Template member functions

Posted on July 24, 2014 by Glennan Carnie

Contents

Introduction

Previously we've looked at template functions and we've looked at template classes. This time, let's look at what happens when you combine them.

Template member functions

A non-template class can have template member functions, if required.

```
#include <iostream>
using namespace std;

class ADT
{
public:
  template <typename T>
  void Func(T val);
};

template <typename T>
void ADT::Func(T val)
{
  cout << "Value is " << val << endl;  // T must have overload for operator<<
}

int main()
{
  ADT a1;

  a1.Func(100);    // Generates void ADT::Func<int>(int val);
  a1.Func(17.4);   // Generates void ADT::Func<double>(double val);
}
```

Notice the syntax. Unlike a member function for a template class, a template member function is just like a free template function but scoped to its containing class. In this case the compiler generates a member function for each (unique) template type. As before, the compiler will favour a non-template overload of a member function.

Template functions on template classes

What about if our class is itself a template? A template class will typically have member functions defined in terms of its own template parameter, but may equally have member functions that are themselves template functions. In this case the member function's template parameter is different to the owning class'.

The syntax for specifying the template member function is rather obtuse (to say the least):

```
template<typename T>
class Utility
{
public:
  void out T vlue);

  template<typename U>
  void Func(U vlue);
};

template<typename T>
void Utility<T>::out T vlue)
{
  // out()'s behaviour...
}

template<typename T>
template<typename U>
void Utility<T>::Func(U vlue)
{
  // Func()'s behaviour...
}
```

Template member function

```
int main()
{
  Utility<int> utility;
  utility.Func(100);
  utility.Func("Hello World");
}
```

The containing class' template declaration must be 'outer' one; foo; is a template function (typename T) that belongs to the class Utility<T>. (typename T)

Notice that the template function *must* have a different template parameter identifier to its containing class (even if the same type is being used to instantiate both the class and the function).

Template constructors

A common application of template member functions on template classes is constructors; particularly if inheritance is involved. (Have a look here for more on template inheritance)

Let's revisit a previous example:

```
class Waypoint
{
public:
  Waypoint(float lat = 0.0, float lon = 0.0)
  void display();

private:
  float longitude;
  float latitude;
};
```

Template parameter is the base class.

```
template<typename T>
class Named : public T
{
public:
  template<typename Arg1, typename Arg2>
  Named(const char* str, Arg1&& arg1, Arg2&& arg2);

  void display();

private:
  string name;
};

template<typename T>
template<typename Arg1, typename Arg2>
Named<T>::Named(const char* str, Arg1&& arg1, Arg2&& arg2) :
  T(std::forward<Arg1>(arg1), std::forward<Arg2>(arg2)),
  name(str)
{
}
```

Template type must support display()

Meyers' "Universal Reference"

Pass on parameter unchanged

```
int main()
{
  Waypoint wp1;
  Named<Waypoint> wp2("Home", 1.30, 53.775);

  wp1.display();
  wp2.display();
}
```

OK... deep breath... let's wade through this mire of syntax and work out what's happening.

The constructor for *Named* has three parameters: a string literal (const char*) and two template parameter arguments. Notice these parameters are passed as r-value references (Arg&&). In this situation Scott Meyers refers to these as *Universal References* – meaning "a reference that can bind to *anything*".

In the body of the constructor, notice the use of the template function std::forward. This function ensures that the types of the parameters are forwarded on without changing their types – that is, l-values remain l-values, r-values remain r-values.

(I'm deliberately glossing over the details of these mechanisms here as they're not the focus of this article. For a detailed description I highly recommend reading Meyers' excellent article "Universal References in C++11"

Now when we construct a *Named* object, its constructor (function) can deduce the types of the supplied arguments and pass them on to the (template-parameter) base class.

```
class Waypoint
{
public:
  Waypoint(float lat = 0.0, float lon = 0.0)
  void display();

private:
  float longitude;
  float latitude;
};
```

```
int main()
{
  Waypoint wp1;
  Named<Waypoint> wp2("Home", 1.30, 53.775);

  // Instantiates a constructor of the form:
  // Named<Waypoint>::Named(const char*, double&&, double&&)

  wp1.display();
  wp2.display();
}
```

So far, so good, but our *Named* constructor is still limited: It will only work for base classes that have exactly two parameters. If we try and use a class with three parameters we get a problem:

```
class Lamp
{
public:
  Lamp(char ro, int dev, bool onState);
  void on();
  void off();
  void display();

private:
  char room;
  int device;
  bool isOn;
};
```

```
int main()
{
  Lamp study;
  Named<Lamp> desk("Desk lamp", 'A', 1, true); // ERROR!

  study.display();
  desk.display();
}
```

```
ERROR C2661: 'Lamp::Lamp' :
             no overloaded function
             takes 2 arguments
```

Our *Lamp* class is a candidate for being the *Named* base class, since it supports the display() method. However, its constructor requires three parameters – and our template constructor can only supply two.

The short-term fix is to overload the *Named* constructor to take three parameters:

```
template<typename T>
class Named : public T
{
public:
  template<typename Arg1, typename Arg2>
  Named(const char* str, Arg1&& arg1, Arg2&& arg2);

  template<typename Arg1, typename Arg2, typename Arg3>
  Named(const char* str, Arg1&& arg1, Arg2&& arg2, Arg3&& arg3);

  void display();

private:
  string name;
};
```

```
int main()
{
  Lamp study;
  Named<Lamp> desk("Desk lamp", 'A', 1, true);
  // +
  // Instantiates a constructor of the form:
  // Named<Lamp>::Named(const char*, char&&, int&&, bool&&)

  study.display();
  desk.display();
}
```

This works; but it means we'll have to overload the constructor for zero parameters, one parameter, two, three, four, etc. This can quickly become onerous, so in the next article we'll have a look at a new mechanism in C++ is designed to make this more flexible – *Variadic Templates*.

Summary

Template member functions allow us to parameterise functions independently of the class they belong to. They can be added to both template and non-template classes.

Template member functions follow all the usual rules of template functions – they can be overloaded (both by template versions and non-template versions) and they may be overridden by derived classes.

If you'd like to know more about templates, and C++ programming – particularly for embedded and real-time applications – visit the Feabhas website. You may find the following of interest:

C++-501 – C++ for Embedded Developers

C++-502 – C++ for Real-Time Developers

AC++-501 – Advanced C++

AC++-401 – Advanced C++ for C++11

Glennan Carnie
Technical Consultant at 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.

This entry was posted in C/C++, Programming and tagged C++, C++11, constructor, forwarding, functions, inheritance, member functions, Overloading, r-value, r-value references, Templates. Bookmark the permalink.

← Templates and polymorphism Variadic templates →

1 Response to *Template member functions*

Kranti Madineni says:
April 21, 2018 at 12:00 pm

I have become a fan of yours.

Leave a Reply

Enter your comment here...