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 less general (more specialized) template*

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

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