

# Programmierkurs C++ für Fortgeschrittene

SS 2011

Teil 1

Sandro Andreotti - [andreott@inf.fu-berlin.de](mailto:andreott@inf.fu-berlin.de)

# **THEMA 1**

## **EINFÜHRUNG**

# Organisation

- Homepage:

[http://www.inf.fu-berlin.de/en/groups/abi/lectures\\_current/K\\_CPP\\_Advanced/](http://www.inf.fu-berlin.de/en/groups/abi/lectures_current/K_CPP_Advanced/)

- Vormittags: Seminar

Hörsaal B (Arnimallee 22)

Anwesenheitspflicht!

- Nachmittags: Übungen am Rechner

Raum K 46/48 (Takustr. 9)

Abteilung A: 13.00 – 15.00 Uhr

Abteilung B: 15.00 – 17.00 Uhr

Die Anwesenheitspflicht beim Seminar gilt für diejenigen Teilnehmer, die einen Schein erwerben wollen.

Für die Nachmittagsübungen gilt keine Anwesenheitspflicht; d.h. die Teilnehmer dürfen sie gerne auch zu Hause bearbeiten.

Einzige Ausnahme ist der Freitag, an dem man eine der Aufgaben vorführen muss, wenn man einen Schein erwerben möchte.

# Scheinvergabe

- **benoteter Schein:**
  - 2 SWS = 2 LP (ECTS)
  - "Allgemeine Berufsvorbereitung"
- **Voraussetzungen:**
  1. Teilnahme am Seminar (Unterschrift!)
- **Note setzt sich zusammen aus:**
  1. Vorführung einer „Prüfungsaufgabe“ am Freitag in der Übung.
  2. Kurztest im Seminar am Freitag

An jedem Tag von Montag bis Donnerstag erhalten Sie eine „Prüfungsaufgabe“, die Sie unbedingt bearbeiten sollten. Am Freitag müssen Sie eine davon am Rechner vorführen, welche entscheidet der Dozent.

Die Vorführung kann in Gruppen von bis zu 3 Personen erfolgen; jeder, der einen Schein bekommen möchte, muss daran teilnehmen. Jeder Teilnehmer muss dazu befähigt sein, Fragen zum erstellten Programm zu beantworten.

Besteht berechtigter Zweifel daran, dass einer der Teilnehmer nicht aktiv an der Programmierung der vorgeführten Aufgabe teilgenommen hat, kann der Teilnahmechein verweigert werden.

# Vorläufiger Plan

- **Montag:** Einführung, Templates
- **Dienstag:** Namespaces, Typkonversion, Funktionsüberladung
- **Mittwoch:** Exceptions, ???
- **Donnerstag:** Speicherverwaltung, Virtual
- **Freitag:** STL, Trickkiste

# Literaturhinweise...

... auf der Veranstaltungshomepage.

ISO/IEC 14882

"Programming Languages – C++"  
(1998)

## Literatur

### **"Programming Languages – C++", ISO/IEC 14882, 1998**

Stroustrup, "Die C++ Programmiersprache" (2000), ISBN 3-8273-1660-X, € 49,95

Lippman, Lajoie, "C++" (2002), ISBN 3-8266-1429-1, € 25,00

Josuttis, "The C++ Standard Library - A Tutorial and Reference" (2002), ISBN 0201379260, c.a. € 55,00

Meyers, "Effektiv C++ programmieren" (1997), ISBN 3-8273-1305-8, € 29,95

Meyers, "Mehr Effektiv C++ programmieren" (1997), ISBN 3-8273-1275-2, € 29,95

- Eine Art "Tipps & Tricks"-Fundgrube

Kernighan, Ritchie, "Programmieren in C" (1990), ISBN 3-446-15497-3, € 32,90

- Der Klassiker zu C - **nicht** zu C++

Links finden sich auf der Veranstaltungshomepage (<http://www.inf.fu-berlin.de/en/groups/abi>)

# Zur Einstimmung: Quiz 1

Frage: Was gibt dieses Programm aus?

```
int i = 10;
for (i = 0; i < 4; ++i);
{
    switch (i)
    {
        case 0: std::cout << 1;
        default: std::cout << i;
    }
}
```

Die Antwort: 4

Grund: Das ; am Ende der for-Anweisung.

Wäre das ; nicht dort, so würde das Programm folgendes ausgeben: 10123

Diese Quiz-Fragen dienen dazu, den Kenntnisstand der Hörer zu testen. Es ist nicht unbedingt notwendig, alle Fragen richtig beantworten zu können, um an der Vorlesung teilnehmen zu können.

## Quiz 2

Was ist folgende Variable und wieviel Speicher benötigt sie?

```
char const * Arr [10];
```

**Faustregel: „Deklarationen liest man von innen nach außen“:**

Lies: „Arr ist ein Array von Pointern auf konstante chars“

Der Speicherbedarf von Arr beträgt auf 32-bit Systemen **sizeof(Arr) == 40** bzw. **80** auf 64-bit Systemen.

kein Array konstanter Pointer auf chars, das wäre

```
char * const Arr [10];
```

kein Pointer auf ein Array von konstanten chars, das wäre:

```
char const (* Arr) [10];
```



## Quiz 3

Die folgende Zeile wurde einem korrekten C++-Programm entnommen.

Frage: Welche Typen haben X, Y und Z?

```
int Z = X *** Y;
```

(Es gibt alternative Antworten)

Antwort:

**int X;**

**int \*\* Y;**

Das erste \* in **X\*\*\*Y** ist die Multiplikation, die beiden anderen \* sind jeweils ein indirection-Operator (Dereferenzierungsoperator).

Alternative Antwort: **X** oder **Y** (oder beide) können Klassen sein, bei denen die Operatoren auf geeignete Weise überladen wurden.

**Z** ist auf jeden Fall **int**.

# **THEMA 2**

## **TEMPLATES**

# Templates: Motivation

Aufgabe: Schreibe eine Funktion `max(a, b)`, die das Maximum zweier Zahlen `a` und `b` ausgibt:

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

Problem: Funktion wird für alle Typen benötigt

```
float x = max(1.4, y);
```

Eine Alternative wäre die Verwendung von Preprocessor-Macros, doch dann müsste man auch alle bekannten Nachteile von Macros hinnehmen.

# Lösung: Templates

Templates sind Schablonen, nach denen der Compiler Code herstellt.

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

...

float x = 1, y = 2;
float z = max (x, y);
```

Nach der template-Definition kann die Funktion **max** mit beliebigen Typen verwendet werden. Wichtig dabei ist aber, dass die Argumente beide denselben Typ haben.

**T** nennen wir einen Template-Parameter oder auch Template-Argument. Statt **typename T** kann man auch **class T** schreiben – eine irreführende Notation, da es sich bei **T** (wie man im Beispiel sieht) ja nicht notwendigerweise um eine Klasse handeln muss.

# Template-Argumente

Es gibt zwei Möglichkeiten, die Template-Argumente zu bestimmen:

## 1. Explizit:

```
template <typename T1, typename T2>
void f (T1 a, T2 b) { ... }

f<int, float>(0, 3.5);
```

## 2. Implizit:

```
int x = 0; float y = 3.5;
f(x, y);
```

Implizite Argumentbestimmung funktioniert selbstverständlich nur bei Template-Funktionen, nicht bei Template-Klassen (siehe unten).

## Template-Argumente, gemischt

Mischung von expliziter und impliziter Bestimmung von Template-Argumenten:

```
template
  <typename T1, typename T2, typename T3>
void f (T3 x, T2 y, T1 z)
{ ... }

f<int>( (char) 0, 3.5F, 7 );
```

1. Argument: `char x`
2. Argument: `float y`
3. Argument: `int z`

Die Festlegung des Typs von `x` erfolgt über einen Cast nach `char`.

Suffix `F` in `3.5F` zeigt an, dass es sich bei der Konstante um ein `float` handelt.

Der Typ von `z` wird durch explizite Bestimmung festgelegt. Da nur ein Typ explizit angegeben wurde, nimmt der Compiler dies als erstes Template-Argument (d.h. für `T1`) an.

Wie man sieht muss die Reihenfolge der Template-Argumente nicht mit der Reihenfolge ihrer Benutzung in der Liste der Funktions-Argumente übereinstimmen.

# Implizit: Problem 1

Folgender Code verursacht ein Problem:

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

double x = 1.0;
double y = max(x, 0);
```

```
error: template parameter 'T' is ambiguous
```

Das Problem ist, dass die 0 im Aufruf von **max** keine Festlegung des Types erlaubt: 0 kann ein **double** sein, aber z.B. auch ein **int** oder ein **char**.

Abhilfe könnte man z.B. so schaffen:

```
double y = max(x, 0.0);
```

Insgesamt gilt: Der Compiler ist recht „pingelig“, was die Übereinstimmung der Typen bei Template-Parametern anbelangt: Wenn deklariert wurde, dass zwei Parameter einer Funktion vom gleichen Typ T seien, dann müssen die Argumente beim Aufruf der Funktion auch tatsächlich vom selben Typ sein.

## Implizit: Problem 2

Bei folgendem Beispiel funktioniert eine implizite Bestimmung des Template-Arguments gar nicht:

```
template <typename T>
T zero ()
{
    return 0;
}

int x = zero();
```

**error: could not deduce template argument**

Das gleiche Problem gab es bei polymorphen Funktionen: Auch hier durften 2 Funktionen sich nicht nur durch ihre Rückgabetypen unterscheiden. Der Typ einer Rückgabe kann vom Compiler generell nicht aus dem Aufruf der Funktion allein ermittelt werden.



# Parameter-Deklarationen

Statt eines Types kann auch eine **Konstante** als Template-Parameter spezifiziert werden:

```
template <int I>
void print ()
{
    std::cout << I;
}

print<5>();
```

Ausgabe: 5

Die Konstante muss explizit übergeben werden. Der Wert muss eine zur Compile-Zeit bekannte Konstante sein. Es sind nur aufzählbare Typen für derartige Konstanten erlaubt (z.B. **int**, aber nicht **double**).

# Template-Klassen

Wie Template-Funktionen lassen sich auch **Template-Klassen** definieren:

```
template <typename T>
struct Pair
{
    T element1;
    T element2;
};

Pair <int> p;
```

Bei Template-Klassen müssen die Argumente immer explizit angegeben werden.

## Defaults für Template-Parameter

Für Template-Klassen können Default-Argumente definiert werden.

```
template <typename T = int>
struct Pair
{
    T element1;
    T element2;
};

Pair < > p;
```

(nur für Template-Klassen)

Nachfolgende Parameter müssen dann auch Defaults haben.

Die Angabe von Defaults funktioniert nur bei Template-Klassen, nicht bei Template-Funktionen.

Auf die leeren spitzen Klammern < > kann bei der Benutzung des Templates nicht verzichtet werden.

Defaults können auch bei Parameter-Deklarationen spezifiziert werden:

```
template <int I = 5>
struct FixArray
{
    int array_member[I];
};

FixArray<> f;
```

# Template Beispiele

**Frage:** Wozu kann man die Argumente von Template-Klassen verwenden?

Beispiel 1: Typen für Member

```
template <typename T>
struct Pair
{
    typedef T MyType;
    T element1;
    T element2;
    T get_max();
    void set_both(T elm1, T elm2);
};
```

# typename

Hinweis für den Compiler: „hier kommt ein Type!“

```
template <typename T>
struct A
{
    typename T::Alphabet x;
    typedef int MyInteger;
};
```

**typename** steht vor Typen, ...

1. die in Template-Klassen definiert sind
2. die von Template-Argumenten abhängen

Siehe auch: ISO/IEC 14882, 14.6 „Name resolution“

## "Patterns" von Template Typen

Typen von Klassentemplates können als eine Art "Pattern" angegeben werden:

```
template <typename T1, typename T2>
struct MyClass;

template <typename T>
void function (T & obj);

template <typename T1, typename T2>
void function (MyClass<T1, T2> & obj);

template <typename T>
void function (MyClass<T, int> & obj);
```

Die dritte Funktion kann offensichtlich nicht für alle Instanzierungen des Templates **MyClass** verwendet werden, sondern nur für solche, die **int** als zweites Templateargument besitzen.

# Template Spezialisierung

Templates können für bestimmte Template-Argumente spezialisiert werden.

```
template <typename T>
struct MyClass
{ int foo; };

template < >
struct MyClass <int>
{ int x; };

MyClass<int> obj;
obj.x = 17;

obj.foo = 18; //error!
```

foo ist in MyClass<int>  
nicht bekannt.

Auch function templates lassen sich (total) spezialisieren. Davon ist aber eher abzuraten (siehe <http://www.gotw.ca/publications/mill17.htm>), denn bei Funktionen stehen uns schließlich die viel mächtigeren Möglichkeiten des function overloadings zur Verfügung.

Statt also ein Template und eine Spezialisierung zu schreiben, schreiben wir besser zwei gleich function templates. Beispiele dazu gibt es z.B. im Teil 3 des Kurses auf der Seite "Der beste Kandidat (3)".

## Quiz 4: Was wird ausgegeben?

```
template <typename T> struct C
{
    static void f() { std::cout << "allgemein"; }
};

template < > struct C <char*>
{
    static void f() { std::cout << "speziell"; }
};

template <typename T> void call_f(T & t)
{
    C<T>::f();
}

call_f("hallo");
```

Antwort: „allgemein“.

Grund: das String-Literal "hallo" ist nicht vom Typ **char \*** sondern vom Typ **char const [6]**, und somit müsste C genau für diesen Typ spezialisiert werden.



## Quiz 4a: Und jetzt?

```
...  
  
typedef char * cptr;  
cptr x;  
call_f(x);
```

Antwort: „speziell“

Durch **typedef** wird kein neuer Typ erzeugt, sondern nur eine Art „alias“ für einen anderen Typ.

# Partielle Spezialisierung

```
template <typename T, unsigned int I>
class Array
{
    T arr [I];
};

template <typename T>
class Array <T, 0>
{
};
```

Nur Template-Klassen können partiell spezialisiert werden. Bei Template-Funktionen gibt es keine partielle Spezialisierung.

# Metafunctions

(Eine Anwendung für Template Spezialisierung)

**Beispiel:** Eine Funktion auf Strings

```
template <typename TString>
char first_character(TString & string)
{
    return string[0];
}
```

**Problem:** Was machen wir, wenn der Alphabettyp von **TString** nicht **char** ist?

Wir brauchen eine Möglichkeit, den "value type" zu einem gegebenen String Typen festzustellen.

# Metafunctions

Idee: Eine Art "Funktion", die Typen zurückliefert

Pseudo-Code:

```
template <typename TString>  
Value(TString) first_character(TString & string)  
{  
    return string[0];  
}
```

Vorsicht: Der angegebene Code ist natürlich kein gültiges C++.

# Metafunctions

**Lösung:** Verwende Klassen-Templates

```
template <typename T>
struct Value
{
    typedef char Type;
};

template <typename TString>
typename Value<TString>::Type
first_character(TString & string)
{
    return string[0];
}
```

... dieser schon.

# Metafunctions

Spezialisiere Template für verschiedene Typen:

```
template <typename TChar,  
         typename TTraits,  
         typename TAlloc>  
struct Value < basic_string<TChar, TTraits, TAlloc> >  
{  
    typedef TChar Type;  
};  
  
template <>  
struct Value <char *>  
{  
    typedef char Type;  
};
```

Wir definieren **Value** auch für STL Strings und C-style Strings .

## „rekursive“ Templates

Bsp.:

```
template <int I>
struct Tupel: Tupel <I-1>
{
};

template < >
struct Tupel<1>
{
};
```

Die eben vorgestellten Techniken gehören zum sogenannten "Template Metaprogramming".

Der Ausdruck "Metaprogramming" erklärt sich daher, dass man sich Templates als eine Art Programmiersprache vorstellen kann, die vom Compiler interpretiert wird, um C++-Code zu erzeugen.

Aufgabe 4 auf der nächsten Seite gibt ein Beispiel für diese Art der Programmierung.

## Aufgaben zum Dienstag:

### 1. Aufgabe: (PRÜFUNGSAUFGABE)

a) Schreiben Sie Funktionen **length(str)** und **charAt(str, i)**, mit denen man die Länge bzw. das i-te Zeichen eines Strings **str** ermitteln kann, und zwar sowohl für Strings der Standard-Bibliothek (**basic\_string**) als auch für 0-terminierte **char** arrays ("c-style Strings").

b) Schreiben Sie ein einzelnes Funktions-Template **printRev(str)**, das unter Verwendung von **length(str)** und **charAt(str, i)** einen String **str** rückwärts ausgibt.

### 2. Aufgabe:

Schreiben Sie ein Klassen-Template **Tuple**, das **N** Objekte des Typs **T** speichert. Dabei sollen **N** und **T** als Template-Argumente festgelegt sein. Definieren Sie Member-Funktionen, mit denen man auf die einzelnen Objekte innerhalb eines Tuple-Objekts zugreifen kann. Testen Sie ihre Klasse in einem Projekt mit mindestens zwei *translation units* (d.h. mit mindestens zwei cpp-Dateien).

### 3. Aufgabe:

Untersuchen Sie folgende Fragen, indem Sie kurze Testprogramme schreiben:

a) Können Konstruktoren von Klassen auch Funktions-Templates sein? Und wie ist es mit Konstruktoren von Klassen-Templates?

b) Können Klassen-Templates auch statische Datenmember enthalten, und wie kann man diese initialisieren?

### 4. Aufgabe: (schwierig!)

Schreiben Sie ein Klassen-Template **template <int I> struct Fib**, mit dessen Hilfe *zur Compilezeit* Fibonacci-Zahlen berechnet werden können, d.h. man bekommt z.B. mit **Fib<6>::Value** den Wert 8 geliefert.