C++ London University Session 12 (part 2)

Tristan Brindle

Feedback and Communication

- Your feedback is vital
- Otherwise, we don’t know what you don’t know!
- Please join the #ug_uk_cpplondonuni channel on the cpplang Slack — Go to https://cpplang.now.sh/ for an “invitation”
- We’ve also got a Subreddit! reddit.com/r/CppLondonUni

Why templates?

- Templates allow us to write code that is *generic*, *type-safe*, and **highly efficient**

- C++ templates look similar to generics in languages like C# and Java. However, the way they work is quite different

- A template is like a “blueprint” that instructs the compiler how to write a given class or function

- When we *instantiate* a template with particular *template arguments*, the compiler creates a new, distinct type (or function) for those arguments

Kinds of templates

- Three kinds of entities can be templated:
  - Function templates
  - Class templates
  - (Since C++14) variable templates
- Today I'll be talking mostly about class templates

Declaring templates

- To declare a template, we introduce it with the `template` keyword, followed by the *template parameters* in angle brackets (< >). There is no practical limit on the number of parameters that can be used.

- For example:

```
template <typename T>
void my_function(T argument);
```

```
template <typename T>
class my_class {
    // ...
};
```

- Here, T is the template parameter, and refers to a *type*. Within the function body, we can use the name T as if it referred to a real type like `int`.

- When we *instantiate* the template with a concrete type, the compiler substitutes that type name wherever T appears in the body

- The name T acts like a placeholder for the real type

Template parameters

- Formally, are three *kinds* of template parameters
  - Type template parameters
  - Non-type template parameters
  - Template-template parameters
- Today we'll only be talking about *type template parameters*, which are the most common
- We can introduce a *type template parameter* with either the `class` or `typename` keywords; in a template parameter list, they mean the same thing. For example:

```
// These are the same template
template <typename T>
void my_function(T argument);

template <class T>
void my_function(T argument);
```

Templates go in headers

- When we write a template, we usually don’t know what types it will be instantiated with

- This means that in general, the entire definition of a template must live in a header file

- This is because the compiler needs to have the body available in order to do type substitution — *in order to build something, you need the blueprints!*

- If we instantiate the same template with the same arguments in multiple .cpp files, then the linker will ensure that only one copy exists in the final executable

- Another way of putting it is that function templates are implicitly inline

Using templates

- We can use (*instantiate)* a template by writing its name, followed by the template arguments in angle brackets

- For example

```
template <typename T>
class my_class {
    T member;
};

my_class<int> m;
```

- This tells the compiler to create a version of `my_class` with `int` substituted for T

- Function templates can use *template argument deduction* to save you having to write out the template parameters explicitly

- In C++17, a similar feature exists for constructors of class templates

Exercise

- https://github.com/CPPLondonUni/templates_exercises

- Please complete Exercise 1

Bonus Slide: dependent names

- We can access a member type or a static member variable of a class with the syntax

```
struct example {
    static int i = 0;
    using type = float;
};

example::i = 3; // example::i is a value
example::type f = 3.0f // example::f is a type
```

- But when the class name is a template parameter, we don't know whether the name `T::something` refers to a *type* or a *value*

- This can lead to ambiguous syntax, for example

```
template <typename T>
struct example2 {
    T::x * y; // multiplication or pointer declaration?
};
```

- To solve this the compiler requires that we say `typename` when referring to dependent member types, for example

```
template <typename T>
struct example2 {
    typename T::x variable{}; // Now we know T::x is a type
};
```

Writing member functions

- Recall that when we write the definition of a class member function outside of the class, we need to write

```
ReturnType ClassName::member_name(/*arguments…*/)
{
    // Implementation...
}
```

- If we write the definition of a class template member function outside of the class, we need to use the template keyword to introduce the parameter names, for example

```
template <typename T, typename U>
ReturnType ClassTemplate<T, U>::member_name(/*arguments...*/)
{
    // Implementation...
}
```

Exercise

- https://github.com/CPPLondonUni/templates_exercises

- Please complete Exercise 2

Specialisations

- After writing a template, it may be the case that we want special behaviour for particular template arguments

- For example, we may be able to use a compressed storage representation for storing bools…

- To do this, we are allowed to supply *specialisations* for particular argument types

Explicit Specialisations

- To write an explicit (full) specialisation, use the syntax

```
template <typename T>
class ClassTemplate {
    // General version
};

template <>
class ClassTemplate<int> {
    // Specialised version for ints
};
```

- The base template and any specialisations are completely separate as far as the compiler is concerned; they don’t have to have the same types, members, member functions etc…

Explicit Specialisations

- Class templates, variable templates and function templates can all have explicit (full) specialisations

- However, it's usually a bad idea to specialise function templates: prefer to provide multiple overloads instead

- A specialisation can only be introduced after the compiler has "seen" the base template

- *"When writing a specialization, be careful about its location; or to make it compile will be such a trial as to kindle its self-immolation"* — ISO C++ Standard [17.7.3]

Exercise

- https://github.com/CPPLondonUni/templates_exercises

- Please complete Exercise 3

Partial Specialisations

- Class templates and variable templates can also have *partial specialisations*
- For example, we can use partial specialisation when a template has multiple parameters, but only some of them need to be specialised
- A partial specialisation is itself another template!

```
template <typename T, typename U>
class ClassTemplate {
    // General version
};

template <typename T>
class ClassTemplate<T, int> {
    // Partial specialisation for U = int
};
```

Partial Specialisations

- We can also use partial specialisation to match arguments which fit certain patterns. For example

```
template <typename T>
class ClassTemplate {
    // General version
};

template <typename T>
class ClassTemplate<T*> {
    // Partial specialisation for pointers of any type
};

template <typename T>
class ClassTemplate<std::vector<T>> {
    // Partial specialisation for vectors of any type
};
```

Partial Specialisations

- It’s possible for a given set of arguments to match multiple partial specialisations
- In this case, the “most specialised” version will be chosen
- In particular, a matching full specialisation will always be preferred to a partial specialisation

Exercise

- https://github.com/CPPLondonUni/templates_exercises

- Please complete Exercise 4

“Homework”

- Finish Exercise 4

- If you find it all too easy and have a very up-to-date compiler, try exercise 5.

Online Resources

- https://isocpp.org/get-started
- cppreference.com — The bible, but aimed at experts
- cplusplus.com — Another reference site, also has a tutorial section
- learncpp.com — Free online tutorial, very up-to-date
- https://www.pluralsight.com/authors/kate-gregory - Comprehensive set of courses from an experienced C++ trainer (free trial)
- reddit.com/r/cpp_questions
- Cpplang Slack channel — https://cpplang.now.sh/ for an “invite”
- StackOverflow (but…)

Thanks for coming!

C++ London University:

- Website: cpplondonuni.com
- Twitter: @cpplondonuni
- Github: github.com/CPPLondonUni
- Reddit: reddit.com/r/CppLondonUni

Where to find Tom Breza:

- On Slack: cpplang.slack.com #learn #ug_uk_cpplondonuni
- E-mail: tom@PCServiceGroup.co.uk
- Mobile: 07947451167

My stuff:

- Website: tristanbrindle.com
- Twitter: @tristanbrindle
- Github: github.com/tcbrindle

See you next time! 🙂