

# **Proseminar Programmiersprachen**

## **C++ Templates**

Johanna Witte

8. November 2006

# Inhaltsverzeichnis

0.1	C++	3
0.1.1	Wo findet C++ Verwendung	3
0.1.2	Was unterstützt C++ im Gegensatz zu anderen Programmiersprachen	3
0.1.3	Generische Programmierung	4
0.2	Templates	4
0.2.1	Was sind Templates?	4
0.2.2	Motivation: Wozu benutzt man Templates?	4
0.2.3	Wie definiert man ein Template?	5
0.2.4	Welche Arten von Templates gibt es	5
0.2.5	Überladen von Templates	5
0.3	STL (Standard Template Library)	6
0.4	Spezialisierung von Templates	6
0.4.1	Explizite Spezialisierung	7
0.4.2	Partielle Spezialisierung	7
0.4.3	Folgerung	7
0.5	Code Organisation	8
0.5.1	Übersetzung	8
0.5.2	Inclusion Model	8
0.5.3	Explicit Instantiation Model	8
0.5.4	Separation Model	8
0.5.5	Template Instanziierung	8
0.5.6	Code Organisation ohne Templates	9
0.5.7	Code Organisation mit Templates	10
0.5.8	Wo soll die Implementierung untergebracht werden?	10
0.6	Fortgeschrittene Templates	11
0.6.1	Template Metaprogramme	11
0.6.2	Introspection	12
0.7	Folgerung	13

Zitat Bjarne Stroustrup :

„The most fundamental problem in software development is complexity. There is only one basic way of dealing with complexity : Divide and conquer.“

## 0.1 C++

C++ ging aus der Programmiersprache C hervor und wurde von dem Dänen Bjarne Stroustrup, Professor der Informatik an der A&M University in Texas, entwickelt. Während ihrer Entstehung hieß die Sprache zuerst “C with Classes“ und erhielt dann, erstmals im Dezember 1983, den Namen C++, welcher aus dem Inkrement-Operator entstand.

C++ enthält C fast vollständig und bietet zusätzlich zu den in C vorhandenen Möglichkeiten weitere Datentypen, Klassen mit Vererbung und virtuellen Funktionen, Ausnahmebehandlung, Templates, Namensräume, Inline-Funktionen, Überladen von Operatoren und Funktionsnamen, Referenzen, Operatoren zur Freispeicherverwaltung und mit der C++ Standardbibliothek eine erweiterte Bibliothek.

Die Kompatibilität mit C war eines der Hauptdesignziele bei der Entwicklung von C++, das sich aus der bereits weiten Verbreitung von C ergab.

Der Vergleich von C++ mit Java und C# ist nahe liegend, da sie eine ähnliche, ebenfalls an C angelehnte Syntax haben, auch objektorientiert sind und Typparameter unterstützen.

Im Moment wird an einer Weiterentwicklung der Sprache C++ (welche inoffiziell den Namen C++0x trägt) gearbeitet. Diese Weiterentwicklung soll schätzungsweise bis spätestens 2009 fertig gestellt werden.

### 0.1.1 Wo findet C++ Verwendung

C++ ist eine der am weitesten verbreiteten und industriell bedeutendsten Programmiersprachen. Sie ist eine sehr vielseitige und allgemein verwendbare Sprache, welche als Mehrzwecksprache konzipiert ist und insbesondere effiziente und maschinennahe Programmierung, Datenabstraktion sowie objektorientierte und generische Programmierung unterstützt.

Somit existieren C++ Implementierungen für einige der kleinsten Mikrocomputer bis hin zu den größten Supercomputern und für nahezu alle Betriebssysteme.

Ihr Hauptanwendungsgebiet ist die Systemprogrammierung im weitesten Sinne.

### 0.1.2 Was unterstützt C++ im Gegensatz zu anderen Programmiersprachen

C++ ist eine so genannte Multiparadigmen-Sprache, die dem Programmierer sehr viele Freiheiten lässt. Sie dient als Sprache für Datenabstraktion und unterstützt die objektorientierte, sowie die generische, die prozedurale, die modulare und die strukturierte Programmierung.

Durch die Objektorientierung wird die Flexibilität und die Wiederverwendbarkeit von Programmen gefördert.

### Besonderheiten von C++

C++ ist eine sehr vielfältige Programmiersprache, mittels derer es möglich ist hocheffizienten Code zu erzeugen. Außerdem ist sie gut für große Projekte geeignet.

Die Sprache ist sehr flexibel und hat eine hohe Ausdrucksstärke, ein Beispiel dafür ist die anpassbare Freispeicherverwaltung, in die sich eine automatische Speicherbereinigung problemlos integrieren lässt. Ein weiterer großer Vorteil ist die Kompatibilität mit C, da hiermit eine breite Codebasis zur Verfügung steht und die aktuellen Frameworks für C++ teilweise plattformübergreifend und weit entwickelt sind.

Jedoch ergeben sich auch einige Nachteile, wie zum Beispiel, dass aufgrund der Kompatibilität mit C einige Details der Sprache Compiler spezifisch sind, was sie aber nicht sein müssten. So ist beispielsweise die Auswertungsreihenfolge von Teilausdrücken je nach Compiler und Plattform unterschiedlich. Zusätzlich sind die aktuellen Compiler unter Umständen bezüglich der Umsetzung der ISO-Norm rückständig.

Die Einhaltung von Programmierrichtlinien ist in C++ besonders wichtig, da sich ansonsten aufgrund des breiten Leistungsspektrums und der vielfältigen Gestaltungsmöglichkeiten im Bezug auf die Wartbarkeit und die Fehleridentifizierung ernsthafte Probleme ergeben können. Zudem ist C++ eine sehr mächtige und deshalb auch nicht sehr leicht zu handhabende Programmiersprache.

### **0.1.3 Generische Programmierung**

Die generische Programmierung ist ein Programmierparadigma, bei dem einzelne Funktionen und Klassen immer möglichst allgemein geschrieben werden, so dass sie für unterschiedliche Typen verwendet werden können.

Die generische Programmierung wird von C++ durch die Verwendung von Templates unterstützt.

Wesentlich bei der generischen Programmierung ist, dass die Algorithmen nicht für einen bestimmten Datentyp geschrieben werden, sondern nur bestimmte Anforderungen an die Typen stellen. Das Prinzip wird auch parametrische Polymorphie genannt.

Paradebeispiel ist die C++ Standardbibliothek, bei der die Algorithmen so weit wie möglich von Datenstrukturen, mit denen sie arbeiten, getrennt werden.

## **0.2 Templates**

### **0.2.1 Was sind Templates?**

Das Wort Template heißt übersetzt Schablone und ist im Falle von C++ wörtlich zu nehmen. Templates sind also Programmgerüste, die eine vom Datentyp unabhängige Programmierung erlauben. Somit unterstützen Templates die generische Programmierung.

Sie wurden Anfang der 90er Jahre in C++ eingeführt. Seitdem erschließen sich immer wieder neue Anwendungsgebiete für Templates.

Gerade in C++ sind Templates von großem Nutzen, insbesondere wenn sie mit Vererbung und Überladen von Operatoren kombiniert werden.

In anderen Programmiersprachen gibt es zu Templates äquivalente Konzepte, die die generische Programmierung unterstützen, wie zum Beispiel Generics in Java 5.

### **0.2.2 Motivation: Wozu benutzt man Templates?**

Anstatt zum Beispiel eine Liste einmal für `int` und einmal für `String` zu schreiben, kann man diese mit Hilfe von Templates verallgemeinern und muss somit die Liste nur einmal schreiben. Welche Typen nun später in der Liste enthalten sind, ist dann irrelevant. Neben Typen können auch konstante Ausdrücke als Templateparameter benutzt werden.

### 0.2.3 Wie definiert man ein Template?

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

Die `template<...>` Zeile sagt dem Compiler, dass die folgende Definition oder Deklaration als Schablone zu behandeln ist.

Ein Template erwartet als Argumente Platzhalter, in unserem Beispiel T, für Typen. Diese Platzhalter sind die Templateparameter.

Später, noch bevor der Code in Maschinensprache übersetzt wird, werden die Platzhalter innerhalb des Templates durch spezifische Typen ersetzt werden.

Templates sind folglich dazu da, den Aufwand für den Entwickler zu reduzieren.

Also werden Templates vom Compiler nach Bedarf in eine normale Funktion oder Klasse umgewandelt, beim kompilieren findet auch die Typprüfung statt.

Wird in unserem Beispiel nun `swap` für zwei `int` Werte aufgerufen, so wird eine weitere Funktion erzeugt. Dasselbe gilt, wenn `swap` für zwei `Auto` Werte aufgerufen wird.

### 0.2.4 Welche Arten von Templates gibt es

#### Funktionstemplates

Ein Funktions-Template verhält sich wie eine Funktion die in der Lage ist, Argumente verschiedener Typen zu behandeln.

Anhand der Argumente der Funktion macht der Compiler fest, dass es sich beispielsweise um einen Aufruf von `int` handelt und erzeugt eine Variante der Funktion, bei der er Typ T zu `int` festgelegt wird.

#### Klassentemplates

Ein Klassen-Template wendet das gleiche Prinzip auf Klassen an.

Klassen-Templates finden oft Verwendung in der Erstellung von generischen Containern. So enthält die C++ Standardbibliothek einen Container, der eine verkettete Liste implementiert.

Mit `list<int>` kann man eine verkettete Liste von `int` erstellen. Um diese nun für `String` zu erstellen schreibt man `list<string>`.

Mit `list` ist ein Satz von Standardfunktionen definiert, die immer verfügbar sind, egal was man als Argumenttyp in spitzen Klammern angibt.

### 0.2.5 Überladen von Templates

Es ist möglich mehrere Funktions-Templates mit demselben Namen zu deklarieren, man kann sogar eine Kombination aus Funktions-Templates und normalen Funktionen mit demselben Namen deklarieren. Um dann für den Aufruf die richtige Funktion, beziehungsweise die richtige Template-Funktion zu finden, muss diese Überladung aufgelöst werden.

Der Vorgang des Auflöser von Überladungen funktioniert folgendermaßen:

Zuerst muss man für jedes Template diejenige Spezialisierung finden, die am besten zu den Funktionsargumenten passt, dann werden die üblichen Kriterien zum Auflösen von überladenen Funktionen überprüft und angewendet.

Man muss zuerst alle an der Überladung beteiligten Spezialisierungen ausfindig machen, indem man alle

Funktions-Templates betrachtet und entscheidet, welche Templateargumente überhaupt benutzt würden, falls es keine Überladungen im Gültigkeitsbereich gäbe. Wenn nun zwei Template-Funktionen aufgerufen werden könnten, wählt man zwischen diesen beiden die Spezialisiertere aus. Dann löst man die entstandenen Überladungen auf, indem man die Kriterien zum Auflösen von überladenen Funktionsnamen durchgeht. Diese sind: Genaue Übereinstimmung beziehungsweise Übereinstimmung ohne oder nur mit trivialen Konvertierungen, Übereinstimmung mit Promotionen, Übereinstimmung mit Standardkonvertierungen, Übereinstimmung mit benutzerdefinierten Konvertierungen und zuletzt Übereinstimmung mit ... (=unspezifizierte Anzahl an Argumenten) in einer Funktionsdeklaration. Wenn mehrere Übereinstimmungen auf der höchsten Stufe gefunden werden, wird der Aufruf als mehrdeutig abgelehnt.

Wenn ein Argument einer Template-Funktion über eine Templateargumentermittlung bestimmt wurde, kann auf dieses Argument keine Promotion, Standardkonvertierung oder benutzerdefinierte Konvertierung mehr angewendet werden.

Eine Funktion wird dann bevorzugt, falls diese Funktion und eine Spezialisierung eine gleich gute Übereinstimmung aufweisen.

Der Funktionsaufruf ist ein Fehler, falls gar keine Übereinstimmungen oder aber mehr als eine gleich gute Übereinstimmung gefunden wurde, da der Aufruf in diesem Fall mehrdeutig ist.

### 0.3 STL (Standard Template Library)

Der Entwurf der C++ Standardbibliothek ist relativ spät um einen sehr bedeutenden Teil erweitert worden, der zunächst STL genannt wurde, weil er unter anderem sehr viele Templates zur Verfügung stellt. Da die STL in der Standardbibliothek integriert ist gibt es keine exakten Grenzen zwischen diesen Bibliotheken mehr. Wichtig ist nur, dass die Komponenten, insbesondere die zur Verfügung gestellten Templates, sehr mächtige Werkzeuge der Programmierung darstellen und deswegen bekannt sind und genutzt werden.

Die STL erhöht die Einsetzbarkeit von C++ enorm, da immer wiederkehrende Templates dort definiert sind. Somit wird eine Vielzahl von Implementierungen dadurch unnötig. Die wesentlichen Komponenten der STL sind:

- Algorithmen, zum Beispiel Sortieren,
- Container, zum Beispiel Listen,
- Iteratoren, mit deren Hilfe unter anderem Container elegant übergeben werden können,
- Funktionsobjekte, die eine Funktion kapseln, die von anderen Komponenten genutzt werden kann, und
- Adaptoren, durch die eine andere Schnittstelle von einer Komponente gebildet werden kann.

### 0.4 Spezialisierung von Templates

Das Ersetzen von Typen in Templates reicht alleine aber noch nicht zum Ausführen von beliebigen Funktionen. Hierzu wird noch die Technik der Spezialisierung benötigt.

Diese erlaubt eine effiziente Implementierung für bestimmte ausgewählte Datentypen, ohne die Schnittstelle des Templates zu verändern. Von dieser Technik machen viele Implementierungen der C++ Standardbibliothek Gebrauch.

Unter der Spezialisierung eines Templates versteht man die Möglichkeit, eine Klasse für einzelne Typen zu spezialisieren. Beispielsweise möchte man, dass die Vergleichsfunktion einer Klasse für den Typ `int` anders funktioniert als sie es für den Typ `String` tut.

### 0.4.1 Explizite Spezialisierung

Im Gegensatz zu Klassen-Templates können Funktions-Templates nur vollständig, also explizit, spezialisiert werden. Von der Spezialisierung von Funktions-Templates wird aber allgemein abgeraten! Ein Beispiel zur expliziten Spezialisierung:

Standard Implementierung:

```
template <class T, int Size>
void MyVector::multiply (int d) {
for (int i=0; i<Size; ++i) {
data[i] *= T(d);
}
};
```

Spezialisierte Implementierung:

```
template <>
void MyVector<double,2>::multiply (double d) {
data[0] *= d;
data[1] *=d;
};
```

Das Präfix `template<>` besagt, dass dies eine Spezialisierung ist, die ohne Templateparameter spezifiziert werden kann. Die Templateargumente, für die die Spezifizierung genutzt werden soll, werden in den `<>`-Klammern angegeben.

### 0.4.2 Partielle Spezialisierung

Falls man jedoch eine teilweise Spezialisierung von Funktions-Templates benötigt, kann man dies in den meisten Fällen, durch Überladen von Funktions-Templates mit anderen Funktions-Templates erreichen. Sollte dies unter Umständen in einem konkreten Fall nicht möglich sein, kann man das Problem der Spezialisierung auf ein Template einer Hilfsklasse verlagern.

Des Weiteren gibt es auch die so genannte teilweise oder partielle Spezialisierung, die die Behandlung von Spezialfällen innerhalb eines Klassen-Templates ermöglicht. Ein Beispiel für eine partielle Spezialisierung wäre:

```
template <class T>
void MyVector<T,2>::multiply (double d) {...};
```

### 0.4.3 Folgerung

Folglich ermöglichen Templates analogen Code nicht mehrfach wiederholen zu müssen. Die Parameter müssen zur Kompilierungszeit bekannt sein. Der Compiler generiert nach Bedarf die nötigen Klassen/Funktionen anhand der Templates. Es entsteht nicht weniger oder mehr kompilierter Programmcode, als von Hand programmiert. Außerdem verzeichnet man einen Geschwindigkeitsgewinn durch spezialisierte Algorithmen.

## 0.5 Code Organisation

### 0.5.1 Übersetzung

Vor der Übersetzung eines C++ Programms, startet ein Präprozessor. Dieser manipuliert das Programm, bevor der Compiler es zu sehen bekommt.

Der Präprozessor hat mehrere Aufgaben. Zu diesen gehören unter anderem die Makrosubstitution und das Einfügen von Headerdateien. In C++ wird zwischen dem Kompilieren und dem Binden eines Programms unterschieden, ganz im Gegensatz zu Java.

Ein Programm besteht also aus Übersetzungseinheiten. Die Aufgabe des Binders (Linkers) ist es, diese Einheiten zu einer ausführbaren Einheit zusammen zufügen.

Beim kompilieren findet eine Typprüfung statt. Fehler die durch Templateparameter entstehen, können nicht vor der ersten Benutzung erkannt werden!

Normalerweise trennen C++ Programmierer die Schnittstellendefinition (Deklaration, \*.h Datei) von der Implementierung (\*.cc Datei). Dies ruft bei Templates jedoch Probleme hervor, zum Beispiel können bestimmte Instanzen noch unbekannt sein oder es gibt andere Übersetzungseinheiten und somit keinen Quelltext des Templates.

### 0.5.2 Inclusion Model

Die heute gängigste Lösung für dieses Problem ist das Inclusion Model, in dem die Implementierung in der \*.h Datei stattfindet. Wenn zwei Übersetzungseinheiten die gleiche Templateinstanz benötigen, werden Duplikate dadurch vermieden, dass Template- und Memberfunktionen von Template-Klassen speziell gebunden werden. Durch diesen Binder werden Duplikate vermieden.

### 0.5.3 Explicit Instantiation Model

Diese Model nutzt die explizite Instanziierung. Diese wird verwendet, wenn zum Beispiel der Code des Templates einer Bibliothek nicht offen gelegt werden soll.

### 0.5.4 Separation Model

Beim Separation Model wird durch Export ein separat implementiertes Template gekennzeichnet. Das Separation Model wird bisher von fast keinem Übersetzer unterstützt, wie jedoch die Übersetzer in der Zukunft damit umgehen werden ist noch unklar.

### 0.5.5 Template Instanziierung

Als Template Instanziierung bezeichnet man die Umwandlung eines Templates in eine konkrete, übersetzbare Einheit.

Das Template wird dabei für einen bestimmten Typ (den Templateparameter) spezialisiert.

#### Implizite Template Instanziierung

Wenn alle Parameter eindeutig aus dem Programmkontext hervorgehen, kann der Compiler Templates automatisch spezialisieren:

```
d1 = 5.0, d2 = 3.0;
swap(d1, d2);
```

Dies ist möglich, da T vom Typ double sein muss.

Ein weiteres Beispiel wäre die Berechnung vom Minimum von zwei Werten:

```
template <class T>
T min (T a, T b) {
return ((a<b) ? a : b);
};
```

```
double d1 =5.0, d2 = 3.0; // wird vom Compiler
double m = min (d1, d2); //akzeptiert, weil T = double
```

```
double d1 = 5.0;
float f1 = 10.0f;
double m = min(d1, f1); // Fehlermeldung: no matching function for call to 'min(double&, float&)'
```

Keine Fehlermeldung gäbe es bei:

```
double m = min<double>(d1, f1);
```

Oder bei:

```
double m = min(d1, double(f1));
```

### Explizite Template Instanziierung

Explizit bedeutet hierbei, dass der Programmierer ausdrücklich spezifiziert, mit welchem Typ die Templateparameter zu ersetzen sind.

Wir kommen nun zu unserem oben genannten Beispiel zurück:

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

Die Syntax für die explizite Spezialisierung ist in unserem Fall:

```
double d1 = 5.0, d2 = 3.0;

swap<double>(d1, d2);
```

### 0.5.6 Code Organisation ohne Templates

Bei der Code Organisation ohne Templates werden Interface und Implementierung in separaten Dateien untergebracht (.h, .cpp). Außerdem stellt der Linker die Verbindung zwischen dem Aufruf einer

deklarierten Methode und dem zugehörigen Maschinencode her.

## 0.5.7 Code Organisation mit Templates

Mit der Benutzung von Templates wird der Code erst bei Bedarf (durch Bindung an spezifische Templateparameter) generiert, dabei kann nur der vom Template generierte Code übersetzt und gelinkt werden.

## 0.5.8 Wo soll die Implementierung untergebracht werden?

Eine Möglichkeit wäre es, die Template Instanziierung für eine feste Menge von Typen zu erzwingen. Dies sähe so aus:

```
myLib.h
//Deklaration
template<class T>
T min (T a, T b);

myLib.cpp
#include 'myLib.h'
template<class T> //Implementierung
T min (T a, T b); {
return ((a<b) ? a : b);
};
template float min<float>(float, float);
```

Eine weitere Möglichkeit wäre es, den Template Code wie eine Inline-Funktion zu behandeln und die Implementierung in die Headerdatei zu verlegen. Folglich liegt die Implementierung offen und jeder, der das Template verwendet, muss den kompletten Code neu übersetzen.

Dies sähe so aus:

```
myLib.h
//Deklaration und Implementierung
template<class T>
T min (T a, T b) {
return ((a<b) ? a : b);
};

myLib.cpp
#include 'myLib.h'
useLib.cpp
#include 'myLib.h'
int x = min<int>(10, 100);
```

Die letzte Möglichkeit wäre wie die Vorherige, aber man packt die Implementierung in einen separaten Header:

```
myLib.h
//Deklaration
template<class T>
T min (T a, T b);
```

```

//Lade Implementierung
#include 'myLib.icc'
myLib.icc
template<class T> //Implementierung
T min (T a, T b) {
return ((a<b) ? a : b);
};

useLib.cpp
#include 'myLib.h'
int x = min<int>(10, 100);

```

## 0.6 Fortgeschrittene Templates

Bei den fortgeschrittenen Templates spielen die Template Metaprogramme, welche eine gezielte Code-Generierung bewirken, eine große Rolle. Unter anderem gibt es auch noch die Introspection, dies ist die statische Analyse der vorhandenen Typen.

### 0.6.1 Template Metaprogramme

Die C++ Metaprogrammierung ist eine Technik, um in C++ Programmcode von Programmcode generieren zu lassen. Dabei finden besonders Templates Verwendung, weswegen man auch von Templatemetaprogrammierung spricht.

Man macht sich zu nutze, dass Templates schon während des Kompilierens ausgewertet werden. Mit Hilfe von Templatespezialisierung hat man so die Möglichkeit Code zu schreiben, der erst den eigentlichen Code generiert und zur Kompilationszeit ausgewertet wird. Dies verkürzt die Laufzeit, verlängert jedoch die Dauer des Kompilierens.

Die Templatemetaprogrammierung, welche eine äußerst mächtige Programmieretechnik ist, wurde speziell für C++ intensiv erforscht und entwickelt. So gibt es einen mit Hilfe von Templates realisierten Parsergenerator. Dieser erzeugt Unterprogramme zur grammatikalischen Analyse anderer Computerprogramme. Diese Unterprogramme werden von einem Compiler oder Interpreter eingesetzt. Üblicherweise werden Parsergeneratoren ( zum Beispiel yacc) in Verbindung mit einem Scannergenerator eingesetzt. Der Maschinengenerator muss dabei zusätzlich für jede Maschine generiert werden.

Die Boost-Bibliothek enthält viel Material zur C++-Metaprogrammierung. Diese Boost-Bibliothek ist eine Sammlung von freien C++ Bibliotheken, die ein breites Spektrum an portablen Problemlösungen bietet. Viele Teile der Boost Bibliothek sind fortgeschrittene Anwendungen von C++ Templates und somit direkte Anwendungen der generischen oder der Metaprogrammierung.

Die Templatemetaprogrammierung birgt auch Nachteile. Zum Beispiel gibt es in den bestehenden Entwicklungswerkzeugen keine Möglichkeiten um die Metagenerierung schrittweise zu verfolgen. Ein weiteres Problem besteht darin, dass es bislang noch schwierig ist, sinnvolle Fehlermeldungen für Metaprogrammierung auszugeben und sich somit nur schwer auf den eigentlichen Fehler schließen lässt.

Die Templatemetaprogrammierung ist Turing-vollständig, was aussagt, dass jeder Algorithmus durch Templatemetaprogrammierung umgesetzt werden kann.

In der Templatemetaprogrammierung gibt es keine veränderbaren Variablen, das bedeutet, dass einmal mit einem bestimmten Wert initialisierte Elemente ihren Wert für immer behalten. Anders als C++-Laufzeitprogramme stellen deswegen C++-Templatemetaprogramme eine Form der funktionalen Programmierung dar. Deshalb erfolgt die Flusskontrolle mit Hilfe von Rekursion.

Ein Beispiel hierfür ist die Potenzberechnung mit Hilfe von Metatemplates:

```

#include <iostream>
template<long B,unsigned long E>
struct pow_helper {
static const double value;
};
template <long B,unsigned long E>
const double pow_helper<B,E>::value=B*pow_helper<B,E-1>::value;
template<long B>
struct pow_helper<B,0>{
static const double value;
};
template <long B>
const double pow_helper<B,0>::value=1;
template<long B,long E>
struct power {
static const double value;
};
template <long B,long E>
const double power<B,E>::value= E<0 ? 1.0/pow_helper<B,-E>:
pow_helper<B,E>:: value;
int main() {
std::cout <<power<10,-3>::value <<std::endl;
};

```

Das Template `power` wird aufgerufen. Wenn der Exponent negativ ist, wird der Ausdruck  $1.0/B^{-E}$  berechnet. Um die eigentliche Potenz zu berechnen wird die Struktur `pow_helper` benutzt, die sich selbst aufruft, wobei der Exponent bei jedem Aufruf um 1 reduziert wird. Für den Fall, dass der Exponent Null ist, besitzt `pow_helper` eine Spezialisierung, welche, falls sie eintritt, das Ergebnis Eins zurückliefert liefert.

Also lässt sich der Code als:

```

P(B,E) := B * P(B,E-1)
P(B,0) := 1

```

etwas leserlicher beschreiben.

Auch die Metaprogrammierung birgt Vor- und Nachteile.

Zum einen muss man eine Abwägung treffen zwischen Übersetzungszeit und Ausführungszeit. Da der gesamte Template Quelltext zur Übersetzungszeit ausgewertet und eingesetzt wird, kann der ausführbare Code zwar effizienter sein, die Übersetzung hingegen dauert aber länger.

Zum anderen kann Templatemetaprogrammierung zu generischem Quelltext führen und zu erhöhter Wartbarkeit, da sie dem Programmierer erlaubt sich ausschließlich mit der Architektur zu beschäftigen.

Ein großer Nachteil ist die schwere Lesbarkeit der Metaprogramme, die die Wartung und Pflege der bereits vorhandenen Metaprogramme von unerfahrenen Metaprogrammierern fast unmöglich macht.

Die Portierbarkeit von Quelltext in dem viel Metaprogrammierung enthalten ist, kann sehr eingeschränkt sein, was von Unterschieden zwischen den Compilern hervorgerufen wird.

## 0.6.2 Introspection

Man kann zur Übersetzungszeit, mittels dem Overload-Resolution-Mechanismus und der Template-Spezialisierung, vieles über die vorhandenen Typen herausfinden.

## 0.7 Folgerung

Abschließend ist zu sagen, dass C++ Templates das notwendige Handwerkszeug bilden um mit C++ generischen Code zu erstellen. Heute existieren zahlreiche Bibliotheken, die auf diese Methode aufbauen, unter anderem die STL , Boost und viele andere.

Der Template Mechanismus ist sehr mächtig, er erlaubt die Template Metaprogrammierung und die Template Spezialisierung.

Jedoch sind Templates durch ihre hohe Komplexität nicht leicht zu erstellen, somit kommt es zu Debugging und Fehlermeldungen.

Zitat Bjarne Stroustrup:

„Software design is hard, and we need all the help we can get“

# Literaturverzeichnis

- [1] B. Stroustrup: *Die C++ Programmiersprache*, Addison-Wesley, 4. aktualisierte Auflage, (2000)
- [2] B. Stroustrup: *The Design and Evolution of C++*, Addison-Wesley, (1994)
- [3] D. Vandervoorde, N. M. Josuttis: *C++ Templates: The Complete Guide*, Addison-Wesley Professional, (2002)
- [4] R. Schneeweiß: *Moderne C++ Programmierung. Klassen, Templates, Design Patterns*, Springer, (2007)
- [5] K. Loudon: *C++. Kurz und gut.*, O'Reilly, (2003)
- [6] H. Tschabitschier: *Einführung in C++*, [http://ladedu.com/cpp/zum\\_mitnehmen/cpp\\_einf.pdf](http://ladedu.com/cpp/zum_mitnehmen/cpp_einf.pdf)