Agenda: Modern C++

Focussing

- Template Meta-Programming

- by: Dipl.-Ing. Martin Weitzel
- for: MicroConsult GmbH Munich

- Example: template_class/demo.cpp
- Example: template_function/demo.cpp
- Example: meta_programming/demo.cpp
- Example: type_calculation/demo.cpp
- Example: type_traits/demo.cpp
- Example: function_sfinae/demo.cpp
- Example: class_sfinae/demo.cpp
- Example: parameter_pack/demo.cpp
- Example: variadic_function/demo.cpp
- Example: my_tuple/demo.cpp

Notice to the Reader

- This document supplies the **Guiding Thread** only
  - It is not recommended to be read it "as-is" stand alone
  - The bulk of teaching material is in the live demos
    - created and augmented throughout this course, while
    - **YOU** – the participants - control for each topic
      - the depth of coverage and
      - the time spent on it
- In the electronic version feel free to follow the links
- The Printed version is provided for annotations in hand-writing

---

If you nevertheless want to use this document for self-study, not only to follow the links to the compilable code, **also vary and extend it**.

If you need suggestions what to try (add or modify) you will find some in this presentation and a lot more usually in the example code itself.

There are two basic forms:

- **Conditional code `#if … #else … #endif`**
  - Usually both versions work, demonstrating alternative ways to solve the particular problem at hand
  - Sometimes not all versions are compatible with C++11 but may require C++14 or even C++14 features

- **Commented-out lines – sometimes with alternatives**
  - Sometimes these are shown because they **do not compile** and should indicate that a particular solution that may even seem attractive at first glance is **not** the way to go.
  - Furthermore, if removing the comment often some other line (in close proximity, typically the previous or next one) needs to be commented.

Understanding the code by trying variations is the crucial step to actually internalise the topics covered!

Template Basics

- Class Templates
- Function Templates

Class Templates

- Originally designed for type-generic container classes
  - At Implementation-Time there is a **definition**
  - At Compile-Time **instantiation** follows

http://en.cppreference.com/w/cpp/language/class_template

Class Template Definition

- At Implementation-Time:
  - Keep some type(s) generic
  - Keep some value(s) generic
- Much like regular class ...
  - ... starting with template argument list

Example: template_class/demo.cpp

```
class point {
    double xc, yc;
public:
    point(double xc_, double yc_) : xc(xc_), yc(yc_) {}
    double x() const {return xc;}
    double y() const {return yc;}
    double x(double x_) {return xc = x_;}
    double y(double y_) {return yc = y_;}
    point shifted(double xdelta, double ydelta) const;
};
point point::shifted(double xd, double yd) const  {
    return {xc + xd, yc + yd};
}
```

- How to parametrize the type of xc, and yc?

Example (cont.): template_class/demo.cpp

```
template<typename T>
class point {
    T xc, yc;
public:
    point(T xc_, T yc_) : xc(xc_), yc(yc_) {}
    T x() const {return xc;}
    // ...
};

template<std::size_t N>
class fstring {
    char fs[N+1];
    void copy(const char* cp) {std::strncpy(fs, cp, N)[N] = '\0';}
public:
    fstring(const char *init) {copy(init);}
    fstring& operator=(const fstring& rhs) {copy(rhs.fs); return *t
    // ...
};
```

- Which of the both classes above templates a type?
- which a compile time constant?
- Could both be combined?
- (For which of both class would the latter perhaps make sense?)

Class Template Instantiation

- Concrete template arguments **must** be supplied on instantiation
  - Class name should be thought "including" concrete types, i.e.
  - Same template with different arguments denote different classes

Example (cont.): template_class/demo.cpp

```
template<typename T>
class point {
    T xc, yc;
public:
    point(T xc_, T yc_) : xc(xc_), yc(yc_) {}
    T x() const {return xc;}
    // ...
    T y(T y_) {return yc = y_;}
    point shifted(T xd, T yd) const;
};
template<typename T>
point<T> point<T>::shifted(T xd, T yd) const {
    return {xc + xd, yc + yd};
}

int main() {
    point<double> a{3.5, 7.0};
    point<int> c{10, 20};
    // ...
}
```

- What if the instantiation types were *move only* ?

Class Template Specialisation

- Existing *Primary Templates* may be specialised
- Syntactically recognized:
  - Angle brackets follow class name in definition ...
    - ... holding argument list **identical** to primary template
  - "Outer" argument list must be different ("specialise" something):
    - Empty for a full specialisation
    - Otherwise holds names for deduction
- **Compiler selects "most specialised" matching version**

http://en.cppreference.com/w/cpp/language/template_specialization

http://en.cppreference.com/w/cpp/language/partial_specialization

Example (cont.): template_class/demo.cpp

Function Templates

- Originally designed for type-generic algorithms

http://en.cppreference.com/w/cpp/language/function_template

Function Template Definition

- At Implementation-Time:
  - Keep some type(s) generic (typical)
  - Keep some value(s) generic (rare but possible)
- Much like regular function ...

  ... but starts with template argument list

Example: template_function/demo.cpp

Function Template Instantiation

- Concrete argument values **may** be supplied ...
- ... but more often are deduced from call arguments
  - Generic name in argument list may be adorned
  - **Be sure to understand what exactly is deduced**

Example (cont.): template_function/demo.cpp

Function Template Overloading

- Function templates may be overloaded by non-templates
  - **Exactly matching overloads get always preferred**
- There may be also may be templated overloads
  - If any, the "closest match" gets selected ...
  - **... or Compile-Error if not unique**
- Finally, function templates ma also be **fully** specialised

http://en.cppreference.com/w/cpp/language/overload_resolution

http://en.cppreference.com/w/cpp/language/template_specialization

---

Note: there is no syntax for partial specialisation of function templates – maybe because the rules are complicated enough even without ... :-/

http://www.gotw.ca/gotw/049.htm

Meta-Programming

- The key insight is:
  - any instantiation of a template is requires
  - that the compiler carries out some internal calculation

Meta-Programming Basics

- The C++ template system is a full programming language ...
  - ... yet often not the most convenient one
  - (at least given for what it was used in the past)
- But inconvenience is at the site of the implementor ...
  - ... to ease the work of its clients
  - **except for messy error messages**

Be sure to understand: anything talked about with respect to Meta-Programming means things that happen at Compile-Time, though often an executable program needs to be created too, but simply to show the effect.

Example: meta_programming/demo.cpp

Template Meta-Programming Input

- Any "input" to a meta program is
  - either a type
  - or a value
- Needs to explicitly stated somewhere in the source code

Example (cont.): meta_programming/demo.cpp

Template Meta-Programming Output

- Any "output" of a meta program is
  - "something that can be compiled"
  - or an error message

Example (cont.): meta_programming/demo.cpp

For demonstration purposes meta programming often includes a small program that can be compiled to an executable, showing the intended effect (e.g. by printing a calculated type).

Implementation Selection

- Typical meta programming goal is "implementation selection"
  - In simple cases with (direct) specialisation
  - Alternatively with overloading and SFINAE
- E.g. choose between
  - value and reference argument in a function call
  - member-wise copy and `std::memcpy` in a container

This page mainly aims to the inpatient who at least want to have a slight idea where all this "meta ..." leads to.

Template Meta-Programming Functions

- Any template class also constitutes a (meta-) function
  - its arguments come from the template argument list
  - it is called by instantiating the template
- Calling conventions make sense, e.g. a member
  - ...::`result` that "calls" the meta function, returning
    - *either* a type
    - *or* a value
  - ...::`type` that "calls" the meta function, returning a type
  - ...::`value` that "calls" the meta function, retuning a value

Example (cont.): meta_programming/demo.cpp

Template Meta-Programming Branches

- **All** branching in Meta-Programming
  - needs to be done by specialisation
- I.e. there needs to be a primary template ...
  - ... **either** covering the general case ...
  - ... **or** just being declared but not defined ...
  - ... if there are specialisations for all possible cases

Example (cont.): meta_programming/demo.cpp

Template Meta-Programming Loops

- In C++98 any meta programming loops
  - need to resolved to recursion
  - terminated by specialisation

Only for parameter packs (variadic templates) this is slightly released in C++11 and further generalised to fold expressions in C++1z.

Example (cont.): meta_programming/demo.cpp

Compile-Time Type Calculations

- Internal calculations may transform types
  - There are many practical uses …
  - … though mostly not visible at first glance

Please keep calm, be patient, and follow the examples, even if you have no idea to where it finally leads.

Example: type_calculation/demo.cpp

Practical Uses of Type Calculations

- Remember: All type calculations
  - finally result in implementation selection
  - are an and in itself only in simple demo programs

Example (cont.): type_calculation/demo.cpp

Standard Type Transformations

- Frequently required transformation are provided
  - as part of the standard type traits
  - especially for *Iterators* via `std::iterator_traits`

---

http://en.cppreference.com/w/cpp/header/type_traits
(From section *Const-volatility specifiers* to section *Miscellaneous transformations*)

http://en.cppreference.com/w/cpp/iterator/iterator_traits

Example (cont.): type_calculation/demo.cpp

Compile-Time Value Calculations

Example: value_calculation/demo.cpp

Template Meta-Programming vs. `constexpr` Functions

Example (cont.): value_calculation/demo.cpp

Dealing with Values as Types

Example (cont.): value_calculation/demo.cpp

Type Traits

Example: type_traits/demo.cpp

Storing Compile-Time Data

- Think of a database organized
  - in columns (kind of data)
  - and rows (accessed by key)
- Row is selected by a type
- Kind of data is detail (information) about this type

Access by (`struct`) Name

- There may be **one** meta function for **each** columns
  - Type (to be used as "key") goes into the argument list
  - Kind of information is determined by the (function) name
- (somewhat like specific functions in Run-Time programming)
- Examples from standard type traits:
  - (too many to list)

http://en.cppreference.com/w/cpp/header/type_traits

Example (cont.): type_traits/demo.cpp

Selection by (`struct`) Member

- There may be **one** meta function for **all** columns
  - Type (to be used as "key") goes into the argument list
  - Kind of information determined by "calling convention"
- (somewhat like member functions in Run-Time programming)
- Examples from standard type traits:
  - `std::iterator_traits`
  - `std::numeric_limits`
  - (and many more)

http://en.cppreference.com/w/cpp/iterator/iterator_traits#Member_types

http://en.cppreference.com/w/cpp/types/numeric_limits#Member_constants

(More) Standard Traits

- C++11 defines many more standard type traits
  - Purpose of many can be derived from their name
  - Nevertheless lookup docs to avoid misinterpretation
- When used indirectly (in a template)
  - for accessing the …`::`type-member …
  - … be sure to prepend `typename`

http://en.cppreference.com/w/cpp/header/type_traits

Example (cont.): type_traits/demo.cpp

Abbreviating Type Access in C++14

- In addition to the ...`::type` calling convention ...
  - ... C++14 adds type aliases ending in `_t`
  - (reduces noise by avoiding `typename ...::type`)

http://en.cppreference.com/w/cpp/header/type_traits
(see `using` type aliases ending with `_t`)

Example (cont.): type_traits/demo.cpp

Abbreviating Value Access in C++14

- In addition to the ...`::value` calling convention ...
  - ... C++14 adds
    - `constexpr` constructor and `operator bool()`
    - (reduces noise by avoiding access to ...`::value`)
  - ... C++1z adds
    - template variable ..._v
    - (reduces noise by avoiding access to ...`::value`)

http://en.cppreference.com/w/cpp/header/type_traits
(all `is_`... traits)

Example (cont.): type_traits/demo.cpp

Applying SFINAE

- SFINAE = Substitution Failure Is Not an Error
  - Selects overload on finer-grained criteria
- Strategy
  - Specify an ambiguous overload set
  - Make all instantiations **except one** fail
- Common idiom
  - Comma separated expression in `decltype` as function result

http://en.cppreference.com/w/cpp/language/sfinae

std::enable_if

- May be considered a "more readable" alternative
  - First argument is condition (Compile-Time evaluated `bool`)
  - Second argument is a type (defaults to `void`)
- Typical use:
  - C++11: `… typename std::enable_if< …cond… , …type… >::type …`
  - C++14: `… std::enable_if_t< …cond… , …type… >`
- Expands to
  - `…type…` if `…cond…` is true
  - substitution failure otherwise (but: SFINAE!)

http://en.cppreference.com/w/cpp/types/enable_if

SFINAE for Functions

- Applying SFINAE to functions can be done via
  - ... function return type
  - ... extra argument
  - ... template value parameter
  - ... template type parameter

Example: function_sfinae/demo.cpp

SFINAE via Return Type

- Use `enable_if` as function return type
  - either in classic syntax (left) to function name
  - or in trailing return type syntax
- Probably the "easiest" and mostly used style
  - but impossible for constructor and destructor

Example (cont.): function_sfinae/demo.cpp

SFINAE via Extra Argument

- Add an extra argument (not accessed in the function)
  - Define its type with `enable_if`
  - Supply default value
- Still "relatively" straight forward
  - but impossible for overloaded operators

Example (cont.): function_sfinae/demo.cpp

SFINAE via Template argument

- Via template type argument:
  - Add unused last template argument
  - Set default with `enable_if` to type `void`
- Via template value argument:
  - Define as `void *` (using `enable_if` for `void`)
  - Set default with `enable_if` to `nullptr`
- More "arcane" and probably harder to understand

SFINAE for Classes

- SFINAE may be used to specialise class templates
- Strategy:
  - Add trailing (otherwise unused) `typename` argument
  - Initialize with default (e.g. void)
  - Specialisation make initialisation succeed or fail

Example: class_sfinae/demo.cpp

Variadic Templates

Defining Parameter Packs

- Parameter packs are defined by adding an ellipsis (three dots)
  - e.g.: `template<typename... Ts>` ...
  - or: `template<int... Vs>` ...
- Value lists may refer to a typename mentioned earlier, e.g.
  - e.g. `template<typename T, T... Vs>` ...
- **Pack must come last in the template argument list**

http://en.cppreference.com/w/cpp/language/parameter_pack

Example: parameter_pack/demo.cpp

Number of Elements in Parameter Pack

- `sizeof...` is a Compile-Time function returning the pack size
  - e.g. `sizeof...(Ts)`
  - or `sizeof...(Vs)`

Do not confuse `sizeof...` (which returns the element count) with classic `sizeof` (which returns the number of bytes in memory).

Emulating Parameter Packs

- Possible to some degree
- Requires much systematic code
  - Boresome and error prone to write
  - Difficult to understand
  - Hard to maintain
  - Inefficient to compile

Compile-Time Only Data

- Parameter packs can represent "pure" Compile-Time data
  - e.g. a list of integral values
    - only existing in the compiler memory
    - not in the programs executable

Note: There are practical applications but the following example should rather demonstrate the idea only:

Example (cont.): parameter_pack/demo.cpp

Compile-Time Type Lists

- Often parameter packs represent lists of types
  - Such lists may only exist at Compile-Time ...
  - ... but also may be used to define Run-Time data

A practical application may be to select the best matching type from a list of given types.

Example (cont.): parameter_pack/demo.cpp

Boost MPL

- Meta-Programming Library, supporting
  - STL-like Compile-Time data structures
  - STL-like Compile-Time algorithms

http://www.boost.org/doc/libs/release/libs/mpl/doc/index.html

Example (cont.): parameter_pack/demo.cpp

Adding Run-Time Data to Type List

- Run-Time Data may be based on Type Lists:
  - Mainly intended for type-safe variadic functions
  - Variadic Class (data-) members also possible
    - e.g. `std::tuple`
    - or `boost::variant`
- Key is "unpacking" a parameter pack with `...`
  - usually combined with Compile-Time recursion

Type-Safe Variadic Functions

Example: variadic_function/demo.cpp

Member Data According to Type List

- To add class member data according to a type list ...
  - ... use a recursive data structure
  - or write many (many, many ...) specialisations

Writing some (few) specialisations may serve as good starting point to recognice a recursively repeating structure (data or code).

Example: my_tuple/demo.cpp

Unpacking Parameter Packs

- Unpacking a parameter pack is requested by an ellipsis (three dots)
  - The ellipsis occur to the right of an expression
  - The expression needs to contain at least one parameter pack
  - It is repeated by substituting elements from the pack
- Valid unpacking contexts are:
  - Function call parameter – **no** left to right guarantee
  - Aggregate initialisation – **left to right guarantee**
- More that one parameter pack can be unpacked simultaneously

Example (cont.): my_tuple/demo.cpp

Fold expressions

- C++1z generalises unpacking parameter packs
  - Syntactically this is called fold expressions
  - Avoids utilising side effects of initialisation

Example (cont.) my_tuple/demo.cpp

Concepts Light

- Concepts can be thought of as
  - Constraints for template arguments
  - One of the main intents is to improve error messages

http://en.cppreference.com/w/cpp/concept

Currently, if a template instantiation fails, the root cause of the problem often is everything but clear to the compiler:

Usually the client (that instantiated the template) has created the problem by using a type that is not a model of the expected concept. But because the manifestation of the problem is in the instantiated template code – **frequently not authored by the client using it** – the error message gives no good idea what actually went wrong.

Furthermore – in an attempt to be helpful – the error message often reveals (too) much of the internal state the compiler had when discovering the problem and is more confusing as enlightening.

Basic Idea

- In a template argument list
  - Use name of a "concept" instead just `typename`
  - Define requirements for models of concept elsewhere
- In case of compile errors:
  - Blame client by referring to point of instantiation
  - Instead showing code from inside the template implementation

---

https://groups.google.com/a/isocpp.org/group/concepts/attach/df9a57cb574ae9f9/lite.pdf?part=4&authuser=0

Current State

- Still experimental – available as compiler extension only:
  - ???