Template specialisation

Posted on September 4, 2014 by Glennan Carnie

Contents

Introduction

Welcome back to the wonderful world of templates.

So far, we have looked at what are known as *base templates*. In this article we're going to look at one of the more confusing aspects of templates – *specialisation*. The choice of the word specialisation is unfortunate, as many confuse it with inheritance and sub-typing; in this case specialised means "more specific".

Template specialisation comes in two forms:

- *Explict specialisation*, where a unique version of the template is defined for a specific type
- *Partial specialisation*, where a template is defined that acts on a qualified range of types (for example, pointers-to-type)

To add to the confusion template functions and template classes behave in different ways:

- Template functions may be explicitly specialised, but not partially specialised. Template functions may be overloaded, though.
- Template classes cannot be overloaded; but they can be both explicitly- and partially-specialised.

To help overcome some of this confusion, think how non-template functions and classes work:

- Functions can be overloaded, but not inherited
- Classes may be inherited, but not overloaded

Template functions and classes attempt to follow similar semantics. But let's have a look at them in more detail.

Overloading template functions

Back in the first article in this series we said that templates can be overloaded with non-template versions.

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

using namespace std;

template <typename T>
T min (T a, T b)
{
  return a < b ? a: b;        // type T must support operator <
}

const char* min(const char* a, const char* b)
{ return strcmp(a,b) < 0 ? a : b; }


int main()
{
  cout << min("Hello","World") << endl;          // Non-template version
  cout << min(const char*("Hello"),"World") << endl;    // Template version
}
```

Template and non-template functions can co-exist.

The compiler will always favour the non-template version of the function.

It is also possible to overload the template function with another template function.

```
template <typename T>
T min (T a, T b)
{
  return a < b ? a: b;        // type T must support operator <
}

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


int main()
{
  int* p1 = new int(100);
  int* p2 = new int(99);

  int* x = min(p1, p2);    // Calls min(T* a, T* b) with T == int*
  int  y = min(*p1, *p2);  // Calls min(T a, T b)   with T == int
}
```

Template overload for min.

With no non-template overload, the compiler chooses the template function that is *most specialised* – that is, the template function that best fits the parameters it deduces from the call.

In the first call, the compiler deduces the parameter type is `int*` (the declared type of `p1` and `p2`) and so instantiates the template that matches best – `T* min(T* a, T* b)`

In the second call, the compiler deduces the type of `T` as `int`, and so instantiates the less-specialised template – `T min(T a, T b)`.

Explicitly specialised template functions

It is also possible to explicitly specialise a template function. An explicitly specialised template function is when the function is declared for a specific type.

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

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

template<>
const char* min(const char* a, const char* b)
{
  return strcmp(a, b) < 0 ? a : b;
}
```

Full specialisation of min function for const char*

Calls template specialisation for const char*

```
int main()
{
  const char* str1 = "Hello";
  const char* str2 = "World";

  cout << min(str1, str2) << endl;
}
```

The above code gives ostensibly the same results as the non-template overload. However, because of the (arcane) rules the compiler uses to determine which template to instantiate you may not always get the result you anticipated.

I recommend that you ALWAYS prefer non-template overloads to template function specialisation.

(The reasons are beyond the scope of this article; and Herb Sutter has written an excellent description of the problem here.)

Template functions may be overloaded for different signatures and explicit specialisations which gives the illusion of function partial specialisation. However, because of subtleties in the way the compiler decides which template function to instantiate, overloading template functions with other template functions is best avoided.(The reasons are beyond the scope of this article; and Herb Sutter has written an excellent description of the problem here.)

Explicitly specialised template classes

Let's switch our attention to template classes. As I said in the introduction, with classes we can provide both explicitly specialised and partially-specialised versions of the template class.

In this example we have a class `MinFinder`, with a single method, `get()`. We can create instances of the template class for different types but in the case of strings the behaviour we get is not (necessarily) what we might want.

```
template<typename T>
class MinFinder
{
public:
  T get(T a, T b)
  {
    return (a < b)? a : b;
  }
};
```

```
int main ()
{
  MinFinder<int> mf1;
  MinFinder<const char*> mf2;

  const char* str1 = "Hello";
  const char* str2 = "World";

  int i = mf1.get(100, 200);
  const char* c = mf2.get(str1, str2);

  cout << i << endl;
  cout << c << endl;
}
```

Probably not give the correct result

We can explicitly specialise the `MinFinder` class, creating a version that works specifically for `const char*` objects.

```
template<typename T>
class MinFinder
{
public:
  T get(T a, T b)
  {
    return (a < b)? a : b;
  }
};

template<>
class MinFinder<const char*>
{
public:
  const char* get(const char* a, const char* b)
  {
    return (strcmp(a, b) < 0)? a : b;
  }
};
```

Template specialisation for const char*

```
int main ()
{
  MinFinder<int> mf1;
  MinFinder<const char*> mf2;

  const char* str1 = "Hello";
  const char* str2 = "World";

  int i = mf1.get(100, 200);
  const char* c = mf2.get(str1, str2);

  cout << i << endl;
  cout << c << endl;
}
```

Uses template specialisation

Notice in this case the compiler is not doing any type deduction – we are explicitly telling the compiler which template class to instantiate. This is a bit clumsy; we'll look into that shortly.

Partially specialised template classes

In the above example we have specialised for a particular type (`const char*`). However, we may want different behaviour for all pointer types – for example, comparing the referents of the pointers, rather than the pointers themselves. Unlike template functions we can't overload template classes. It would be impracticable to provide explicit specialisations for every pointer type, so C++ allows us to provide a *partial specialisation*.

```
template<typename T>
class MinFinder
{
public:
  T get(T a, T b)
  {
    return (a < b)? a : b;
  }
};


template<typename T>
class MinFinder<T*>
{
public:
  T* get(T* a, T* b)
  {
    return (*a < *b)? a : b;
  }
};


template<>
class MinFinder<const char*>
{
public:
  const char* get(const char* a, const char* b)
  {
    return (strcmp(a, b) < 0)? a : b;
  }
};
```

Explicit specialisation for string literals

Now we have defined:

- A *base template*, which works for everything; except...
- A *partial specialisation*, which will be favoured for all pointers-to-type; except...
- An *explicit specialisation*, which works specifically for `const char*` types.

The compiler will choose the appropriate template to instantiate based on the type of the template parameter it is instantiated with. Partial specialisations may be created for:

- Pointers
- References (r-value and l-value)
- const objects
- ...and (valid) combinations of the above

Note that, if an explicit specialisation exists the compiler will prefer it to the partial specialisation. The compiler will always prefer the 'most specialised' (read: 'most specific') template. This extends for non-template classes – which, like template functions, will always be preferred to the equivalent template version

Hiding the template specialisations

At this point we have the ability to provide unique behaviours through partial- and explicitly-specialised template classes; but we must explicitly create the class we want to use.

To improve the usability of our code we need to combine template functions and template classes:

- Use the compiler's ability to deduce template types for template functions
- Use the ability to partially specialise template classes to get different beahviours for different (groups of) types.

The mechanism is to wrap the instantiation of a template class inside a template function:

```
template<typename T>
class MinFinder
{
};

template<typename T>
class MinFinder<T*>
{
};

template<>
class MinFinder<const char*>
{
};
```

```
template<typename T>
T min(T a, T b)
{
  MinFinder<T> mf;
  return mf.get(a, b);
}
```

Instantiates the appropriate specialisation, depending on the type of T

The compiler deduces the template parameters from the function call; this is then used to create the appropriate (partially) specialised class, which does the work for us.

```
int main ()
{
  int a = 10;
  int b = 45;

  int* p = min(&a, &b);                 // creates MinFinder<int*>
  const char* h = min("Hello", "World"); // creates MinFinder<const char*>
  int i = min(a, b);                    // creates MinFinder<int>
}
```

Compiler deduces template type from call

Summary

Template specialisation is a powerful way of building expressive library code, by allowing us to define different beahviours for different 'categories' of types. The real strength of this is allowing us to extend libraries after-the-fact, for types that may have not been considered during the library's conception.

Template functions may be overloaded for different signatures and explicit specialisations which gives the illusion of function partial specialisation. However, because of subtleties in the way the compiler decides which template function to instantiate, overloading template functions with other template functions is best avoided.

Template classes can be both explicitly and partially specialised, but must be explicitly instantiated. This presents a possible limitation for library code, but can be circumvented by combining template partial specialisations with template function type deduction.

In the final article of this series we'll have a look at the problem of communicating type information between different templates, using *trait template* classes.

Glennan Carnie
Technical Consultant @ Feabhas Ltd

Glennan is an embedded systems and software engineer with over 20 years experience, mostly in high-integrity systems for the defence and aerospace industry.

He specialises in C++, UML, software modelling, Systems Engineering and process development.

Glennan Carnie

Website | + posts

Glennan is an embedded systems and software engineer with over 20 years experience, mostly in high-integrity systems for the defence and aerospace industry.

He specialises in C++, UML, software modelling, Systems Engineering and process development.

6 Responses to *Template specialisation*

Andrea says:
September 4, 2014 at 4:28 pm

Great post as usual, but I think there's a slight glitch in your second "slide": when choosing between the two overloads for "min":

int *x = min (p1, p2);

Chooses the second overload, as you say, but I think the deduced type for "T" is still int, it's the parameter(s) type that is "T*". IOW I think the comment for that call should be
//Calls min(T *a, T *b) with T == int

Dan says:
September 7, 2014 at 2:54 pm

Always enjoy the blog posts.

Maybe we have different ways of saying the same thing, but I'm not sure I agree with the definition of partial specialisation: "Partial specialisation, where a template is defined that acts on a qualified range of types (for example, pointers-to-type)"

I always think of partial specialization as a specialization which specifies exact types for some of the primary template parameters, but not all of them.

glennan says:
September 8, 2014 at 8:29 am

Dan,

I like that way of saying it. Would you mind if I appropriated that and added it to the article?

I always appreciate any feedback that can make the articles better.

Pingback: Sticky Bits » Blog Archive » Trait+classes

Kranti Madineni says:
April 23, 2018 at 5:48 am

Another interesting topic explained clearly in real time usage

Dan says:
April 23, 2018 at 12:36 pm

Please Glennan, feel free to do so. I appreciate the time you spend creating these clear write-ups. (Sorry, I realise your comment was from 2014, but I'm only now receiving the notification!)