Templates

Warwick RSE

28/09/2023

Concept

What are templates?

- Sometimes you have code that is used in similar ways but with a few differences
  - For example doing the same thing to different types of data
  - You want to avoid writing the same code multiple times
- This is classically done in many languages with the use of tricks with a preprocessor
  - Runs before your code is compiled and alters the text that is compiled
- The idea of the template system is to allow you to do things like that in a formal, consistent way
  - Is still a preprocessor, but one that is very powerful and actually intended for this type of work

Simple Example

Simple motivation

- In C++ it can be annoying to have to remember to put "\n" at the end of every **cout** statement
- Why not write a function that works like **print** statements do in other languages and automatically puts a newline in?
- Well, most obvious reason is that the compiler has to know what type something is to be able to print it
  - Remember that to the computer an **int** and a **float** are both just 4 bytes of binary data it **needs** that type information to know that they should be printed differently

Simple motivation

```
int main(){
  print("Hello world!");
  print(14);
  print(1.234);
}
```

- Imagine that we want to have code like this that will just work
- Each of those things printed and then a newline after them
- We have already seen one way to do that
  - Function overloads

Simple motivation

```
void print(int i){
 std::cout << i << "\n";
}

void print(double d){
  std::cout << d << "\n";
}

void print(std::string s){
  std::cout << s << "\n";
}
```

- This will work
- Three functions, one for each of the “major” types
- Each prints their contents and then a newline

Result!

```
Hello world!
14
1.234
```

Simple motivation

```
void print(long int i){
 std::cout << i << "\n";
}

void print(double d){
  std::cout << d << "\n";
}

void print(std::string s){
  std::cout << s << "\n";
}
```

- Now try this version with our driver code
- The line **print(14);** is now (usually) ambiguous since it is neither a **long int** nor a **double**
- Compiler doesn't know which to pick as being "better" so it won't compile
- Have to have both **int** and **long int** etc. etc. - rapidly runs out of control

What we actually want

- The body of **all** of these functions will be exactly the same
- For a given type either we call
  - **std::cout << value << "\n";**
- or we don't know how print it and compilation should fail
- So, why can't the compiler generate the code itself?
- That is where templates come in

Template

```
template <typename T>
void print(T value){
  std::cout << value << "\n";
}
```

- Believe it or not, that's it
- That makes a print function that can print any type that could be printed with <<
- Let us look at the things bit by bit

Template

```
template <typename T>
void print(T value){
  std::cout << value << "\n";
}
```

- The word "template" says that you are creating a template
- Always goes before the "thing" that you are templating
- Only affects the next "thing"

Template

```
template <typename T>
void print(T value){
  std::cout << value << "\n";
}
```

- Triangular braces (**< >**) indicate parameters to a template just like round brackets indicate parameters to a function

- They are used both when defining a template and when specifying a specific entity (specifying T manually) to a template

  - We'll come to that later, but mostly you don't need to do that for functions

Template

```
template <typename T>
void print(T value){
  std::cout << value << "\n";
}
```

- **typename** says that the parameter to this template is a type
  - I.e. I am using this template to generate code that is different for different types
- There is an older synonym keyword **class**
  - You will still see **class** used here commonly, but **typename** is considered more correct

Template

```
template <typename T>
void print(T value){
  std::cout << value << "\n";
}
```

- **T** is a name for the type that we are templating on
  - This is just like the name of a parameter to a normal function - it is a name to let you use the parameter and is arbitrary but must be unique in a template
- **typename T** is actually **very** common in real code for simple templates

Template

```
template <typename T>
void print(T value){
  std::cout << value << "\n";
}
```

- **T** here is used as a type specifier, like **int**, **float**, **std::string** etc would be
- The **T** is obviously the name of the type from the template parameter and should be changed if you use a different parameter name
- **T** can be used anywhere within your function, so if you want to create a variable with the same type as your parameter you just type
  - **T tempvar;**

What happens?

- Now when the compiler encounters a call to the function **print** it will look at the type of the argument that the function is called with

- If it is the first time that it has encountered a call with that parameter type it will **generate** a function with a type matching the type that the function is called with

  - If it has already generated a function for a type then it will reuse it

- The function with the correct type will be used

- No worries about **int** vs. **long int** - it will generate a function for each of them!

Further templates

```
template <typename T>
void print(T value1, T value2){
  std::cout << value1 << " " << value2 << "\n";
}
```

- What about if I wanted to have a version of print that took two parameters?
- If both parameters are of the same type then nothing different at all
  - Two parameters both type **T**
- All of the normal rules for overloaded functions apply, so you can call that function **print** as well since it is distinguished from the others by having two parameters

Further templates

```
template <typename T1, typename T2>
void print(T1 value1, T2 value2){
  std::cout << value1 << " : " << value2 << "\n";
}
```

- What about two differently typed parameters?
- No problem. Two template parameters
- Each template parameter is considered independently
- You can have as many templates parameters as you want
    - There is a limit somewhere which is compiler dependent but you probably won't run into it

Template Selection

- So how are templates selected?
- That is actually a hard question!
  - In the details at least
- Essentially the template engine selects the “lest general” template that will work
  - What do I mean by least general?

Another **main**

```
int main(){
  print("Hello", "World");
  print("Test value is",42);
}
```

- Consider the above main function of two print calls with two parameters
  - One has two strings, one a string and an integer
- Of our earlier two parameter print functions
  - The one taking two parameters of different types will work fine
  - The one taking two parameters of the same type will fail on the second line
- What will happen if we have **both** template functions in the same code?

Two templates

```
template <typename T>
void print(T value1, T value2){
  std::cout << "Two matched parameters\n";
  std::cout << value1 << " " << value2 << "\n";
}

template <typename T1, typename T2>
void print(T1 value1, T2 value2){
  std::cout << "Two different parameters\n";
  std::cout << value1 << " : " << value2 << "\n";
}
```

- If you think back to out earlier approximate rule you get
- **string,string** will call the first one
- **string,int** will call the second one
- Is that right?

Yep!

```
Two matched
Hello World

Two different
Test value is : 42
```

- In fact, C++ will go through and select the first function that matches the parameters going up in genericness

Template Selection

- Non-templated functions that are an exact match to the specified parameters
- Non-templated functions where the parameters can be converted to be of matching type
- Specialised templates (Special implementations of functions for specific values of template parameters)
- Partially specialised templates (Where you create a special implementation where some, but not all of the template parameters have specific values)
- Templated functions going from fewest to most template parameters
- Variadic templates (Templates with variable numbers of parameters)

Chose Later

Advice for Templates

- Be careful with templated functions - you can destroy your mind
  - Actually the same with just overloads
- If two functions have the same name they should do the same job
  - Sometimes the code to do the same job might look different but it should be the same job
- Try to make sure that you know what is going to happen for any call to a function that you might make

Already seen templates

- Those **< >** brackets should look pretty familiar from our STL slides
- The STL containers are indeed templated on what they are going to store
- They are templated **classes** rather than templated **functions**
  - We're going to come back to them
- But can I use templating to pass an arbitrary vector to a function?
  - Yes! Just like you can use a function parameter to call another function, you can use a template parameter as a parameter to another template

Template to template

```
template <typename T>
void print(std::vector<T> &vec){
  for (auto &el:vec){
    std::cout << el <<"\n";
  }
}
```

- This function is templated on a type T
- That T is then used as a template parameter to **std::vector**
- This means that you have a function that takes a vector storing anything

More advanced template

- What about if we wanted to do some kind of adaptor that gets an element from a vector, and returns either it or zero?
  - You'd probably actually just write a normal function that takes a value and checks it, but this is *sort of like* a real thing that people do
- This introduces a couple of new elements
  - Non template parameters to template functions
  - Template return types

More advanced template

```cpp
#include <iostream>
#include <string>
#include <vector>

template <typename T>
T check_and_return(std::vector<T> &vec, int index){
  T value = vec[index];
  if (value > 0) return value;
  return 0;
}

int main(){
  std::vector<int> v;
  for(int i=0;i<10;++i) v.push_back((i-5));
  for(int i=0;i<10;++i) std::cout << check_and_return(v,i) << "\n";
}
```

- Template parameters are just normal parameters
- Just put your other parameters in as well

More advanced template

```
#include <iostream>
#include <string>
#include <vector>

template <typename T>
T check_and_return(std::vector<T> &vec, int index){
  T value = vec[index];
  if (value > 0) return value;
  return 0;
}

int main(){
  std::vector<int> v;
  for(int i=0;i<10;++i) v.push_back((i-5));
  for(int i=0;i<10;++i) std::cout << check_and_return(v,i) << "\n";
}
```

- **T** really is just a type specifier as used here
- You can use it when defining variables
- You can use it in return types

Template **inference**

- In all of these functions the use of the templates is seamless
  - Apart from when we are specifying **std::vector**
- This is because of **inference**
- Because we are using the templates to produce parameters to functions the compiler can **infer** the template types from the type of the parameters that we are passing to the function
- **std::vector** is a templated class, so there are no parameters that it can use to infer the type that it is templated on so you have to specify them manually
- This is also true for a function that only uses the template parameter for the return type

Templated return

```
template <typename T>
T ten_over_three(){
  return T(10)/T(3);
}

int main(){
  //Set the code to print to 15dp of precision for floating point numbers
  std::cout << std::setprecision(15);
  std::cout << "10/3 as integer is " << ten_over_three<int>() << "\n";
  std::cout << "10.0/3.0 as float is " << ten_over_three<float>() << "\n";
  std::cout << "10.0/3.0 as double is " << ten_over_three<double>() << "\n";
```

- Write the function the same as before
- Now the compiler can't infer the template type so you have to put it in manually
  - Can't try to infer it from return type because I may be using it in a way that doesn't give clues
- **<type>** before the function call **( )**

Consolidation

- Templates
  - Allow you to write code in terms of arbitrary types that you either specify later or the compiler will automatically determine them if possible
  - Can be used for parameters to functions, variables within functions and return types for functions
  - Can be passed to other templates (i.e. **std::vector<T>**)

Further Templates

Further Templates

- Template parameters can be types other than **typename**s
  - Before C++20 mainly integer types
  - C++20 and later almost any type
- Templates can have default values, in which case if type inference fails they are given the default value
- You can write **variadic** templates that take an unknown number of template parameters - they are tricky to work with so avoid them until you need them!

Further Templates

- You can **specialise** a template by providing values for template parameters and then a custom implementation of the function or class for those values of the parameters
  - The specialised implementation will be used in preference to the auto-generated one
- If you only provide values for some template parameters, then this is called **partial specialisation** and has some more rules associated with it
- You can pass templates as parameters to templates, which are called **template template**s - they can be quite tricky to use right

Further Templates

- Technically the templating system in C++ is a complete programming language itself
- There is a whole branch of C++ programming called **template metaprogramming** to write programs using the template system
- This is very powerful for some things, but is also very tricky to do well - use with caution
- There are a variety of tools that templates provide you with to help if you want to go this way
- **std::enable_if**, **if constexpr()**, **std::invoke_result**