimmt
- Beim Instanziieren einer Klassentemplates müssen die Datentypen explizit angegeben werden.

Klassentemplate

```
template<typename DATA>
class Liste
{
  protected:
    DATA* mData;
  public:
     Liste(){mData=new DATA[10];}
     Liste(const Liste<T>& s){...}
    ...
     DATA getCurrent(){...}
    ...
};
```

Verwendung eines Klassentemplates

```
Liste<int> intListe;
intListe.addElement(25);
intListe.addElement(2.7);

Liste<double> doubleListe;
doubleListe.addElement(34.5);
doubleListe.addElement(2);
```

Stack

```
template<typename T>
class Stack
{
   ...
   T* mData;
   ...
   Stack(int MAX_SIZE)
   {
     mData=new T[MAX_SIZE];
   }
   ...
};
```

Benutzung von Stack

```
Stack<double> s1(100);
Stack<double> s2(50);

// s1 und s2 sind Instanzen
// derselbe Klasse!!!
```

Werte als Template Argumente

- die Typenliste einer Template Deklaration kann nicht nur unbestimmte Datentypen enthalten, sondern auch unbestimmte Werte

```
template<typename T,int MAX_SIZE>
class Stack
{
    T mData[MAX_SIZE];
        ...
};
```

Werte als Template Argumente

```
Stack<double,100> s1;
Stack<double, 50> s2;

// s1 und s2 sind unterschiedliche
 Klassen!!!
```

Template Default-Argumente

- wie bei Funktionen und Methoden können Template Argumente Default Werte haben

```
template<typename T,int MAX_SIZE=100>
class Stack
{
    T elements[MAX_SIZE];
        ...
};

Stack<double> s1;
Stack<double,200> s2;
```

Template Default-Argumente

- die Default Template Argumente stehen immer am Ende der Liste
- will man ein Default-Argument ändern, so muss man grundsätzlich alle (Default-) Argumente angeben, die davor stehen

Stack Beispiel

```
template<typename T=int,int MAX_SIZE=100>
class Stack
{
    T elements[MAX_SIZE];
        ...
};
  ...
Stack<> s1; // Stack für 100 int Werte
Stack<double> s2;// Stack für 100 double
                          Werte
Stack<20> s3;     // FALSCH
```

Spezialisierung von Klassentemplates

- ermöglicht es dem Programmierer, ein Klassetemplate für bestimmte Datentypen zu spezialisieren
- man kann eine ganze Klassetemplate spezialisieren oder nur einzelne Methoden
- wird die gesamte Klasse spezialisiert, müssen alle Methoden spezialisiert werden
- werden einzelne Methoden spezialisiert, kann anschließend nicht mehr die gesamte Klasse spezialisiert werden

Spezialisierung von Klassentemplates

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

template<> class Stack<const char*>
{   ...
    const char** elements;
    void push(const char* s){...};
    ...
};
```

Partielle Spezialisierung

- im Gegensatz zur normalen Spezialisierung wird bei der partiellen Spezialisierung nur ein Teil, der in den Template-Parametern enthaltenen Information festgelegt
- die restlichen Informationen müssen weiterhin bei der Instanziierung der Klasse angegeben werden

Partielle Spezialisierung von Klassentemplates

```
template<typename T1,typename T2>
class MyClass
{...};

template<typename T> class MyClass<T,T>
{...}; // T1 und T2 vom gleichen Datentyp

template<typename T> class MyClass<T,int>
{...}; // T2 ist ein int

template<typename T1,typename T2> class
  MyClass<T1*,T2*>
{...}; // T1 und T2 sind Zeiger
```

`this` in Templates

- benutze **`this`** für alle Zugriffe auf Methoden und Attribute in Template Klassen

```
template<typename T> class Base {
  void print();
};

template<typename T>
class Derived : public Base {
  void ausgabe(){ print(); }
       // ruft nicht Base::print auf!!!
};
```

Initialisierung von Werten in Templates

- Werte sollten in Templates immer über einen Konstruktoraufruf initialisiert werden

```
template<typename T>
class MyClass{
  protected:
     T member;
  public:
    MyClass():T(){};
       ...
};
```

- funktioniert auch für elementare Datentypen (`int`, `double`, `bool`, etc)

Vorsicht bei C Strings als Template Parameter

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

::max("Apfel","Birne");
::max("Apfel","Banane"); // Fehler
```

Vorsicht bei C Strings als Template Parameter

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

const char[6]

```
::max("Apfel","Birne");
::max("Apfel","Banane"); // Fehler
```

const char[7]

Vorsicht bei C Strings als Template Parameter

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

::max(“Apfel”,”Birne”);
::max(“Apfel”,”Banane”); // OK
```

Vorsicht bei C Strings als Template Parameter

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

::max<std::string>("Apfel","Birne");
::max<std::string>("Apfel","Banane");
```

Weitere Themen

- Template Parameters
  - Template Parameter können selbst wieder Templates sein
- Verschachtelte Templates
  - Methoden und/oder Attribute einer Template Klasse können selbst Templates sein
- Literatur:
  D. Vandevoorde/N. Josuttis: C++ Templates

numeric_limits

- definiert eine Reihe von systemspezifischen numerischen Konstanten, z.B. die größt mögliche und kleinst mögliche **`double`** Zahl
- um die Funktionen zu benutzen, muss man die Header Datei <limits> einbinden
- Funktionen sind in Template-Klassen implementiert wobei der Datentyp das Template Argument ist

numeric_limits

```
#include <limits>
      ...

double minD=numeric_limits<double>::min();
long maxL=numeric_limits<long>::max();
bool b=numeric_limits<char>::is_signed;
```

Standard Template Library (STL)

- definiert eine Reihe von Container Klassen (list, set, vector, deque, map, multiset, multimap, string,...)
- definiert eine Reihe von Algorithmen (z.B. Sortieralgorithmen)
- smart pointer
- alle Container und Algorithmen sind Templates
- aller Code steckt in Header Dateien