BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

C / C++ für Java-Programmierer - Generische Programmierung mit Templates -

Bachelor Medieninformatik

Prof. Dr.-Ing. Hartmut Schirmacher
http://schirmacher.beuth-hochschule.de
hschirmacher@beuth-hochschule.de

Lernziele + Inhalte

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Einführung in Template-Programmierung
- Inline-Code
- Code-Organisation für Templates, Inklusion vs. Separation
- Klassen-Templates, int-Parameter, Methoden-Templates
- Disambiguierung mittels typename
- Template-Spezialisierung

- Anhang
  - Template-Argumente...
  - Noch mehr Template-Spezialisierung
  - Type Traits

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Einführung in Templates

Generisches Programmieren

BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

- Definitionen für generisches Programmieren
  - Programmierung mit ***Typen als Parameter***
  - ***Möglichst abstrakte Repräsentation*** von Algorithmen und Datenstrukturen
  - Ada, Eiffel, Java, C++, C#, F#, Visual Basic .NET, ML, Scala, Haskell, ...
- Geschichte (auszugsweise)
  - 1971: Erste Ansätze generischer Programmierung für die Computer-Algebra (Musser)
  - 1983: *Generic Units* in der Programmiersprache Ada
  - 1985: Generische *Klassen* + Objektorientierung gemeinsam in *Eiffel*
  - 1987: Stepanov + Musser veröffentlichen generische Library für Ada;
    Forschung bei AT&T Bell Labs und Hewlett Packard Research Labs in C und C++
  - 1994: *Standard Template Library (STL)* wird Teil des ISO/ANSI C++ - Standard-Entwurfs
  - 1994: Eine Implementierung der STL (Stepanov, Lee, Musser) wird von Hewlett Packard frei zugänglich gemacht
  - 1994: *parameterized types* im Buch *Design Patterns* (Gamma et al.)
  - 1999: Start für das Projekt *Boost* – viele Vorlagen für neue C++ Standard-Templates
  - 2005: Support für Generics in Java J2SE 5.0

http://en.wikipedia.org/wiki/Generic_programming
http://en.wikipedia.org/wiki/Standard_Template_Library
http://en.wikipedia.org/wiki/List_of_C%2B%2B_template_libraries

Beispiel: Funktionstemplate in C++

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
template<typename T>
T myMax(T val1, T val2) {
    return val1 < val2? val2 : val1;
}
```

- Funktions-Template
  - *Template*-Parameter: Typ `T`
  - *Funktions*-Parameter: `T val1`, `T val2`
- Schlüsselworts `typename`
  - Anstelle von `typename` kann hier auch das Schlüsselwort `class` verwendet werden, dazu später mehr

Beispiel: Funktionstemplate in C++

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
template<typename T>
T myMax(T val1, T val2) {
    return val1 < val2? val2 : val1;
}

int main()
{

    cout << myMax<int>(1,4) << endl;

    cout << myMax(7,4) << endl;
```

⟵ ***Explizite Angabe*** des Template-Arguments.

⟵ ***Deduktion*** des Template-Arguments (`int`) anhand der Funktions-Argumente!

Beispiel: Funktionstemplate in C++

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
template<typename T>
T myMax(T val1, T val2) {
    return val1 < val2? val2 : val1;
}

int main()
{
    cout << myMax(1.0,4.1) << endl;

    cout << myMax(1,4.1) << endl;


    cout << myMax<double>(1,4.1) << endl;
```

← Auch hier automatische Deduktion des Template-Arguments (`double`)

← Compilerfehler!
Ist T `int` oder `double`?
An dieser Stelle ist keine automatische Umwandlung anwendbar.

← Auflösung durch explizite Angabe des Template-Arguments `T`.
Die Funktionsargumente werden dann wenn möglich nach `T` umgewandelt.

Mehrere Template-Argumente

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
template<typename T1, typename T2>
bool equals(const T1& val1, const T2& val2) {
    return val1 == val2;
}

int main() {

    cout << equals(5, 5.0) << endl;

    cout << equals(string("Hallo"), string("Welt")) << endl;
```

Zwei Template-Argumente.

T1 ist `int`, T2 ist `double`.

T1 und T2 sind `std::string` (funktioniert!)

Partielle Typ-Deduktion / Deduktion des Rückgabe-Typs

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
template <typename R, typename T>
R convert(T value) {
    return (R) value;
}

int main() {

    cout << convert<int,double>(5.0) << endl;

    cout << convert<int>(5.0) << endl;
```

← `R` und `T` explizit angegeben.

← `R` explizit angegeben, `T` deduziert!

Hier spielt die Reihenfolge der Template-Argumente eine Rolle; man kann nur die „hinteren“ weglassen, und nur dann, wenn sie automatisch deduziert werden können.

Es existieren komplexe Regeln dafür, wann welche Typen deduziert werden können. Siehe z.B. Josuttis, Vandevoorde, *C++ Templates*, Addison-Wesley, 2003.

Funktionsweise von Templates

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
template<typename T>
T myMax(T val1, T val2) {
    return val1 < val2? val2 : val1;
}
```

Template: Schablone

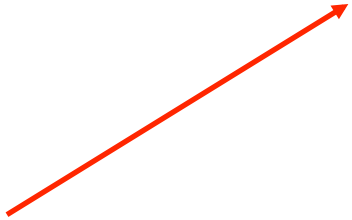
Textuelle Ersetzung

```
int myMax(int val1, int val2) {
    return val1 < val2? val2 : val1;
}
```

Template-***Instanz***

Template-***Anwendung***

```
cout << myMax(7,4) << endl;
```



- Ein Template ist eine ***Schablone***, die zur Compile-Zeit angewendet wird.
- Instanziierung funktioniert praktisch wie ***textuelle Ersetzung***
  - Anstelle der formalen Template-Parameter (`T`) werden die entsprechenden Argumente (`int`) eingesetzt.
  - Dann findet die eigentliche Kompilation inkl. Typ-Prüfung etc. statt.
  - Templates erzeugen ***keinerlei Laufzeit-Overhead***. Sie funktionieren genau so, als hätte der Entwickler den Code für die Template-Instanz tatsächlich geschrieben.

Funktionsweise von Templates

BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

```
int myMax(int val1, int val2) {
    return val1 < val2? val2 : val1;
}
```

Template-Instanz 1

```
double myMax(double val1, double val2) {
    return val1 < val2? val2 : val1;
}
```

Template-Instanz 2

Template-Anwendung

```
cout << myMax(7,4) << endl;
cout << myMax(3,9) << endl;
cout << myMax(1.0,4.1) << endl;
```

Achtung: Template-Instanz != Objekt-Instanz

- Code für ein Template wird nur für die ***Instanzen*** des Templates erzeugt, die ***tatsächlich angewendet*** werden.
- Für jede angewendete Kombination von Template-Parametern wird eine Template-Instanz erzeugt

  → ***bei Anwendung*** benötigt der Compiler ***Zugriff auf den Quellcode des Templates***

A propos „textuelle Ersetzung“: *inline*-Code

Inline-Funktionen und -Methoden

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
inline float calc(float a) {
  return a * 3.1415 + 4.0;
}
```

```
float f = calc(7.0);
```

Funktion wird im Code „quasi textuell“ ersetzt

```
float f = 7.0 * 3.1415 + 4.0;
```

- `inline`-Code: anstelle eines echten Funktionsaufrufs kann eine „textuelles Einsetzen“ des Funktionscodes treten

http://www.parashift.com/c++-faq-lite/inline-functions.html

Inline-Methoden

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
class X {
   ...
public:
   float calcRX(float a);
};
inline float X::calcRX(float a) {
   return a * m_rx * 3.1415 + 4.0;
}
```

- Das Schlüsselwort `inline`
  - gibt dem Compiler eine ***Empfehlung***, den Code einer Funktion direkt in den aufrufenden Code einzusetzen, ohne einen Funktionsaufruf
  - Kann Performance steigern (Übergabe der Parameter via Stack entfällt)
  - Kann erzeugten Code „aufblähen“ (Code wird mehrfach erzeugt)

http://www.parashift.com/c++-faq-lite/inline-functions.html

Inline-Methoden (2)

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
class X {
   ...
public:
   inline float calcRX(float a);
};

float X::calcRX(float a) {
   return a * m_rx * 3.1415 + 4.0;
}
```

- Alternative 2: `inline` in der *Deklaration*
  - ist äquivalent `inline` in der *Definition;*
  - gilt als nicht so guter Stil – denn inline ist ein ***Implementierungsdetail***.

http://www.parashift.com/c++-faq-lite/inline-functions.html

Inline-Methoden (3)

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
class X {
   ...
public:
   float calcRX(float a) {
     return a * m_rx * 3.1415 + 4.0;
   }
};
```

- Alternative 3: Methoden-Definition innerhalb der Klassendefinition
  - alle in der Klasse definierten Methoden sind ***implizit inline deklariert***
  - gilt ebenfalls als nicht so guter Stil: Die Klassendefinition soll die ***Schnittstelle*** zeigen, nicht die Implementierung

http://www.parashift.com/c++-faq-lite/inline-functions.html

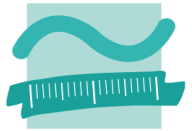
Code-Organisation für `inline`

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Bei inline setzt der Compiler den Quellcode an jeder Stelle wieder neu ein, wo die Funktion / Methode verwendet wird
- Quellcode muss dem Compiler überall bekannt sein, wo er „eingesetzt“ werden soll
- Zumal sollte der Code von inline-Funktionen relativ kurz sein

→ Daher schreibt man die Definition von inline-Code meistens in die ***Headerdatei***!

- Ein ähnliches Problem ergibt sich bei der Verwendung von Templates



BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Code-Organisation für Templates

Code-Organisation: das Problem

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Problem bei der Instanziierung von Templates
  - Bei der Anwendung eines Templates muss der Compiler ggf. eine neue Template-Instanz erzeugen.
  - D.h. an dieser Stelle muss die Implementierung (der Code) des Templates verfügbar sein.
  - Vergleiche `inline`-Funktionen!

Problem: Sichtbarkeit der Template-Implementierung

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Bei der Anwendung eines Templates muss dessen Implementierung für den Compiler sichtbar sein. Aber der Compiler übersetzt jede `.cpp` ***separat***.

x.h

```
/ Deklaration
template<typename T>
T mult(T a, T b);
```

Inklusion + Instanzierung

main.cpp

```
#inlcude x.h

...
  int i=..., j=...;
  int a=mult(i,j);
```

x.cpp

```
// Definition
template<typename T>
T mult(T a, T b) {
  return a*b;
};
```

Problem:

- Beim Compilieren von `x.cpp` wird ***keine*** Instanz von `mult()` benötigt / erzeugt.
- Beim Compilieren von `main.cpp` wird `mult(int,int)` benötigt; `x.h` versichert, dass es diese „irgendwo gibt“. Der Compiler kann hier nichts weiter tun.
- Der Linker merkt dann, dass `mult(int,int)` nirgendwo instanziert wurde.

```
Undefined symbols:
  "int mult<int>(int, int)", referenced from:
      _main in main.o
```

Inklusions-Modell

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Lösung: Implementierung des Templates steht zusammen mit der Deklaration in der ***Header-Datei***

x.h

```
// Deklaration + Definition
template<typename T>
T mult(T a, T b) {
  return a*b;
};
```

main.cpp

```
#inlcude x.h

...
  int i=..., j=...;
  int a=mult(i,j);
```

xy.cpp

```
#inlcude x.h

...
  float x=..., y=...;
  float f=mult(x,y);
```

Inklusions-Modell: `#include` der Implementierungs-Datei

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Alternative: Implementierung des Templates steht in einer separaten **`.cpp`- oder `.h` Datei**, die ***von der Header-Datei inkludiert*** wird.

x.h

```
#ifndef INCLUDE_X_H
#define INCLUDE_X_H

// Deklaration
template<typename T>
T mult(T a, T b);

#include "x.cpp"

#endif
```

x.cpp
oder
_x.h

```
///// TEMPLATES /////

// Definition
template<typename T>
T mult(T a, T b) {
  return a*b;
};
```

Auf diese Weise kann die Implementierung immer konsistent in der `.cpp`-Datei stehen; bei Templates wird diese jedoch in die Header-Datei eingefügt.

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Template-Klassen, Integer-Parameter

Definition einer Template-**Klasse**

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Klassen-Definition

```
template<typename T>
class Stack {
    T* m_data;
    int m_size, m_capacity;
  public:
    Stack(int capacity = 10);
    ~Stack();
    void push(const T& elem);
    T pop();
    T top() const;
};
```

Implementierung

```
template<typename T>
Stack<T>::Stack(int capacity) {
    m_data = new T[capacity];
    m_size = 0;
    m_capacity = capacity;
}
template<typename T>
Stack<T>::~Stack() {
    delete[] m_data;
}
```

Die Klasse wird als Template mit einer beliebigen Liste von Template-Parametern deklariert. Verwendung der Parameter-Namen analog wie bei den Template-Funktionen.

Außerhalb der Klassen-Definition muss vor dem `::` jeweils **der vollständige Typname** der Klasse inkl. Template-Argumentliste angegeben werden.

Instanziierung und Kompilation der Methoden

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Anwendung

```
int main(){
    Stack<float> s;
    return 0;
}
```

Kompiliert, alles ok?

```
int main(){
    Stack<float> s;
    s.push(5.0);
    return 0;
}
```

Kompiliert nicht – erst jetzt merkt der Compiler, dass bestimmte Methoden zwar deklariert, aber gar nicht implementiert sind!

```
Undefined symbols:
  "Stack<float>::push(float const&)", referenced from:
```

Bei der Entwicklung von Templates muss unbedingt darauf geachtet werden, alle Klassen, Funktionen und Methoden auch tatsächlich zu instanziieren, um Kompilier-Fehler zu finden.
(siehe auch: explizit Instanziierung von Templates)

`int` als Template-Parameter. Beispiel `Vec<T,N>`

BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

```
template<typename T, int N>
class Vec {
    T m_data[N];
  public:

    Vec();

    const T& operator[](int i) const { return m_data[i]; }
    T& operator[](int i) { return m_data[i]; }

    float dot(const Vec& rhs) const;
    Vec cross(const Vec& rhs) const;

    Vec& operator+=(const Vec& rhs);
    Vec& operator-=(const Vec& rhs);
    Vec& operator*=(const Vec& rhs);
    Vec& operator/=(const Vec& rhs);
};
```

Neben Typen können auch „gewöhnliche“ Ganzzahlen-Werte als Template-Parameter verwendet werden (hier ein `int`-Wert namens `N`)

In der Implementierung kann der Parameter `N` dann wie eine ***Konstante*** verwendet werden.

Skalarprodukt und Kreuzprodukt als Methoden

Komponentenweise Operatoren

Vec<T,N>: Implementierung

BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

```
template<typename T, int N>
Vec<T,N>::Vec() {
    for(int i=0; i<N; ++i)
        m_data[i]=T();
}
```

Der vollständige Typname ist **`Vec<T,N>`**.
Der Klassenname (z.B. Konstruktor-Name) ist **Vec**.

Ruft den Default-Konstruktor für jedes Element einzeln auf (würde z.B. einen `int` mit 0 initialisieren).

```
template<typename T, int N>
Vec<T,N>&
Vec<T,N>::operator+=(const Vec<T,N>& rhs) {
    for(int i=0; i<N; ++i)
        this->m_data[i] += rhs.m_data[i];
    return *this;
}
```

Auch beim Rückgabe-Typ und bei Argumenten in der Parameterliste muss der vollständige Typname **`Vec<T,N>`** angegeben werden.

Nicht vergessen: Definitionen müssen letztlich in der Headerdatei stehen / inkludiert werden.

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Template-Methoden

Template-Methoden

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Nicht nur Funktionen und Klassen können Templates sein, sondern auch ***einzelne Methoden*** einer Klasse.
- Dabei muss die Klasse selbst kein Template sein.

```
class MyFloat {

    float m_value;

public:

    MyFloat(float v=0) : m_value(v) {}

    template<typename T, int N>
    MyFloat& addComponents(const Vec<T,N> &rhs) {
        for(int i=0; i<N; ++i)
            m_value += (float) rhs[i];
        return *this;
    }
};
```

Klasse ist kein Template.

Methode ist ein Template und bildet einen Adapter zu beliebigen `Vec<T,N>`.

(sorry, kein besonders sinnvolles Beispiel)

Template-Methoden in Template-Klassen

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Beispiel für eine Template-Methode in einer Template-Klasse:

```
template<typename T, int N>
class Vec {

    // ...

    template<typename T2>
    Vec& operator=(const Vec<T2,N>& rhs);

};

template<typename T, int N>
template<typename T2>
Vec<T,N>& Vec<T,N>::operator=(const Vec<T2,N>& rhs) {
    for(int i=0; i<N; ++i)
        m_data[i]= (T) rhs[i];
}
```

Klasse ist ein Template.

Methode ist ebenfalls ein Template mit einem ***weiteren*** Parameter.

Template-Parameter der Klasse

Template-Parameter der Methode

Disambiguierung mittels `typename`

Disambiguierung mittels `typename`

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Innerhalb einer Template-Definition gibt es Situationen, in denen der Compiler einen Typ nicht von einem Attribut unterscheiden kann:

```
template<typename T>
class MyClass {
  T::X *ptr;
};
```

← Was ist X?

Es könnte ein Attribut der Klasse `T` sein,
dann hieße der Ausdruck: `X` multipliziert mit `ptr`.

- Zur Auflösung dieser Fälle wurde „typename“ eingeführt

```
template<typename T>
class MyClass {
  typename T::X *ptr;
};
```

← `T::X` ist ein Typ-Bezeichner.

Also wird ein Zeiger namens `ptr` deklariert.

Template-Spezialisierung

Spezialisierung eines Templates

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Template-Spezialisierung
  - `template<typename T> class X;` ← Allgemeines Template
  - `template<> class X<int> { ... };` ← Spezialisierung für **`int`**
  - `template<typename T> class X<T*> { ... };` ← Spezialisierung für beliebige Zeiger
- Partielle Spezialisierung
  - `template<typename T1, typename T2> class X;`
  - `template<typename T1> class X<T1,float> { ... };`

Partielle Spezialisierung:
erster Parameter bleibt “offen“,
zweiter Parameter ist **`float`**

Spezialisierung einer Template-Funktion oder -Klasse

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Ein Template fungiert als Schablone für ***alle möglichen*** Template-Argumente
- C++ erlaubt es, für ***einige*** der möglichen Argumente eine ***spezialisierte Implementierung*** anzugeben

```
template<typename T1, typename T2>
class MyClass {
public: MyClass() { cout << "T1:T2" << endl; }
};
```

← Allgemeines Template mit zwei Parametern `T1` und `T2`.

```
template<>
class MyClass<float,int> {
public: MyClass() { cout << "float:int" << endl; }
};
```

← Vollständig spezialisiertes Template

`T1 = float` und `T2 = int`

Anhang: Template-Argumente

Default-Template-Argumente, geschachtelte Templates

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Einfaches Default-Argument
  - `template<typename T = float> class Vec;`
- Default-Argument, welches sich auf vorhergehenden Parameter bezieht
  - `template<typename T, typename Order=Less<T> >`
    `class SortedContainer;`

Achtung! Zwischen dem inneren und dem äußeren Template-Typ muss ein Leerzeichen stehen, sonst Verwechslung mit `operator>>`! Funktioniert aber endlich ab C++11 ....

Anwendungsbeispiel: Funktor zur Definition einer Ordnung

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Definition eines Template-Funktors Less<T>
  - ```
    template<class T>
    class Less {
      public:
       bool operator()(const T& lhs, const T& rhs) {
         return lhs < rhs;
       }
    };
    ```
- Verwendungs-Arten
  - ```
    Less<float> lessFloat;
    cout << "5.0 < 4.9 == " << lessFloat(5.0,4.9);
    ```
  - ```
    cout << Less<int>()(3,4);
    ```
    ```

Template-Template-Argumente

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Zurück zum `SortedContainer`-Template
  - ```
    template<typename T, typename Order = Less<T> >
    class X {
      Order m_order;
    };
    ```
  - ```
    X< float, Greater<float> > myX;
    ```

- Unschön (?), dass der ***Typ float hier zweimal angegeben*** werden muss. Lösung mittels Template-Template-Parametern:
  - ```
    template<typename T, template<typename> class Order = Less >
    class Y {
      Order<T> m_order;
    };
    ```
  - ```
    Y< float, Greater > myY;
    ```


Hier darf nur **`class`** stehen, nicht etwa typename.

Anhang: noch mehr Template-Spezialisierung

Partielle Spezialisierung

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Es ist auch möglich, nur einen Teil der Template-Parameter zu spezialisieren und den Rest offen zu lassen → ***partielle Spezialisierung***.

```
template<typename T>
class MyClass<T,int> {
public: MyClass() { cout << "T:int" << endl; }
};
```

← Partiell spezialisiertes Template

`T1` bleibt offen, `T2 = int`

Allgemeines Template:

```
template<typename T1, typename T2>
class MyClass {
public: MyClass() { cout << "T1:T2" << endl; }
};
```

Partielle Spezialisierung: weitere Beispiele

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Partielle Spezialisierung auf nur einen Parameter; dieser wird dann für beide Template-Parameter eingesetzt:

```
template<typename T>
class MyClass<T,T> {
public: MyClass() { cout << "T:T" << endl; }
};
```

Partiell spezialisiertes Template

Der neue Parameter T wird für `T1` und `T2` eingesetzt

Allgemeines Template:

```
template<typename T1, typename T2>
class MyClass {
public: MyClass() { cout << "T1:T2" << endl; }
};
```

Spezialisierung auf Zeigertypen

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Spezialisierung mit gleicher Anzahl Template-Parameter, jedoch nur für Zeigertypen:

```
template<typename T1, typename T2>
class MyClass<T1*,T2*> {
public: MyClass() { cout << "T1*:T2*" << endl; }
};
```

T1 und T2 bleiben offene Parameter

Aber es müssen Zeiger sein!

Allgemeines Template:

```
template<typename T1, typename T2>
class MyClass {
public: MyClass() { cout << "T1:T2" << endl; }
};
```

Spezialisierung und Konflikte

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Folgende Spezialisierungen von MyClass<T1,T2> liegen vor:
  - **`MyClass<float,int>`**
  - **`MyClass<T,int>`**
  - **`MyClass<T,T>`**
  - **`MyClass<T1*,T2*>`**

- Welche Spezialisierung wird jeweils verwendet?
  - `MyClass<float,float> f_f;` **`MyClass<T,T>`**
  - `MyClass<float*,int*> fp_ip;` **`MyClass<T1*,T2*>`**
  - `MyClass<float,int> f_i;` **`MyClass<T,int>`**
  - `MyClass<float*,int> fp_i;` **`MyClass<T,int>`**
  - `MyClass<int,int> i_i;` **Compilerfehler!**

Wie löst man diesen Konflikt?

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
ambiguous class template instantiation for 'class MyClass<int, int>'
candidates are: class MyClass<T, T>
    error:              class MyClass<T, int>
```

- Die versuchte Template-Instanzierung `MyClass<int,int> i_i;` erzeugt obige Fehlermeldung

- Die einfachste (und einzige?) Lösung ist es, eine weitere Spezialisierung für den Konfliktfall zu definieren:

```
template<>
class MyClass<int,int> {
public: MyClass() { cout << "int:int" << endl; }
};
```

Anwendung der Spezialisierung: Optimierung

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Anwendungsbeispiel: optimierte Berechnung für einen bestimmten Typ

```
template<typename T, int N>
inline Vec<T,N>
operator+(const Vec<T,N>& lhs, const Vec<T,N>& rhs) {
    Vec<T,N> result(lhs);
    return result += rhs;
}
```

Allgemeines Template mit zwei Parametern T und N.

```
template<>
inline Vec<float,4>
operator+(const Vec<float,4>& lhs, const Vec<float,4>& rhs) {
    return SSE::add(lhs,rhs);
}
```

Spezialisierung der Template-Funktion für `T=float` und `N=4`.

hier Z.B. Ausnutzung von Prozessoren mit SSE-Befehlssatz.

Interesse an SSE-Optimierung? http://neilkemp.us/src/sse_tutorial/sse_tutorial.html

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Anhang:
Type Traits

Anwendung der Spezialisierung: Type Traits

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Type Traits: Zusatz-Informationen zu Typen, um einfach generische Algorithmen programmieren zu können.

```
template< class T >
T findMax(const T* data, size_t numItems) {
    T largest =  <Minimal-Wert des Typs T>;
    for(size_t i=0; i<numItems; ++i)
        if (data[i] > largest)
            largest = data[i];
    return largest;
}
```

Dieser Algorithmus kann natürlich auch programmiert werden, ohne den Minimum-Wert zu verwenden. Allerdings muss man sich dann eine besondere Behandlung für `numItems == 0` ausdenken.

Anwendung der Spezialisierung: Type Traits

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Type Traits: Zusatz-Informationen zu Typen, um einfach generische Algorithmen programmieren zu können.

```
template< class T >
T findMax(const T* data, size_t numItems) {
    T largest = numeric_limits<T>::min();
    for(size_t i=0; i<numItems; ++i)
        if (data[i] > largest)
            largest = data[i];
    return largest;
}
```

Hier kann auf generische Weise ein „Trait“ eingesetzt werden.

- In C++ sind Type Traits Templates, die Typ-Informationen kapseln.
- Dabei wird die Trait-Klasse einfach für jeden Typ spezialisiert.
- Traits ermöglichen es, generischen Code stark zu vereinfachen.
- Später mehr unter dem Stichwort ***generische Programmierung***.

Anwendung der Spezialisierung: Type Traits

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Beispiel eines Type Traits: Spezialisierung für `unsigned int`

```
template<>
class NumTraits<unsigned int> {
  public:
    static const char* name() { return "unsigned int"; }
    static bool hasSign() { return false; }
    static unsigned int minValue() { return 0u; }
    static unsigned int maxValue() { return UINT_MAX; }
};
```

Spezialisierung der Traits

Weiteres Beispiel für Type Traits

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Generischer Algorithmus, der das größte Element in einem Array findet

```
template< class T >
T findMax(const T* data, size_t numItems) {
    T largest =  <Minimal-Wert des Typs T>;
    for(size_t i=0; i<numItems; ++i)
        if (data[i] > largest)
            largest = data[i];
    return largest;
}
```

Was liefert die Funktion bei `numItems == 0`?
Häufige Lösung: minimal möglichen Wert des Typs T.
(Achtung, `null` hier formal nicht möglich!)

Dieser Algorithmus kann natürlich auch programmiert werden, ohne den Minimum-Wert zu verwenden. Allerdings muss man sich dann eine besondere Behandlung für `numItems == 0` ausdenken.

Weiteres Beispiel für Type Traits


```
#include <limits>
```

- Generischer Algorithmus, der das größte Element in einem Array findet

```
template< class T >
T findMax(const T* data, size_t numItems) {
    T largest = numeric_limits<T>::min();
    for(size_t i=0; i<numItems; ++i)
        if (data[i] > largest)
            largest = data[i];
    return largest;
}
```

Hier kann auf generische Weise ein „Trait“ eingesetzt werden.

Weiteres Beispiel für Type Traits

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Analoges Beispiel:
  Spezialisierung einer Traits-Klasse für `unsigned int`

```
template<>
class NumTraits<unsigned int> {
  public:
    static const char* name() { return "unsigned int"; }
    static bool hasSign() { return false; }
    static unsigned int minValue() { return 0u; }
    static unsigned int maxValue() { return UINT_MAX; }
};
```

Spezialisierung der Traits

Type Traits und Generische Programmierung

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

[...] by encapsulating those ***properties that need to be considered on a type by type basis*** inside a traits class, we can minimise the amount of code that has to differ from one type to another, and ***maximise the amount of generic code***.

John Maddock and Steve Cleary*

*) http://www.boost.org/doc/libs/1_32_0/libs/type_traits/cxx_type_traits.htm