Template inheritance

Posted on June 19, 2014 by Glennan Carnie

Contents

Introduction

Previously we looked at template class syntax and semantics. In this article we'll extend this to look at inheritance of template classes.

Inheriting from a template class

It is possible to inherit from a template class. All the usual rules for inheritance and polymorphism apply.

If we want the new, derived class to be generic it should also be a template class; and pass its template parameter along to the base class.

```
template<class T>
class Base
{
public:
  void set(const T& val) { data = val; }

private:
  T data;
};

template<class T>
class Derived : public Base<T>
{
public:
  void set(const T& val);
};

template<class T>
void Derived<T>::set(const T& v)
{
   // Call base-class behaviour
   Base<T>::set(v);
   // …plus any derived-class behaviour
}
```

Override base class behaviour

Notice that we 'pass-down' the template parameter from the derived class to the base class since *there is no class Base*, only class Base<T>. The same holds true if we wish to explicitly call a base class method (in this example, the call to Base<T>::set())

We can use this facility to build generic versions of the *Class Adapter Pattern* – using a derived template class to modify the interface of the base class.

```
template<class T>
class Utility
{
public:
  void operation1(const T& val);
  int  operation2();
  bool operation3();
  void operation4(const T& val);
  T    operation5();
  void operation6(const T& val);

private:
  T data;
};

template<class T>
class Adapter: private Utility<T>
{
public:
  void newOperation(const T& val)
  {
    Utility<T>::operation1(val);
  }

  T anotherOperation()
  {
    return Utility<T>::operation5();
  }
};
```

Private inheritance to hide base class operations

Notice, in this case we use *private* inheritance. This hides the base class methods to any clients (but keeps them accessible to the derived class. Calls to the Adapter's methods are forwarded on to the base class. For more on the Adapter Pattern see this article.

Extending the derived class

The derived class may itself have template parameters. Please note there must be enough template parameters in the derived class to satisfy the requirements of the base class.

```
template<class T>
class Base
{
public:
  void set(const T& val) { data = val; }

private:
  T data
};

template<class T, class U>
class Derived : public Base<T>
{
public:
  void set(const T& val);

private:
  U derived_data;
};
```

Derived class may have its own template parameters

Specialising the base class

The derived class may also specialise the base class. In this case we are creating an explicit instance of the base class for all versions of the derived class. That is, all derived class instances, regardless of the type used to instantiate them, will have a base class with a data object of type int.

```
template<class T>
class Base
{
public:
  void set(const T& val) { data = val; }

private:
  T data
};

template <typename U>
class Derived : public Base<int>
{
public:
  void set(const T& val);

private:
  U derived_data;
};
```

Explicitly specialised base class

Deriving from a non-template base class

There is no requirement that your base class be a template. It is quite possible to have a template class inherit from a 'normal' class.

```
class NonTemplate
{
public:
  void setData(int val);
  int  getData();
  // Other non-template methods...

private:
  int data;
};

template <typename T>
class Derived : public NonTemplate
{
public:
  void set(const T& val);
  T    get();

private:
  T derived_data;
};
```

Non-template type data and operations

template type-specific data and operations

This mechanism is recommended if your template class has a lot of non-template attributes and operations. Instead of putting them in the template class, put them into a non-template base class. This can stop a proliferation of non-template member function code being generated for each template instantiation.

Parameterised inheritance

So far, we have been inheriting explicitly from another (base) class, but there is no reason why we can't inherit from a template type instead – with the (reasonably obvious) requirement that the template parameter must be a class type. This mechanism allows some neat facilities to be implemented.

In the example below we have built a template class, Named. The class allows a name (string) to be prepended to any class, with the proviso that the base class (the template parameter) supports the member function display().

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

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

```
template<typename T>
class Named : public T
{
public:
  Named(const char* str) : name(str) {}
  void display();

private:
  string name;
};

template<typename T>
void Named<T>::display()
{
  cout << name << " ";
  T::display();
}
```

Template parameter is the base class.

Template type must support display()

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

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

In this example we have constructed a normal class, Waypoint, which supports the display() method (presumably, this displays the current coordinates of the Waypoint; but who knows…).

When we create a Named object, we give it the type of the object we want to 'decorate' with a name (in this case, a Waypoint). Calling display() on a Named object will cause it to output the object's name, before calling display on its base class – in our example Waypoint::display().

(NOTE: I'm quite deliberately ignoring the 'Elephant in the Room' with this code – how to generically construct a base class object with a non-default constructor. I'll revisit this example again in the future when we look at *Variadic Templates*.)

Summary

This time we've covered most of the basic inheritance patterns for templates. Next time we'll have a look at a practical application for templates in your application code.

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.

Like (30) Dislike (1)

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 Class, inheritance, Overloading, Overriding, Specialisation, Templates. Bookmark the permalink.

← Template classes Templates and polymorphism →

9 Responses to *Template inheritance*

Dan *says:*
June 30, 2014 at 4:49 am

Please keep the informative, well-written articles coming. Looking forward to the next post.

I use this technique a lot, but most of the people I work with are either baffled by this, or say, "You mean you can do that?"

Also, good use of private inheritance.

Couple quick notes on the first box (code snippet):

- I think to "properly" get the expected polymorphic behavior when overriding set() in Derived(), the function should be virtual in Base

- I think there is a missing semicolon in the definition of "data" for class "Base" (hazard of writing code for PowerPoint, I suspect... I can relate)

Like (2) Dislike (3)

Peter Bushell *says:*
July 1, 2014 at 1:50 pm

A good article (again) but I don't necessarily agree with this, in the second sentence: "The derived class should be a template class too"

One reason for inheriting a template class might be to allow the creator of the derived class to configure certain things, at compile time, *without* having to make the derived class, itself, a template. For example, the way to make a task class using a suitable C++ RTOS might be:

const unsigned int myStackSize = 512;
class MyTask: public Task
{
//---
};

[This is a somewhat over-simplified version of a facility offered in Dyagem (an ongoing C++ RTOS project).]

Like (2) Dislike (1)

Peter Bushell *says:*
July 1, 2014 at 1:54 pm

Unfortunately, your blog's HTML conversion completely removed the template parameter following "Task" in my previous post! There were some angle brackets enclosing the identifier "myStackSize".

Like (2) Dislike (0)

glennan *says:*
July 3, 2014 at 4:20 pm

Oops. A cut-and-paste error left the beginning of this article in a messed-up state. I've edited it to parse properly now.

Peter,

I agree, you needn't make the derived class a template class; in fact, this is covered in the section "specialising the base class".

Like (2) Dislike (0)

thanks *says:*
March 13, 2016 at 9:07 pm

Thank you for the article.

Brief short informative.

Like (1) Dislike (0)

DG *says:*
October 11, 2016 at 3:48 am

Firstly thank you for this article its extremely informative and to the point.
I have a question, is it mandatory that the template parameters of both derived and base class match
eg
template
class base
{
.....
};
template
class derived:public base
~~{~~
~~.....~~
~~};~~
~~Is it okay that the template parameter of base is T and that of derived is s?~~

Like (0) Dislike (0)

Glennan Carnie *says:*
October 12, 2016 at 4:02 pm

Yep, that's fine; as long as it doesn't render the code confusing the reader 🙂

Like (0) Dislike (0)

Kranti Madineni *says:*
April 21, 2018 at 12:06 pm

First time i am feeling really comfortable with templates...It''s like a horror movie to me all these 10 years of my c++ career... but now its seems like a romantic movie because of the authors good words in the starting, nothing to worry about templates. that has given me some confidence firstly.

Like (0) Dislike (0)

Pingback: *Template inheritance from a template class [duplicate] – Windows Questions*

Leave a Reply

Enter your comment here...

Search

Follow Us

Categories

- Agile
- ARM
- Build-systems
- C/C++ Programming
- CMSIS
- Cortex
- Design Issues
- General
- Industry Analysis
- Linux
- Python
- Python3
- RTOS
- Testing
- Toolchain
- training
- UML
- Uncategorized
- webinar

Archives

- February 2022
- October 2021
- September 2021
- August 2021
- July 2021
- June 2021
- May 2021
- January 2021
- November 2020
- October 2020
- August 2020
- April 2020
- February 2020
- January 2020
- October 2019
- September 2019
- July 2019
- May 2019
- April 2019
- March 2019
- February 2019
- January 2019
- December 2018
- October 2018
- September 2018
- August 2018