Advanced C++ Programming

Advanced Templates

Preliminaries

Overview & Goals

- This chapter introduces template metaprogramming and a few other advanced template-related concepts
- Generally, we need to use these concepts in three cases:
  - Writing **more flexible** and generic code
  - Writing **faster** code
  - **Understanding** existing code (e.g. the standard library, or boost)

Parts of this chapter are based on
"Modern Template Metaprogramming: A Compendium"
by Walter E. Brown

“Template Metaprogramming”

- What is **Metaprogramming**?

  Writing a program which manipulates other programs (or itself) as its data, or performs computations at compile time.

- **Template** metaprogramming uses template instantiation (recall chapter 4) to drive compile time evaluation
- A basic example is shown in 06_01_simple_template_metafunction.cpp
  - This particular example could also be accomplished with a `constexpr` function

General Mindset

- Template metaprogramming is similar to *pure functional programming*
- In particular, this means…
  - **No mutability**
  - Nothing that depends on runtime behaviour, e.g. virtual dispatch
  - Recursion instead of loops, pattern matching instead of conditionals

Operating on Types

Types as Parameters

- Metafunctions can take types as their parameters
  - … because types are possible template parameters
- Example of a built-in function that takes a type parameter…?

  `sizeof()`

- We can implement our own with template metaprogramming!
  06_02_type_parameter.cpp shows how to implement a `dimof<>` for arrays

Types as Results

- To really operate on types, we need to be able to produce them as results
- This can easily be accomplished with **aliases** (either by `using` or `typedef`)
  - E.g. we create an alias member “`type`” which contains the result
    (this is the general convention also used in the standard library)
- In 06_03_type_result.cpp we implement a simple example of this principle

Refactoring & Conventions

- Metaprograms *are* ***programs***

→ We can refactor them and apply good coding practices

- The code example 06_04_refactoring_conventions.cpp demonstrates simple refactoring on our previous samples

  Note the convention:

  - “**_t**” for **alias** templates referring to the **::type** member
  - “**_v**” for **variable** templates referring to the **::value** member

Metaprogramming Implementation Strategies

And Their Underlying Principles

Mapping Constructs to Template Metaprograms

- We've already seen several mappings:
  - Return values → static member values (`::value`) *or* member aliases (`::type`)
  - Loops → template instantiation recursion (e.g. `dimof<>`)
  - Conditionals → distinct specializations (e.g. `remove_const<>`)
- Let's look at another example to get more experience with these

Practice

- The standard library includes a variadic template type **`tuple<…>`**
- Let's say we want to create a metafunction **`includes_type<U, T>`** which returns true if the tuple **U** includes the type **T**
- We want to do this from scratch without using any library (meta-)functions
- An implementation of this is shown in 06_05_tuple_includes.cpp
  - It still has a bit of a niggle: we can call it for non-tuples and won't get a compiler error

Refactoring and Error Handling

- How can we stop this implementation from compiling for non-tuple types?
- Can we improve the error message?
- Result in 06_06_tuple_includes_prime.cpp

➲ Try to use `static_assert()` whenever applicable to improve the user experience for your template code

Conditionals using Template Specializations

- We use template specializations to implement case distinctions/conditionals in metaprogramming
- How does this work?
  How does the compiler know which specialization to choose?
- Intuitively, it should use the "*most specialized*" version
- This intuition is encoded using a **partial order** on template specializations

Partial Ordering on Template Specializations

- Described in the C++17 standard in 17.5.6.2
- Intuitively: *a template is more general (>) if it can match on any instantiation of a less general (more specialized) template*

```
T > T[N] > int[N] ~ T[8]

T, U<V> > T, U<char> > T, std::vector<char>
```

Practical Metaprogramming

SFINAE and Unevaluated Contexts

The Type-based Dispatch Challenge

- Assume we use many types in our program, from different libraries
  - Some implement an output stream operator for printing
  - Others have a `print_to` member function
- We want to implement a function that can deal with all of them
  - E.g. for debugging or logging

We'll investigate this problem in **06_07_dispatch.cpp**

What is this Sorcery?

```cpp

template<typename T, class = void>
struct has_print_to : public std::false_type {};

template<typename T>
struct has_print_to<T,
    std::void_t<decltype(std::declval<T>().print_to(std::declval<std::ostream&>()))>
> : public std::true_type {};

```

- It’s rather dense, but each component can be understood easily in isolation

std::declval<T>()

- `std::declval<T>()` returns an instance of type T
  - An r-value by default, can generate an l-value by using `declval<T&>()`
- Can even be used if the type has no (publicly available) default constructor
- *It doesn't have an implementation/definition*
  - Specifically designed to be used in **unevaluated contexts**
  - E.g. in template metaprogramming

So what is an unevaluated context?

Unevaluated Contexts

- A context in which an expression is not actually evaluated (i.e. executed)
- 4 cases:
  - **`sizeof`**`(expr)` – oldest, we know this one
  - **`noexcept`**`(expr)` – checks whether *expr* can throw an exception
  - **`typeid`**`(expr)` – yields a std::type_info object for the type of *expr*
    NOTE: only unevaluated if there is no polymorphism!
  - **`decltype`**`(expr)` – The type of the expression *expr*

Examples for decl* in **06_08_decltype_declval.cpp**

Putting it into Practice

```
template<typename T, class = void>
struct has_print_to : public std::false_type {};

template<typename T>
struct has_print_to<T,
    std::void_t<decltype(std::declval<T>().print_to(std::declval<std::ostream&>()))>
> : public std::true_type {};
```

- We now understand this part: *in an unevaluated context, an instance of T is created, and "print_to" is called on it with an l-value of type std::ostream*
- **But why?**

The Selection Mechanism

```
template<typename T, class = void>
struct has_print_to : public std::false_type {};

template<typename T>
struct has_print_to<T,
    std::void_t<decltype(std::declval<T>().print_to(std::declval<std::ostream&>()))>
> : public std::true_type {};
```

- We are again using the “more specialized” template mechanism to make case distinctions
- But with an additional twist: **SFINAE**

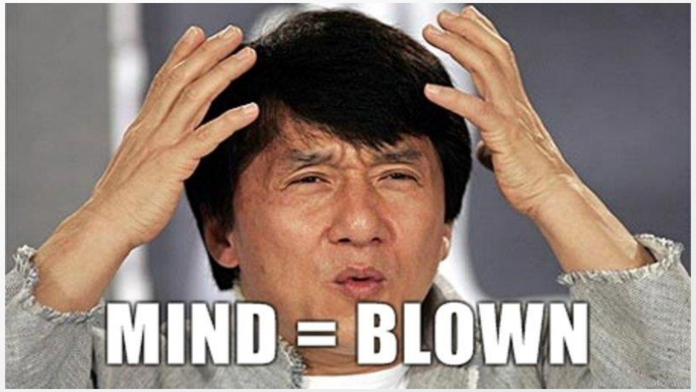
Substitution Failure Is Not An Error

- Principle that applies when the compiler instantiates templates (and their specializations)
- When substitution of a template Parameter leads to erroneous code, that particular specialization is *removed from the overload set*, rather than creating a compiler error
- The general usage is to **create an error on purpose in some circumstances to remove an implementation from the overload set**

Example in **06_09_sfinae.cpp**

So what is `std::void_t`?

```
template< class... >
using void_t = void;
```



“This metafunction is used in template metaprogramming to detect ill-formed types in SFINAE context” - cppreference

→ If any of the argument types is ill-formed, it’s an error
→ Otherwise, it is simply a fancy way to write “void”

```
template<typename T, class = void>
struct has_print_to : public std::false_type {};

template<typename T>
struct has_print_to<T,
    std::void_t<decltype(std::declval<T>().print_to(std::declval<std::ostream&>()))>
> : public std::true_type {};
```

**Note*: the second template parameter needs to default to void for the specialization to work

- **If** the type T has a member function with the desired signature
  - `void_t<…>` in the specialization is void
  - The specialization is less general and is chosen → **`true_type`**
- **otherwise,**
  - The type passed to `void_t` is ill-formed → `void_t` causes an error
  - The specialization is dropped due to *SFINAE*
  - The primary template is chosen → **`false_type`**

Curiously Recurring Template Pattern

Curiously Recurring Template Pattern (CRPT)

- Not Metaprogramming, but an advanced template ***idiom***
- Derived classes inherit from a base specialized with *themselves:*

```
template<class T>
class A {
  // methods within A can use template
  // to access members of derived classes
};
class B : public A<B> {
  // ...
};
```

CRTP Usage Scenarios

- Useful whenever bases want to customize operations of derived classes, or the other way around
- E.g.
  - Static Polymorphism
  - Implementing special semantics (e.g. Singleton)
  - Implementing metainformation/logging/tracking (e.g Instance counter)

https://github.com/nitingupta910/crtp_bench

Example in **06_10_crtp.cpp**

Conclusion

Summary

- Template Metaprogramming allows us to
  - Compute results at compile time
  - Operate on Types as arguments and return values from our metafunctions
- Implementation Strategies
  - Specializations for case distinctions, recursive instantiations to loop
- Language Principles Required
  - Partial Ordering on Specializations
  - SFINAE