C++ FOR
(Thursday Morning)

---

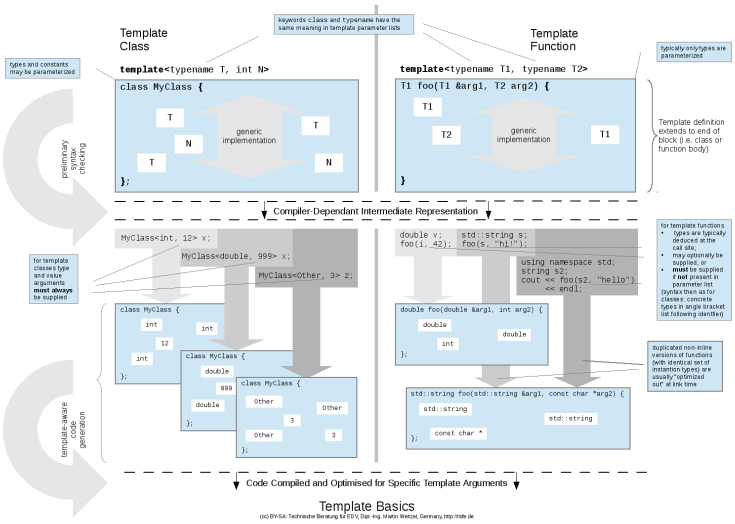
1. Templates selbst schreiben
2. Optimierung von Templates
3. Templates als Compilezeit-Funktionen
4. Fortgeschrittene Nutzung des Präprozessors
5. Praktikum

Kürzere Pausen werden jeweils nach Bedarf eingelegt.

Die Besprechung der Musterlösung(en) erfolgt im direkten Anschluss an die Mittagspause.

Templates selbst schreiben

- 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

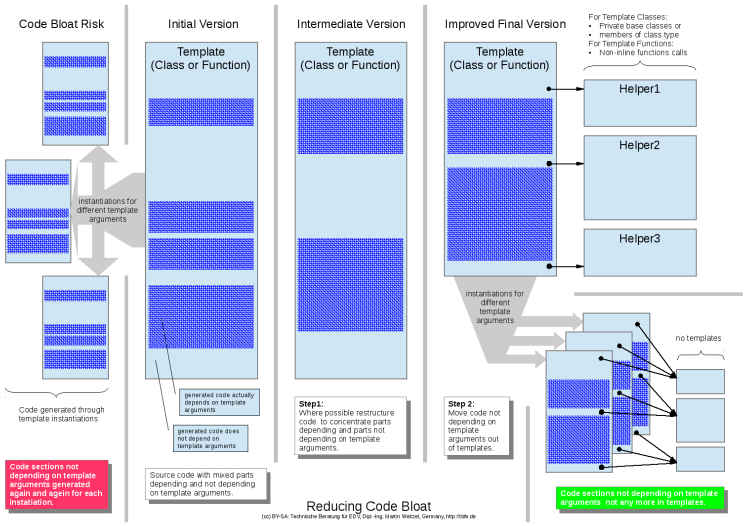
Bei der Verwendung von Template-Funktionen ergeben sich die tatsächlich zu verwendenden Typen oft direkt oder indirekt aus den Typen der Aufrufargumente.

*: 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"

"Unnötig erzeugter" Maschinencode ergibt sich oft aus einer ungeschickten Strukturierung von Templates.

Problematisch ist eine erhebliche Vermischung von

- Abschnitten, welche abhängig von den Instanziierungsparametern stets **unterschiedlichen** Maschinencode erzeugen, und
- Abschnitten, die stets **ein und denselben** Maschinencode erzeugen.

Vorbereitender Zwischenschritt

Die Vorbereitung für die Reduzierung von Code-Bloat sieht wie folgt aus:

In einer Template-Klasse oder -Funktion sind möglichst große, zusammenhängende Abschnitte zu schaffen,

- die Maschinencode erzeugen, der tatsächlich von den Instanziierungsparametern abhängt, und
- diese damit zu trennen von anderen, immer auf ein und denselben Maschinencode hinauslaufenden.

Optimierte Template

Abschnitte einer Template, die auf ein und denselben Maschinencode hinauslaufen, können ausgelagert werden, und zwar

- für eine Template-Klasse in eine **Nicht-Template** Basisklasse, und
- für eine Template-Funktion in eine **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" besteht aus Datentypen.*
- Jeglicher "Output" besteht aus Datentypen.

> (i) **Anders ausgedrückt:**
> Templates sind (auch) Typ-Transformationen zur Compilezeit.

*: Mit einem kleinen Kunstgriff fallen darunter auch sämtliche zur Compilezeit konstanten Werte von Grundtypen, sowie daraus berechenbaren Werte.

Wiederholung und Verzweigung

Allerdings gibt es kein Konstrukt für zur Compilezeit ausgeführte

- Schleifen (analog `while` zur Laufzeit) und
- Verzweigungen (analog `if` zur Laufzeit).

> Verwendbar sind dagegen die entsprechenden Alternativen der *Funktionalen Programmierung* (FP).

Es muss vielmehr

- Wiederholung durch **Rekursion** und
- Fallunterscheidung durch **Spezialisierung**

ausgedrückt werden.

Ein bisschen sollte man das vielleicht zunächst trainieren ...*

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

Fakultäts-Funktion als Beispiel

Der übliche Ansatz …

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

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

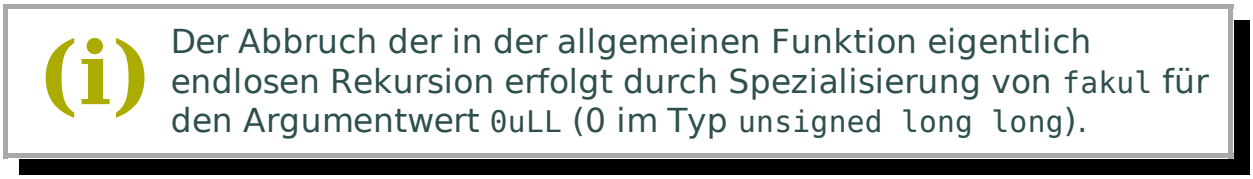
Oder auch – zur Laufzeit rekursiv – so:

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

… 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?



Berechnung mit C++-Template

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

```
// the primary template (internally recursive) ...
template<unsigned long long n>
struct fakul {
    static const unsigned long long result = n*fakul<n-1>::result;
};

// ... and its specialisation (stops recursion)
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;
}
```

Beispiele für Typ-Transformationen

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 zuvor 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/

Fortgeschrittene Nutzung des Präprozessors

- Der C++-Präprozessor beherrscht kein C++
- Verwendung von Makro-Argumenten als String-Literal (Stringizing)
- Verketten von Makro-Argumenten zu neuen Tokens (Token Pasting)
- Systematische, tabellengesteuerte Quelltext-Erzeugung
- Allgemeine Stil-Hinweise und weitere Tipps

Der Präprozessor kennt kein C(++)

Der Syntax des Präprozessors ist extrem einfach und erkennt nur:

- Kommentare sowie Zeichen- und Zeichenketten-Literale*
- Zeilenverkettung durch Abschluss einer Zeile mit Gegenschrägstrich
- Präprozessor-Direktiven in Zeilen beginnend mit einem Hash-Zeichen (White-Space direkt vor und hinter # optional möglich).
- Bezeichner, denen optional ein paar runder Klammern folgt, und innerhalb dieser Kommata (als Argumenttrenner) sowie weitere, paarige runde Klammern (welche die Wirkung enthaltener Kommata als Argumenttrenner aufheben).

[!] Makro-Bezeichner liegen außerhalb der C++-Namespaces und ihr Ersatztext wird ohne Beachtung des syntaktischen Kontexts als einfache Text-Substitution eingesetzt.

*: Dies bedingt sich gegenseitig, denn sonst könnten z.B. Zeichen-Literale und Kommentare keine Gänsefüßchen (`"`) und Zeichenketten-Literale keine Kommentarbegrenzer (`/*`, `*/` und `//`) oder einfache Apostrophe (`'`) enthalten.

Stringizing

Unter *Stringizing* wird die Möglichkeit verstanden, das Argument einer Makro-Expansion als String-Literal zu verwenden:

- Es wird dann quasi automatisch in doppelte Gänsefüßchen eingeschlossen.
- Wenn nötig, werden für einzelne Zeichen Escape-Sequenzen ersetzt.

Als Beispiel ein Makro, der einen Ausdruck textlich und mit dem berechneten Wert auf Standardausgabe schreibt:

```
#define PrintX(x) \
        std::cout << #x << ": " << (x) << '\n';
```

Die Anwendung kann (beispielsweise) wie folgt aussehen:

```
int main() {
    auto value = 42;
    PrintX(value++)
    PrintX(++value)
    PrintX(value)
    PrintX(value *= 2)
}
```

Token Pasting

Hierunter versteht man das "Aneinanderkleben" von Makro-Argumenten und fest vorgegebenen Bestandteilen, um daraus neue Bezeichner zu erzeugen:

```
#define DEFINE_ERROR_THROWER(clazz, name)\
    virtual void clazz::throw_ ## name() {throw clazz::name();}
…
DEFINE_ERROR_THROWER(MyParser, PrematureEndOfFile)
DEFINE_ERROR_THROWER(MyParser, InvalidExpression)
```

"Tabelle" als Vorbereitung

Hierfür könnte z.B. eine Liste von Bezeichnern und Texten vorliegen, für die – evtl. an ganz unterschiedlichen Stellen eines Programms – unterschiedlicher Quelltext systematisch erzeugt werden soll:

```
#define FOR_ALL_ERRORS(m)\
    m(PrematureEndOfFile, "file ends prematurely")\
    m(IncompleteLine    , "line ends prematurely")\
    m(InvalidExpression , "bad expression syntax")\
    … // usw.
```

Die Grundlage dieser Technik besteht darin, den Namen eines zu expandierenden Makros einem anderen Makro als Parameter zu übergeben. Dieser Makro könnte nun mit den beiden auf der nächsten Seite gezeigten Makros aufgerufen werden:

```
// somewhere ...
FOR_ALL_ERRORS(DEF_ERROR_CLASS)
```

```
// ... somewhere else
FOR_ALL_ERRORS(DEF_THROW_HELPER)
```

Eigentliche Quelltexterzeugung

Die einzige Regel ist, dass die an `FOR_ALL_ERRORS` zu übergebenden Makros genau zwei Argumente entgegennehmen müssen:

```
#define DEF_ERROR_CLASS(name, desc)\
    class name : public std::runtime_error {\
        public: name() : std::runtime_error(desc) {}\
    };
```

Diese müssen natürlich nicht zwingend im Ersatztext verwendet werden:

```
#define DEF_THROW_HELPER(name, desc)\
    virtual void throw_ ## name() { throw name(); }
```

Eine intensive, praktische Verwendung dieser Techniken zeigt das Beispiel: Examples/Utilities/TypePrinter/tp-final.cpp

Einige weitere Präprozessor-Tipps

Generell sollten

- Makros zur systematischen Quelltext-Erzeugung einfach gehalten werden, und
- der Präprozessor nur dann für die systematische Code-Erzeugung genutzt werden, wenn es im Sinne der folgenden Ziele geschieht:
  - Signifikante Vereinfachungen
  - Höhere Fehlersicherheit
  - Verbesserte Wartbarkeit*

> Es kann durchaus sinnvoll sein, im Ersatztext von *Helfer-Makros* wiederum *Helfer-Funktionen* und/oder *Helfer-Templates* zu verwenden, insbesondere wenn deren Argumentlisten systematische Ähnlichkeiten und Wiederholungen aufweisen.

*: Z.B. weil absehbar ist, dass weitere, ähnlich gelagerte Fälle nach und nach zu ergänzen sind und/oder künftig für alle bisherigen Fälle gemeinsame Anpassungen erforderlich werden könnten.

Fortsetzungszeilen fördern die Lesbarkeit

Dies wurde in den gezeigten Beispielen schon häufig praktiziert.

Ggf. kann zusätzlich auch erwogen werden,[*]

- den Gegenschrägstrich für die Fortsetzungszeilen immer in ein und dieselbe Spalte zu setzen,
- und das Ende der Makro-Definition durch einen Kommentar expliziter sichtbar zu machen.

```
#define DEF_ERROR_CLASS(name, desc)                \
class name : public std::runtime_error {           \
    public: name() : std::runtime_error(desc) {}   \
};                                        // END-OF-MACRO
```

[*]: Probieren Sie aber besser vorab aus, ob eine solche, auf gute Lesbarkeit zielende Formatierung nicht schon beim nächsten Lauf eines "C++-Beautifiers" wieder zerstört wird ... und auch nicht durch die gutgemeinte "Einrückungshilfe" des syntaxbewussten Editors Ihres Kollegen, der damit Ihre Quelltexte ab und zu bearbeitet.

Namenskonventionen für Makros

Da Makronamen nicht mit C++ Namespaces verbunden sind, sollte jedes Projekt verbindliche Regeln aufstellen, welche Namen für Makros verwendet werden dürfen.*

Diese könnten z.B. sein:

- Erstes Zeichen Großbuchstabe,
- gefolgt von zwei oder mehr weiteren Zeichen,
  - die Großbuchstaben, Ziffern oder Tiefstrich sein dürfen,
- Ende mit `_H` aber nur bei Makros, die Include-Guards steuern.

Die "Mindestens-drei-Zeichen"-Regel beugt Komplikationen vor, die andernfalls durch die verbreitete Praxis entstehen könnten, für Typ-Parameter von Templates `T` (oder `T1`, `T2`, `T3`, ...) und für Wert-Parameter `N` (oder `MIN`, `MAX` ...) zu verwenden.

*: Oftmals weniger bekannt ist auch, dass nicht nur der C89-Standard alle Makronamen, welche mit einem Tiefstrich beginnen, für Zwecke der Implementierung reserviert, sondern andere, evtl. in einem Projekt relevante Standards mitunter deutlich weitergehende Regeln enthalten. So reserviert z.B. POSIX alle Makros für interne Zwecke, deren Namen die mit `LC`, `E`, `SIG` oder `SIG_` beginnen, sofern ein Großbuchstabe oder eine Ziffer folgt.

Makros nur lokal definieren

Abschließend noch zwei weitere vorbeugende Maßnahmen gegen Überraschungen, welche nicht beabsichtigten Ersetzungen des Präprozessors vermeiden helfen:

- Nur über kurze Quelltextabschnitte hinweg nötige Makros zur systematischen Erzeugung von Quelltext sollten ggf. direkt nach ihrer letzten Verwendung mit `#undef` wieder gelöscht werden.
- Ferner kann durch einen vorgeschalteten Test mit `#ifdef` das unbeabsichtigte Überschreiben eines (anderen) Makros verhindert werden, der eine weiter ausgedehnte Sichtbarkeit hat und diese auch gezielt haben soll.

Beides zeigt das Beispiel auf der nächsten Seite, welches die oft notwendige Umwandlung von per `enum` definierten Bezeichnern in entsprechende (druckbare) Zeichenketten demonstriert.*

---

*: Dieses Beispiel verwendet noch zwei weitere, mit C++11 eingeführte Neuerungen, nämlich die neue Syntax zur *Typ-Definition mit* `using` und `enum` *Klassen*. Bei letzteren sind die einzelnen Bezeichner qualifiziert zu verwenden. Somit zeigt sich zugleich eine Stärke der Quelltext-Erzeugung durch Makros: um wieder auf die klassischen Aufzählungstypen (zurück) zu wechseln, müsste lediglich der Makro `MAP_COLOUR_TO_STRING` angepasst werden, nicht aber dessen Verwendungen.

Beispiel für lokalen Makro

```
enum class Colour : unsigned char { Red, Blue, Green };
…
#ifdef MAP_COLOUR_TO_STRING
#error "macro 'MAP_COLOUR_TO_STRING' already defined"
#endif
std::string colour2string(Colour c) {
    switch (c) {
       #define MAP_COLOUR_TO_STRING(c)\
        case Colour::c: return "Colour::" #c;
           // ----------------------------
              MAP_COLOUR_TO_STRING(Red)
              MAP_COLOUR_TO_STRING(Blue)
              MAP_COLOUR_TO_STRING(Green)
           // add more colours to map here
           // ----------------------------
       #undef MAP_COLOUR_TO_STRING
        default: {
            using ULL = unsigned long long;
            return "Colour::#"
                 + std::to_string(static_cast<ULL>(c))
                 + " (not mapped to a name)"
        }
    }
}
```

Praktikum