C++ London University Session 13

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”

Today's Lesson Plan

- Brief revision of last week's session
- More about templates!
  - Class template specialisation
  - Template argument deduction
  - The Curiously Reoccurring Template Pattern

Previously on C++ London University…

- Templates are a C++ feature enabling *compile-time polymorphism*

- A template is like a “blueprint” for the compiler, instructing it how to write a class, function, variable or alias

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

- Template definitions must generally go in header files

Previously on C++ London University…

- We *declare* a template using the `template` keyword, followed by the *template parameters* in angle brackets. For example:

```
template <typename T>
struct my_template {
    // ...
};
```

- We *instantiate* a template by writing its name followed by the *template arguments* in angle brackets. For example:

```
my_template<int> m; // instantiate my_template with T = int
```

- A template parameter can be a type name, a value (with certain restrictions), or another template name

- Function templates can deduce the template arguments from the function arguments (as can class template constructors in C++17).

Previously on C++ London University…

- Any questions about last week’s material?

Template Specialisation

- 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 Specialisation

- 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 Specialisation

- Class templates, variable templates and function templates can all have explicit (full) specialisations, as can member templates of classes and class templates

- 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

- If you didn’t complete Exercise 1 and 2 last week, you can find the code in the `solutions_ex2` branch

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

Explicit Instantiation

- Normally, a template is instantiated when it is first used
- However, we can tell the compiler to explicitly instantiate a template using the syntax

```
template <typename T>
void my_function(T arg) {
    // ...
}

template void my_function(int i); // explicit instantiation with T = int
```

- Note that this is very similar to the syntax for specialisation!

```
template void my_function(int i); // explicit instantiation with T = int

template<> void my_function(int i); // explicit *specialisation* for T = int!
```

- An explicit instantiation is not a template: it is a declaration of a “real” function or class.

extern template

- Normally, if we use a template in multiple implementation files, the compiler will instantiate the template as it compiles each file. The linker will then merge the definitions with constructing the final executable

- This means that work is being duplicated as we compile each file: the same instantiation is being done over and over again, only for all but one of the copies to be thrown away at link time!

- To avoid this, we can use `extern template` to say to the compiler "don't fully instantiate this template, just trust me that it's happened elsewhere". For example:

```
// don't instantiate vector<int> in this TU
extern template class std::vector<int>;

std::vector<int> vec{1, 2, 3};
```

Template Argument Deduction

- When calling function templates, we can supply the template arguments as normal
- If we supply fewer arguments than there are parameters, the compiler will attempt to deduce the remaining template arguments

```
template <typename T, typename U>
void my_function(T t, U u);

my_function<float>(1, 2); // T = float (supplied), U = int (deduced)
```

Template Argument Deduction

- If two different arguments would lead to different types being deduced, then this leads to a *deduction failure*

- In this case, we need to supply the template parameter explicitly. For example:

```
template <typename T>
T max(T first, T second)
{
    if (second > first) {
        return second;
    }
    return first;
}

auto m = max(1, 2.0f); // deduction failure, int vs float

auto m2 = max<float>(1, 2.0f); // ok
```

Template Argument Deduction

- Argument deduction can only happen if the template parameter appears in the function's parameter list

```
template <typename T, typename U>
void do_something(T arg);

do_something(3); // error: could not deduce template param U
```

- Templates can also have default parameters. For example, the full signature of `std::vector` is

```
template <class T, class Allocator = std::allocator<T>>
class vector;
```

Template Argument Deduction

- Deduction can happen via references and pointer arguments…

```
template <typename T>
void func_taking_ref(const T& arg);

template <typename T>
void func_taking_ptr(T* ptr);

int i = 0;
func_taking_ref(i); // T deduced as int
float f = 1.0f;
func_taking_ptr(&f); // T deduced as float
```

- …and also when using function arguments that are themselves templates

```
template <typename T>
void func_taking_vector(const std::vector<T>& vec);

std::vector<int> v{1, 2, 3};
func_taking_vector(v); // T deduced as int
```

Deduction Playground

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

Advanced Templates: CRTP

- CRTP stands for the “curiously recurring template pattern”

- It’s the name given to a pattern where a derived class inherits from a base class which is templated on the derived class (!). For example:

```
template <typename T>
class Base {
    // …
};

class Derived : public Base<Derived> {
    // ...
};
```

- It is most useful for making *static* polymorphism (with templates) closely resemble *dynamic* polymorphism (with virtual functions).

Advanced Templates: CRTP

- In this pattern, the base class defines the *interface* (just like with dynamic polymorphism)

- The derived classes provide the *implementation*

- Rather than using virtual dispatch, functions are bound *at compile time* — with zero overhead

CRTP Example

- Please go to

https://github.com/CPPLondonUni/crtp_example

That’s all folks

- “Homework”: finish the polymorphism examples

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! 🙂