

C++ FOR

(Thursday Morning)

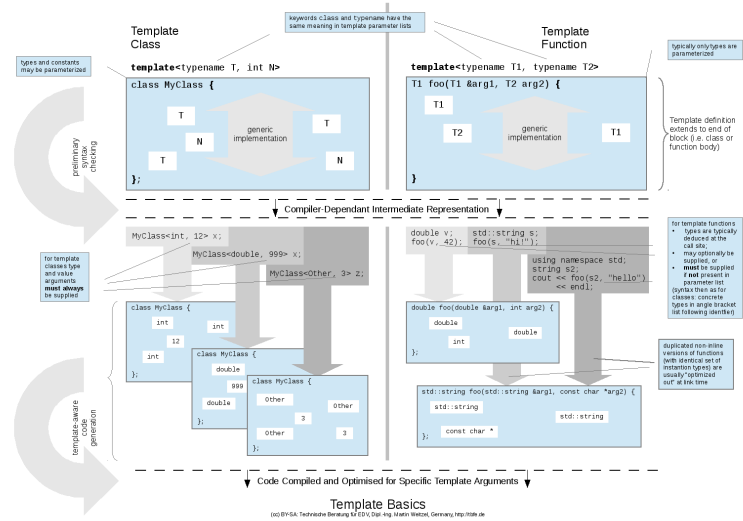
1. Implementing Templates
 2. Optimising Templates
 3. Perfect Forwarding with Templates
 4. Templates as Compile Time Functions
 5. Variadic Templates (Parameter Packs)
 6. Advanced Use of the Preprocessor
 7. Praktikum
-

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.

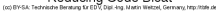
Bei der Verwendung von Template-Klassen sind die entsprechenden Typ- und Wertargumente stets anzugeben.

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.



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 Instanziierungs-Parametern 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 Instanziierungs-Parametern 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.

Perfect Forwarding with Templates

A special use case for templates is *Perfect Forwarding*, which basically is about keeping the "Rvalue-ness" of a function argument until it is used.

It can be applied

- either in [cookbook style](#) or
- with first explaining (and consequently trying to grasp) a lot of its [background](#).

One key point with both approaches is to understand that the rvalue reference declarator (&&) is special if used with a template argument.

Scott Meyers has even coined a special term for it: [Universal References](#)

Perfect Forwarding - Cookbook Style

Perfect forwarding is achieved by applying the following recipe:^{*}

```
template<typename T1, typename T2>
void foo(T1 &&arg1, T2 &&arg2) {
    bar(std::forward<T1>(arg1));
    baz(std::forward<T2>(arg2));
};
```

If there are overloads of bar and baz optimized for lvalue and rvalue arguments, the appropriate one will be called.

^{*}: Perfect forwarding always requires the use of templates, even if it is applied to cope with the exponential explosion of overloads necessary if only arguments of known types are to be forwarded, like here:

```
// Assume objects of these classes can be arguments to foo, which it
class MyClass { ... }; // it forwards to bar and baz and the caller may
class Another { ... }; // either supply lvalues or rvalues as argument:
...
void foo(const MyClass &arg1, const Another &arg2) { ... }
void foo(const MyClass &arg1,         Another &&arg2) { ... }
void foo(MyClass &&arg1, const Another &arg2) { ... }
void foo(MyClass &&arg1,         Another &&arg2) { ... }
```

Perfect Forwarding - The Background

The background of perfect forwarding combines three essential pieces:

1. For a template argument T&& there is a special deduction of T:
 - T is *reference* for lvalue arguments,
 - otherwise it is the plain argument type (as usual with & and const stripped away).
2. Reference collapsing defines what happens if two reference declarators are applied in a row:
 - & and & → &
 - & and && → &
 - && and & → &
 - && and && → &&
3. `std::forward<T>(expr)` statically casts `expr` to T&&.



For more information on `std::forward` see:
<http://en.cppreference.com/w/cpp/utility/forward>

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.

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 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 obiges **kein gültiges C++** ... aber doch irgendwie verständlich, oder nicht?

Die Idee ist, dass der Abbruch der gemäß der allgemeinen Funktion eigentlich endlosen Rekursion durch die Spezialisierung von fakul für den Argumentwert 0uLL (0 im Typ unsigned long long) erfolgt.

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 ...*

*: The solution – of course – is this:

```
// primary template  
template<typename T> struct remove_all_ptr { typedef T result; };  
// specialisation  
template<typename T> struct remove_all_ptr<T*> { typedef remove_all_ptr<T> result; };
```

Type-Traits Library

With C++11 the [Type-Traits](#) originally developed as a Boost library became a part of the standard.



For more information on standard type traits see:

http://www.cplusplus.com/reference/type_traits/

<http://en.cppreference.com/w/cpp/types/>

C++14 simplified the use of the standardised type traits with a number of template aliases (instead of accessing `::type` for traits returning types) and conventional conversion to `bool` (`as_constexpr`) is implemented for traits with a static `constexpr bool` value).

- For some type trait `xxx` accessed conventionally via `typename xxx<T>::type`, in C++14 just `xxx_t<T>` may be used.
- For some type trait `yyy` accessed conventionally via `yyy::value`, in C++14 just `yyy{}.` may be used.

Using SFINAE with `std::enable_if`

Often, after doing type calculations, the final goal is to choose among several implementations of a given function.

If this is not the automatic effect from calculating a type that selects the appropriate overload, `std::enable_if` and **SFINAE** can be applied.

The basic idea is this:

- Create an overload set of (template) functions that is deliberately ambiguous.
- Make all but one instantiations of the functions in this set fail for a particular condition.

As compile time calculations must express every result as a type (last and finally), `std::enable_if` creates an illegal type as "substitution failure" and the particular instantiation is removed from the overload set.

Example for SFINAE with `std::enable_if` (C++11)

The following example selects between two implementations of `foo`, depending on whether the arguments `arg1` and `arg2` are objects related with each other as base and derived class:

```
template<typename T1, typename T2>
typename std::enable_if<
    std::is_base_of<T1, T2>::value
>::type foo(T1 arg1, T2 arg2) { ... } // called for T1 is base class of T2

template<typename T1, typename T2>
typename std::enable_if<
    !std::is_base_of<T1, T2>::value
>::type foo(T1 arg1, T2 arg2) { ... } // called for T1 is NOT base class of T2
```

Example for SFINAE with `std::enable_if_t` (C++14)

With the additions C++14 made to the standard type traits, the previous example can be slightly simplified to:*

```
template<typename T1, typename T2>
std::enable_if_t<
    std::is_base_of<T1, T2>{}
> foo(T1 arg1, T2 arg2) { ... } // called for T1 is base of T2

template<typename T1, typename T2>
typename std::enable_if_t<
    !std::is_base_of<T1, T2>{}
> foo(T1 arg1, T2 arg2) { ... } // called for T1 is NOT base of T2
```

*: Note that the support for this simplifications is close to trivial In namespace `std { ... }`:

```
template<bool C, typename T = void> using enable_if_t = typename std::enable_if<C, T>::type;
```

And:

```
template<T1, T2>
struct is_base_of {
    /* as in C++11, set according to result */ static constexpr bool value = ...;
    /* added in C++14 */ constexpr explicit operator bool() const { return value; }
}
```

Variadic Templates

C++11 introduced [Parameter Packs] to allow templates to accept a variable number of arguments.

-
- Defining Variadic Templates
 - Unpacking Parameter Packs
-

Generally variable argument lists help

- to avoid much repetitive systematic coding and
- at the same time removes arbitrary upper limits.



For more information see on variadic templates see:
http://en.cppreference.com/w/cpp/language/parameter_pack

*: The ellipsis was originally introduced with C to indicate variable length argument list.

Defining Variadic Templates

In the **definition** of template a symbolic name be introduced with three dots appended.

In this syntax the identifier right to the dots introduces kind of a list,^{*} actually representing

- either sequences of type names
- or values of a specified type.

This can be used with [template classes](#) or [template functions](#).

^{*}: Before C++11, to implement templates with a variable number of arguments required repeated, similar code (though some libraries like [Boost.Preprocessor](#) provided means to generate such using the C/C++ preprocessor):

```
// defining a tuple class ...
template<typename T1>
class tuple { T1 first; ... };
template<typename T1, typename T2>
class tuple { T1 first; tuple<T2> rest; ... };
template<typename T1, typename T2, typename T3>
class tuple { T1 first; tuple<T2, T3> rest; ... };
template<typename T1, typename T2, typename T3, typename T4>
class tuple { T1 first; tuple<T2, T3, T4> rest; ... };
... // etc. up to some upper limit
```

Example: Variadic Template Class

```
// Definition:
template<typename... Ts>
class MyClass {
    ...
};

// Valid Uses:
... MyClass<int, double> ...
... MyClass<const char *> ...
... MyClass<std::string, bool> ...
... MyClass<int, int, int, int> ...
```

```
// Definition:
template<int... Ints>
class MyIntegers { ... };
template<std::string... Words>
class MyStrings { ... };

// Valid Uses:
... MyInts<2, 3, 5, 7, 11, 13> ...
... MyStrings<> ...
... MyStrings<
    std::string("hello"),
    std::string("world")> ...
```

In a template class, if this mechanism is combined with other (fixed) template arguments, the variable part must come last:

```
template<typename T1, unsigned int N, typename T2, bool... Bs>
class Whatever { ... };

...
... Whatever<std::string, 7u, int, true, false, true, true, false> ...
... Whatever<const volatile unsigned long&, (~0u >> 12), void> ...
```


Example: Variadic Template Function

The general syntax is similar to variadic template classes, but the type list usually also occurs as part of the argument list:

```
template<typename... Ts>  
std::string concat(char, Ts...); // prototype only, so far
```

In signature of `concat` state that the function is to be called with a first (mandatory) argument of type `char` and any more arguments of arbitrary type, hence – in principle – all of the following calls were correct:

```
... concat(' ', "hello", 7, "world" '!') ...  
... concat('+', std::string("whatever"), 2/1.0, true, false) ...
```

Implementing Variadic Templates

Implementing variable templates usually has two options:

- Internally forwarding to another variadic template.*
- Applying recursion at compile time.

For the latter the following has to be understood:

- Recursion is the (only) way to write "loops" at compile time, and
- there must be a "condition" to stop an otherwise endless recursion.

Furthermore:

Conditions at compile time are typically expressed

- as specialisation for some border case, or
- by selecting from an overload-set with **SFINAE**.

*: Of course, this only shifts "the burden of implementation" to the callee ...

Unpacking Parameter Packs

The second use of the ellipsis is to unpack a parameter pack, by writing it after some "pattern" containing at least one name of a variadic template representing a parameter pack.

- If the name represents a type list, it may be used wherever type lists are acceptable, e.g. to instantiate some (other) variadic template.
- If the name represents a value list, it may be used to expand to parameter lists of a variadic function (template or classic) or as initialiser for data structures or be used as an `std::initializer_list`.
- If the name is the formal argument name representing an unpacked formal parameter list, the pattern will be expanded once for each parameter.
- Finally, several packs may be unpacked simultaneously.

Some practical examples (following) should help to clarify the above.

Example: Compile Time Recursive Data Structure

The following class shows an approach how something like `std::tuple` might be implemented with a variadic template:*

```
template<typename T,  
        typename... Ts>  
struct MyTuple {  
    T first;  
    MyTuple<Ts...> rest;  
    MyTuple(T f, Ts... r)  
        : first(f), rest(r)  
    {}  
};
```

The general case (left) ...

... and its specialisation for the empty border case, required to stop endless recursion (below):

```
template<>  
struct MyTuple<> {  
};
```

With this, `MyTuple` objects could be created as follows:

```
MyTuple<int, double, const char*> x(1, 0.5, "one by three");  
MyTuple<std::string, const char*> y("one", "two");  
MyTuple<> z;
```

*: Note that this example is stripped down to its bare bones deliberately omits the finer points that were important for useful and performant tuple class.

Example: Compile Time Recursive Functions

The function `concat` was already introduced as example for a variadic template function.*

```
// FIRST(!) the basic form that stops recursion:
std::string concat(char) {
    return {};
}

// THEN the recursive form (to unfold at compile time):
template<typename T, typename... Ts>
std::string concat(char sep, T, arg, Ts... args) {
    return arg.append(&sep, 1) + concat(sep, args...);
};
```



The definition of the basic version (stopping recursion) and the recursive version must not be reversed – or the program will not compile any more.

*: This example too is stripped down to its bare bones and deliberately omits the finer points that would make sense for useful and performant implementation.

Example: Stopping Recursion with SFINAE

Stopping recursion by using SFINAE is demonstrated in the following implementation of a `get<N>` member function of class `MyTuple`:^{*}

```
template<typename T, typename... Ts>
class MyTuple {
    ...
    ... // as shown on a previous page
    ...
    template<std::size_t N>
    typename std::enable_if<N == 0, T>::type
    get() { return first; }

    template<std::size_t N>
    using Rtype = decltype(rest.template get<N-1>());

    template<std::size_t N>
    typename std::enable_if<N != 0, Rtype<N>>::type
    get() { return rest.template get<N-1>(); }
};
```

^{*}: Note that implementing `get<N>` as a member is like [Boost.Tuple](#) provides member access, while [std::tuple](#) uses a non-member overload.

Simultaneous Unpacking of Parameter Packs

Two (or more) parameter packs may also be unpacked simultaneously, as shown in the following example. It demonstrates how `std::make_unique` (missing from C++11*) may be implemented:

```
template<typename T, typename... Args>
std::unique_ptr<T> make_unique(Args&&... args)
{
    return std::unique_ptr<T>{new T(std::forward<Args>(args)...)};
    //           /^^^^ ^^^^^\
    // simultaneously unpack argument types and names
}
```

The interesting part with respect to simultaneous unpacking is marked in the comment.

The example also demonstrates how [perfect forwarding](#) can be applied to whole argument lists with ease in a single statement

*: Though a factory function for `std::unique_ptr` similar to `std::make_shared` for `std::shared_ptr` makes sense, it was not defined in C++11, but only added in C++14.

C++14 `std::integer_sequence` Template

As code implementing variadic templates needs to "unfold" at compile time, C++14 provides the following compile time integer sequences:

- A template `std::integer_sequence` holding a sequence of arbitrary values of a given integral type.
- A template `std::index_sequence` as special form of the above where the integral type is `std::size_t`.
- A helper template `std::make_index_sequence` to generate a given number of consecutive values, starting from zero.
- A helper template `std::index_sequence_for` to generates an index sequence with the length of a given parameter pack.



For more information on the above including examples see:
http://en.cppreference.com/w/cpp/utility/integer_sequence

Fortgeschrittene Nutzung des Präprozessors

- Der C++-Präprozessor beherrscht kein C++
 - Verwendung von Makro-Argumenten als String-Literal (Stringizing)
 - Verketteten 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)
```

Systematische Quelltexterzeugung

"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 ...                // ... somewhere else
FOR_ALL_ERRORS(DEF_ERROR_CLASS)  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 Quelltext-Abschnitte 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