

C++ FOR

(Mittwochnachmittag)

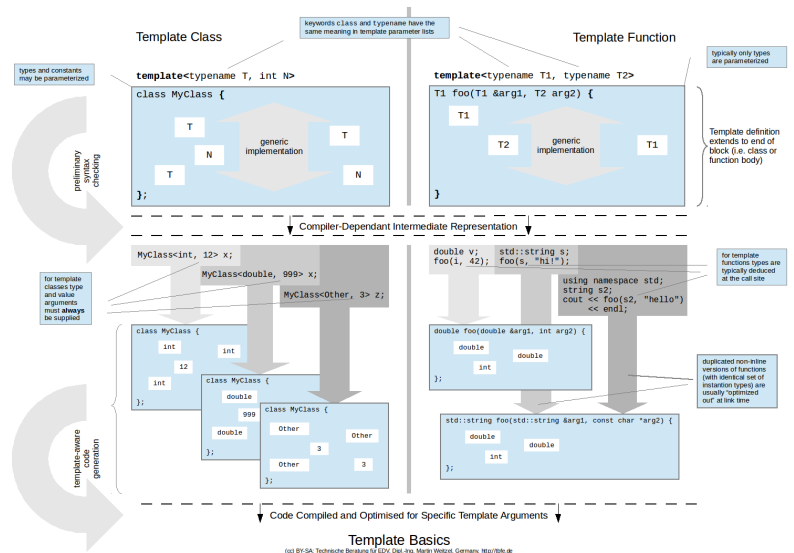
1. Templates selbst schreiben
 2. Optimierung von Templates
 3. Templates als Compilezeit-Funktionen
 4. Übung
-

Kürzere Pausen werden jeweils nach Bedarf eingelegt.

Die Besprechung der Musterlösung(en) erfolgt zu Beginn des folgenden Vormittags.

Temp

- Template-Klassen
- Template-Funktionen



Template-Klassen

Template-Klassen sind im Hinblick auf Datentypen und/oder andere Compilezeit-Konstanten parametrisierte Klassen.

Man spricht in diesem Zusammenhang auch von generischen Klassen.

Bei der Verwendung von Template-Klassen sind die entsprechenden Typ- und Wertargumente stets anzugeben.

Template-Funktionen

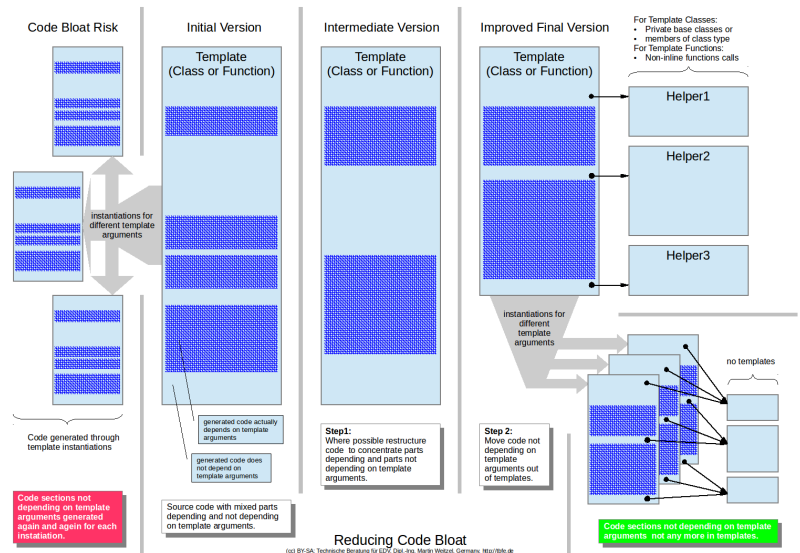
Template-Funktionen sind in der Regel im Hinblick auf Datentypen^{*} parametrisierte Funktionen

Bei der Verwendung von Template-Funktionen ergeben sich die tatsächlich zu verwendenden Typen oft direkt oder indirekt aus den Typen der Aufrufargumente.

^{*}: Die Parametrisierung im Hinblick auf Compilezeit-Konstanten ist bei Funktionen ebenfalls möglich, tritt in der Praxis aber sehr selten auf.

Optimierung von Templates

- Ursache für "Code-Bloat"
- Über einen Zwischenschritt ...
- ... zur optimierten Template



Ursache für "Code-Bloat"

Ein wesentlicher Grund für "unnötig erzeugten" Maschinencode ergibt sich oft aus der ungeschickten Strukturierung von Templates, bei denen eine unsystematische Vermischung existiert zwischen:

- Teilen, welche tatsächlich von den Instanziierungsparametern abhängigen Maschinencode erzeugen;
- anderen Teilen, welche immer ein und denselben Maschinencode hinauslaufen.

Vorbereitender Zwischenschritt

Die Vorbereitung für die Reduzierung von Code-Bloat besteht darin,

- möglichst große, zusammenhängende Abschnitte Abschnitte im Quelltext einer Template-Klasse oder -Funktion zu schaffen, die tatsächlich von den Instanziierungs- (Typ) -argumenten abhängigen und
- andere möglichst große Abschnitte, die immer immer auf ein und denselben Maschinencode hinauslaufen.

Optimierten Template

Alle Teile einer Template, die immer den selben Maschinencode erzeugt werden, können anschließend ausgelagert werden, nämlich bei einer

- Template-Klasse in eine eine **Nicht-Template** Basisklasse, und
- Template-Funktion in **Nicht-Template** Hilfsfunktionen.

Templates als Compilezeit-Funktionen

Eine weitere Sicht auf Templates ist es, sie als zur Compilezeit ausgeführte Funktionen zu verstehen.

Der wesentliche Schlüssel dazu ist, das folgende zu verstehen:

- Jeglicher "Input" muss aus Datentypen bestehen.*
- Jeglicher "Output" wird nur aus Datentypen sein.



Anders ausgedrückt:
Templates sind (auch) Typtransformationen zur Compilezeit.

*: Mit einem kleinen, später noch erläuterten Kunstgriff fallen darunter auch alle zur Compilezeit-Konstanten sowie die daraus berechenbaren Werte.

Wiederholung und Verzweigung

Da sich zur Compilezeit keine Schleifen mit `while` und keine Verzweigungen mit `if` programmieren lassen, muss man zu den entsprechenden Alternativen aus **Funktionalen Programmiersprachen** greifen:

- Jegliche Wiederholung muss per Rekursion und
- jegliche Fallunterscheidung durch Spezialisierung erfolgen.

Ein bisschen muss man das aber vielleicht trainieren ...*

*: ... was ganz gut mit einer Einführung in eine "echte" FP-Sprache wie beispielsweise **Haskell** geht.

Fakultäts-Funktion als Beispiel

Der übliche Ansatz ...

Die allseits bekannte Funktion zur Fakultätsberechnung kann man in C++ mit einer Schleife so programmieren programmieren:

```
unsigned long long fakul(unsigned long long n) {  
    auto result = 1uLL;  
    while (n > 0)  
        result *= n--;  
    return result;  
}
```

Oder auch zur - Laufzeit rekursiv - so:

```
unsigned long long fakul(unsigned long long n) {  
    return n ? n*fakul(n-1) : 0;  
}
```

Fakultäts-Funktion als Beispiel (2)

... und der im "Haskell-Stil"

```
unsigned long long fakul(unsigned long long n) {  
    return n*fakul(n-1);  
}  
unsigned long long fakul(0uLL) {  
    return 1;  
}
```

Natürlich ist das obige **kein gültiges C++** ... aber doch irgendwie verständlich, oder nicht?



Der Abbruch der in der allgemeinen Funktion eigentlich endlosen Rekursion erfolgt durch Spezialisierung von fakul für den Argumentwert 0 (als unsigned long long).

Fakultäts-Berechnung zur Compilezeit

Das folgende aber **ist** gültiges C++ ...

```
template<unsigned long long n>
struct fakul {
    static const
    unsigned long long result = n*fakul<n-1>::result;
};
template<>
struct fakul<0uLL> {
    static const unsigned long long result = 1;
};
```

... und doch auch irgendwie "verständlich", oder?

Der "Aufruf" in einem kleinen Testprogramm könnte dann so aussehen:

```
#include <iostream>
int main() {
    std::cout << fakul<5>::result << std::endl;
}
```

Weitere Typ-Transformationen in Beispielen

Eine Zeigerstufe hinzufügen

```
template<typename T>
struct add_pointer {
    typedef T* result;
}
```

Eine Zeigerstufe wegnehmen

```
template<typename T>
struct remove_pointer;
template<typename T>
struct remove_pointer<T*> {
    typedef T result;
}
```

Alle Zeigerstufen wegnehmen

(Denken Sie doch einfach mal selbst kurz nach!)

Type-Traits Library

Mit C++11 wurden die davor unter Boost entwickelten [Type-Traits] zum Standard.

Mehr dazu finden Sie hier:

- http://www.cplusplus.com/reference/type_traits/
- <http://en.cppreference.com/w/cpp/types/>

Übung

Ziel der Aufgabe:

Implementierung eines Algorithmus (im Stil der STL) und schrittweiser Ausbau zu einer Template.*

Weitere Details werden vom Dozenten anhand des Aufgabenblatts sowie der vorbereiteten Eclipse-Projekte erläutert.

*: Die letzten Ausbaustufen liegen etwas oberhalb der Zielsetzung dieser Schulung. Insofern bietet sich dazu vielleicht vorerst nur ein Studium der Musterlösungen zu diesen Aufgabenteilen an.