IB9N7 C++ for Quantitative Finance

Lecture 16: Templates

J.F.Reizenstein@warwick.ac.uk

3 March 2016
(Week 22)

Outline

In this lecture, we'll introduce templates.

Templates may seem scary at first, but are extremely powerful for code reuse and totally worth it. You are not required to write your own templates in your projects.

Template functions

Motivating example

Suppose you are interested in solving the following task:

Task

Write a function to compute the sum of the elements elements in a vector.

You might define the following function:

Example: Sum function

```
1 double calculate_sum (const std::vector<double> & data)
2 {
3     double sum = 0.0;
4     for (double d : data)
5         sum += d;
6     return sum;
7 }
```

Motivating example continued

But you can't use the previous function to sum an `int` vector.
We could use function overloading...

Example: Sum function for `int`s

```
1 int calculate_sum (const std::vector<int> & data)
2 {
3    int sum = 0;
4    for (int i : data)
5       sum += i;
6    return sum;
7 }
```

Q: What's different about this overload compared to the first function?

Motivating example, again

But you can't use the previous function to sum an `unsigned int`/`Rational`/`Complex` vector.

You might despair that you need to write a new overload for every type,

(at least where you might want to compute the sum of an vector of that type).

But worry not!

In order to solve this for a general type, we first need to answer this question:
For what types might we want to be able to apply the `calculate_sum` function?

Potentially, for all types which support `+=`.

Templates

Clearly, it is undesirable to overload functions to provide identical functionality for a number of types,

especially where the function body is identical (or almost so).

C++ provides a feature called **templates** that allows us to solve this problem somewhat more elegantly, without even having to list the types we would like to be able to use.

Before we discuss templates in detail, let's see a templatised solution to this particular problem.

Template solution

First example with templates

```
1 template <typename T>
2 T calculate_sum (const std::vector<T> & data)
3 {
4    T sum(0); // assuming can construct a T with value 0
5    for (typename std::vector<T>::size_type i = 0u; i
         != data.size(); ++i)
6       sum += data.at(i);
7    return sum;
8 }
```

When using this function, the actual template parameter type `T` just needs to support construction with the value 0, copy construction, and the `+=` operator with a `const T` operand.

Using the template solution

With just the templatised `calculate_sum` function from the previous slide, the following `main` function works as intended:

Example

```
int main ()
{
    std::vector<double> a(3u);
    std::vector<int> b(3u);
    a.at(0u) = 1.1; a.at(1u) = 1.2; a.at(2u) = 1.3;
    b.at(0u) = 1; b.at(1u) = 2; b.at(2u) = 3;
    std::cout << calculate_sum(a) << std::endl; // 3.6
    std::cout << calculate_sum(b) << std::endl; // 6
    return EXIT_SUCCESS;
}
```


Uses of templates

Here are some of the most important uses of templates:

Code reuse: Templates are parameterised functions and classes.
As seen in the last example, we only need to write one function and then can use it with different types, without needing to specify or have any knowledge of those types.

Performance: Templates can move many runtime computations into compile time.

Flexibility: Templates allow many design patterns previously not possible through traditional object-oriented programming.

But: can hurt code size, can be slow to compile, can be hard to debug/test.

Template concepts

The most important aspect of template programming is to capture **concepts**.

In the above example,

summation of a vector of elements is a general concept applicable to vectors of any type `T` that supports construction with the value 0, copy construction, and the `+=` operator with a const `T` operand.

We don't need inheritance to abstract this generality.

Note that templates are less suitable when the function bodies will be different for different types — there, function overloading should be used instead.

STL solution

Our `calculate_sum` example took a vector argument.

In fact, the STL provides a templatised function `std::accumulate` in header `<numeric>` to apply a binary operator to a collection of variables indexed through iterators.

To use the default binary operator `+`, use the overload with prototype

```
template <typename Iterator, typename T>
T std::accumulate (Iterator begin, Iterator end, T
    init);
```

STL solution(2)

The earlier summations can be easily obtained using **std::accumulate**, as follows:

```
#include <iostream>
#include <numeric>
#include <cstdlib>
int main ()
{
    //... a, b as before
    std::cout << std::accumulate(a.begin(), a.end(),
    0.0) << std::endl; // 3.6
    std::cout << std::accumulate(b.begin(), b.end(),
    0) << std::endl; // 6
    return EXIT_SUCCESS;
}
```


Example 2: max function

Consider the simple problem of finding and return the maximum of two values of the same type.

Example: maximum of two ints

```
int max (int a, int b) { return (a < b) ? b : a; }
```

However, this function clearly does not work as desired with values of type **double**:

Example: Incorrect for other types

```
std::cout << max(3.5, 3.6) << std::endl; // Oops! Get
    integer 3
```


Templatised max function

Thus we need a new overload

```
double max (double a, double b) { return (a < b) ? b
    : a; }
```

but:

- the function bodies are exactly the same,
- we are still restricted to the types **int** and **double**.

Just like in the **calculate_sum** example, we should use function templates.

A function template provides a way to define a family of functions where some of the types involved are left open.

Templatised max function (2)

Here is the definition of a templatised **max** function:

```
template <typename T>
T max (T a, T b) { return (a < b) ? b : a; }
```

However, **T** might be an object type, so it might be better to use **const** references (also valid when **T** is a fundamental type):

```
template <typename T>
const T &max (const T &a, const T &b) { return (a < b
    ) ? b : a; }
```

C++ provides a templatised max function

C++ even provides templatised versions of the **max** function already in the <algorithm> header (in the **std** namespace).

- One such version generalises to allow use of a different comparison method.
- The one that uses the < binary operator is essentially equivalent to the definition above.

Function templates

A normal function parameterises its input.
A function template (or template function) adds another layer of parameterisation — in our cases, types. These parameters are known at compile time.

A function template definition therefore has two kinds of parameters:

- Template parameters, e.g., the parameter **T**.
- Function parameters, e.g., the parameters **a** and **b**.

Template parameters

The actual type of the template parameter(s) **T** can vary for each function call and is determined at compile-time.

> For each type(s) that a templatised function is used with, a version of the function is generated corresponding to those type(s) — see template instantiation.

Warning! When permitting template type parameters, note that the substituted type argument could be a fundamental type or an object type.

Function templates: syntax

The function prototype(s) are preceded with:

- the keyword **template**, then
- in angle brackets: a comma-separated list of (zero or more) template parameters, which consist of
  - either: one of the keywords **typename** or **class** (or the type of a non-type template parameter —see later)
  - followed by an identifier,
  - and possibly a default type/value preceded by the equals symbol.

Template specialisation

Can override generic functions for specific types, usually for handling special cases (either it can be done more efficiently, or the code has to be different to the general case).

Specifying zero template parameters in this part of the prototype allows such specialisation of an existing templatised function.

Example

```
1 template <>
2 bool max(bool a, bool b)
3 {
4     return a || b;
5 }
```


Template instantiation

Compiler generates code when required (**instantiates** the template).

Note that the instances with differing template types are actually different function units.

Compiler needs to see the implementation, not just the interface, to instantiate templates.

- Put both the template prototypes and implementations together in the **header** file.
  Don't use normal interface/implementation separation into header and implementation files.
- Note that the linker can cope with seeing the same template definitions in multiple translation units.

Using function templates

After the function template is defined, it can be used just like any other function.

However, unlike other functions, the parameter types are unknown when the function is defined.

Therefore, there needs to be a way for the compiler to determine (or the programmer to specify) the actual type of the template parameter.

Recall that C++ is a static language; everything has to have a type at compile time.

There are three ways to determine the type `T`:

- implicit deduction;
- explicit specification;
- specification of defaults

Implicit deduction of types

When the template parameters are the types of the function's arguments, then the compiler can often work them out by itself. Recall that, when we call `max` on two `int`s, the template parameter `T` is automatically determined to be `int`.

This is because both arguments `a` and `b` are of type `int`, and thus the compiler can absolutely be sure of the template parameter type.

The detailed rules of this kind of deduction are quite complicated.

However, the following slides give enough information to understand the basic situations and thus use the STL which will be introduced in the next lecture.

Type deduction: obvious case

If a function parameter type is the same as the template parameter type, then that type is the type of the argument when the function is used.

```
template <typename T> T max (T a, T b);
max(1, 2); // T: deduced to be int
```


Type deduction: References

If the function parameter is declared to be a reference to the template parameter type, then the template type is determined to be the type of the original object referred by the reference.

```
template <typename T> T rmax (T &ar, T &br);
const int &ar = a;
const int &br = b;
rmax(ar, br); // T: deduced to be const int
```


Type deduction and const

The deduced type is allowed to be more `const` than the argument type.

```
template <typename T> const T &crmax (const T &a, const
    T &b);
int a, b;
int &ar = a;
int &br = b;

crmax(a, b);
// Actual argument type: int
// Deduced template type T: int
// Formal parameter type: const int &

crmax(ar, br);
// Actual argument type: int &
// Deduced template type T: int
// Formal parameter type: const int &
```


Explicit type specification

If the template parameter is specified explicitly, no deductions are performed.

However, it is still possible that the function template is invalid.

Example

```
template<typename T> T &rmax (T &a, T &b);
const int a = 2, b = 5;
rmax(a, b); // OK!
// Actual argument type: const int
// Deduced template type T: const int
// Formal parameter type: const int &

rmax<int>(a, b); // Oops! Cannot convert type 'const int' to type 'int&'
// Actual argument type: const int
// Specified template type T: int
// Formal parameter type: int &
```

Type deduction and return types

The return type and types used in the body of the function do not participate in template parameter deduction.

However, they can still be a template type parameter, but it is required to specify the type explicitly on every call.

SFINAE

Substitution failure is not (always) an error (SFINAE) concept:

If the compiler attempts a substitution during type deduction that would result in an error. . .

it gives up consideration of that substitution from the candidate substitutions

. . . rather than displaying the error and halting.

In other situations, substitution failure will result in a hard error.

Type deduction and implicit conversion

Most implicit type conversions are not allowed in the deduction process.

Example

```
1 max(1, 1.2); // Oops
2 // a: int => T deduced to be int
3 // b: double => T deduced to be double
4 // Compiler won't be able to determine what to use as T
```


Explicit type specification

In the last example,

```
max(1, 1.2); // Oops
```

the compiler can't determine the type of the template parameter `T`.

In this situation, one needs to explicitly specify the type, as in the following:

Example

```
1 max<double>(1, 1.2); // Evaluates to 1.2
2 max<int>(1, 1.2); // Evaluates to 1
```

Non-type template parameters ☆

The template system also allows specification of standard parameters as template parameters, with certain restrictions.

These are known as non-type template parameters.

There are restrictions on what types can be used. In common cases, only enums, bool, and integer types are found.

Non-type template actual parameters must be compile-time constant expressions.

Also accept default values.

Non-type template parameters: example ☆

Toy example

```
1  template <bool flag, typename T>
2  void output (T const & element)
3  {
4      // 'flag' is effectively const here, cannot assign to it
5      std::cout << (flag ? "Left" : "Right") << "␣" <<
           element << std::endl;
6  }
7  int main()
8  {
9      double z = 5.2;
10     output<true, double>(z); //or output<true>(z)
11     const bool x = false;
12     output<x, double>(z); // x must be const for this to work
13 }
```


Class templates

Class templates

The **std::vector** class is a template class, and its definition looks something like this:

```
1  namespace std {
2      template <T>
3      class Vector<T>
4      {
5          // ...
6      };
7  };
```

Using a templated class

To use a templatised class, we need to explicitly specify the template parameter.

```
std::vector<double> a(10u);
```

When the compiler first sees `std::vector<double>`, it creates a class based on the class template `std::vector<T>` by substituting `T` with `double`.

Unlike for function templates, there is no template parameter deduction.

All template parameters have to be specified explicitly.

Default type specification ☆

Example

```
1  // Suppose the vector class has been defined instead as:
2  template <typename T = double>
3  class vector
4  {
5      // ... as before ...
6  }
7  // (but note that it was not so defined by the standard library)
8  // then we could do
9  int main()
10 {
11     std::vector<> a(10u);
12     a.at(0u) = 1.1;
13     // ...
14 }
```


Lab

Objectives for lab session