← Template inheritance | Template member functions →

Templates and polymorphism

Posted on July 10, 2014 by Glennan Carnie

Contents

Introduction

Template functions and classes tend to cause consternation amongst programmers. The conversation tends to go something like this:

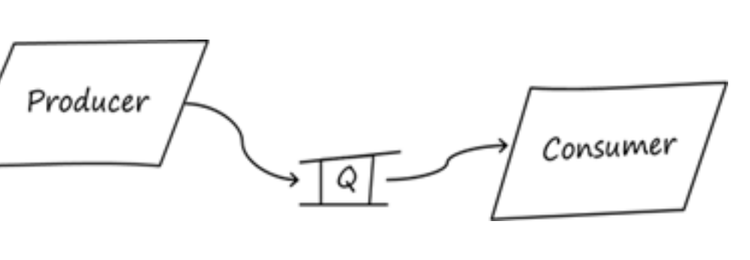
- I understand the syntax of templates (although it's ugly)
- I get the idea of replacing function-like macros with template functions
- I can see the application of template classes for containers
- Most containers and generic functions are library code
- I don't write libraries
- What's the point of me using templates?

In this article we're going to look at an application of templates beyond writing library code – replacing run-time polymorphism (interfaces) with compile-time polymorphism. This idea is known as a *Policy*. The idea is reminiscent of the *Strategy Pattern*, but uses templates rather than interfaces.

I'm going to assume you already have an understanding of basic template syntax and semantics. If you're a little rusty on these things look here and here for a refresher.

Setting the scene

In a multi-threaded environment it's typical for threads to communicate asynchronously using First-In-First-Out message queues. (Don't fret at this point about multi-threaded design; it's the construction of the queue we're going to worry about here, not how it would be used)



Because of the potential for race conditions within the queue (both threads trying to modify the queue 'simultaneously' – see here for a more detailed description) we should protect it with a mutual exclusion mechanism. For the purposes of this article we'll deliberately keep the queue code simplistic (in other words: don't try and use this queue in your production code!)

Here's our first pass at the queue code:

```
template<typename T>
class MessageQueue : std::queue<T>
{
public:
  MessageQueue();
  void post(T val);
  T get();

private:
  Mutex mutex;      <- Queue protected by its own Mutex object
};

template<typename T>
void MessageQueue<T>::post(T val)
{
  mutex.lock();             // LOCK
  std::queue::push(val);
  mutex.unlock();           // UNLOCK
}

template<typename T>
T MessageQueue<T>::get()
{
  T temp;
  mutex.lock();             // LOCK
  temp = std::queue::front();
  std::queue::pop();
  mutex.unlock();           // UNLOCK
  return temp;
}
```

For this example we've written the code as an adapter for the Standard Library queue class (using a variation of the Class Adapter Pattern). You can explore the semantics of template inheritance here if you're not familiar with it.

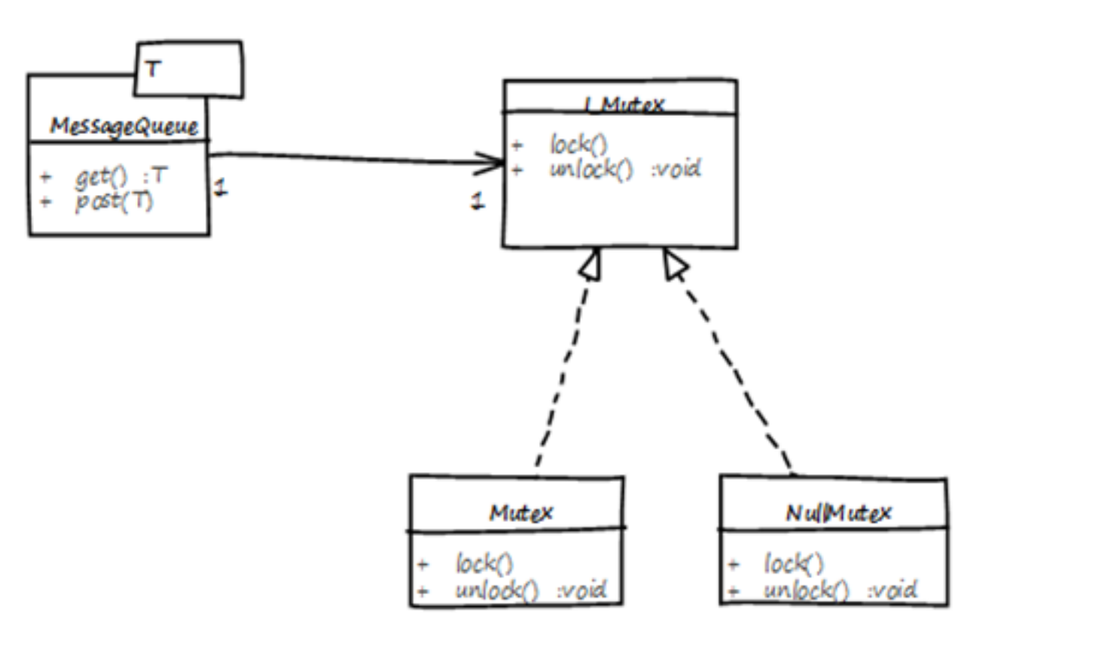
The flexibility of our design is limited by the fact we have hard-coded a concrete Mutex object into the MessageQueue. This could be a serious penalty if:

- We want to port to a different operating system (perhaps with more efficient mutual exclusion mechanisms)
- We want to use the MessageQueue in non-threaded code, where we don't want (or require) the overhead of locking/unlocking a Mutex object.

Let's look at a couple of solutions that can improve our design.

An interface solution

Firstly, we want to de-couple our MessageQueue class from any particular Mutex implementation. The usual way to do that is via an Interface.



The MessageQueue has an association to the Interface; and we can substitute classes that realise the interface.

Here's the code:

```
class I_Mutex
{
public:
  virtual void lock()   = 0;
  virtual void unlock() = 0;
};
```

```
template<typename T>
class MessageQueue : std::queue<T>
{
public:
  MessageQueue(I_Mutex& mtx) : pMutex(&mtx);
  void post(T val);
  T get();

private:
  I_Mutex* pMutex;   <- Association to interface
};

template<typename T>
void MessageQueue<T>::post(T val)
{
  pMutex->lock();             // LOCK
  std::queue::push(val);
  pMutex->unlock();           // UNLOCK
}

template<typename T>
T MessageQueue<T>::get()
{
  T temp;
  pMutex->lock();             // LOCK
  temp = std::queue::front();
  std::queue::pop();
  pMutex->unlock()            // UNLOCK
  return temp;
}
```

If you're familiar with using Interfaces the code above should be very familiar. The association between the MessageQueue and the Interface is implemented as a pointer. The binding between a MessageQueue object and a (concrete) Mutex class is done during the construction of the MessageQueue.

Now we have a mechanism for dealing with our above problems.

If we move to a new OS we can substitute a new class that realises the I_Mutex interface. (You might want to consider using the *Abstract Factory Pattern* for this code to eradicate a host of conditional compilation).

```
class WinMutex : public I_Mutex
{
protected:
  virtual void lock();
  virtual void unlock();

private:
  // Win32 implementation...
};
```

```
class PosixMutex : public I_Mutex
{
protected:
  virtual void lock();
  virtual void unlock();

private:
  // Posix implementation...
};
```

```
int main()
{
#ifdef WINDOWS
  Win32Mutex mutex;
#else
  PosixMutex mutex;
#endif

  MessageQueue<int> msgQ(mutex);  // Bind...
  ...
}
```

For non-threaded code we may implement a 'Null Mutex' – that is, a concrete implementation with empty methods. These calls will typically be optimised away by the compiler to leave a sequential version of the MessageQueue, with no lock/unlock overheads.

```
class NullMutex : public I_Mutex
{
protected:
  virtual void lock()  {}  // Empty...
  virtual void unlock(){}
};
```

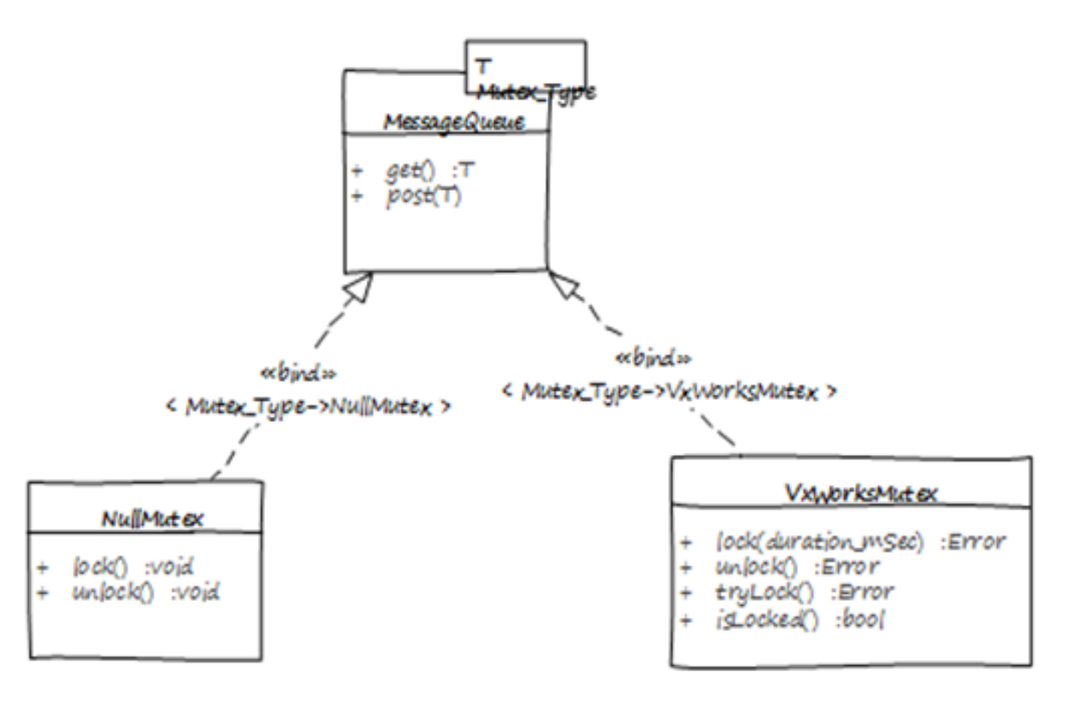
```
int main()
{
  NullMutex mutex;
  MessageQueue<int> msgQ(mutex);  // Bind...
  ...
}
```

However this flexibility comes at a cost. The run-time lookup for virtual functions requires additional code and data. Each dynamic polymorphic class requires a virtual table (*v-table*), and each object of that type a *v-table* pointer (*vtptr*). To call the polymorphic function the run-time system requires indexing into the *v-table* via the object's *vtptr* to actually call the function. In certain environments this can be up to twice as slow as a normal function call.

The template policy

One of the benefits of Interfaces is that it allows substitution of implementation even at run-time. However, you have to ask: are you likely to replace your operating system during program run-time? Reasonably, you're more likely to select the OS at compile time (like the example above) and not change it. Thus we are paying the price for flexibility we aren't going to use.

Step up, templates.



One of the characteristics of templates is that they impose certain requirements on the types that can be used to instantiate the template. It could be particular attributes, but it's more likely to be particular behaviours (for example, the need for types to support operator< if they want to be sorted in a container).

Let's redefine our MessageQueue class using a template parameter to enforce a Policy: whatever type is used to instantiate the template must support the methods lock() and unlock(). The MessageQueue class is referred to as a *host* class

```
template<typename T, typename Mutex_Type>   <- 'Policy'
class MessageQueue : std::queue<T>
{
public:
  MessageQueue();
  void post(T val);
  T get();

private:
  Mutex_Type mutex;
};

template<typename T, typename Mutex_Type>
void MessageQueue<T, Mutex_Type>::post(T val)
{
  mutex.lock();          <-
  std::queue::push(val);     Mutex_Type must support lock() and unlock()
  mutex.unlock();        <-
}

template<typename T, typename Mutex_Type>
T MessageQueue<T, Mutex_Type>::get()
{
  T temp;
  mutex.lock();
  temp = std::queue::front();
  std::queue::pop();
  mutex.unlock();
  return temp;
}
```

Please note, this is the *only* requirement we are demanding. There is no need to support any other methods, or realise an Interface; in fact, we're even being pretty lenient on the parameters and return types of the functions.

We define our 'policy' when we instantiate the MessageQueue class by specifying the type of mutex implementation we want.

```
class VxWorksMutex
{
public:
  Error lock(duration_mSec timeout = WAIT_FOREVER);
  Error unlock();
  Error tryLock();
  bool  isLocked();

private:
  // VxWorks implementation...
};
```

```
class NullMutex
{
public:
  void lock()   {}
  void unlock() {}
};
```

```
int main()
{
  // Set mutual exclusion policy
  //
  #ifdef VxWORKS
    typedef VxWorksMutex mutex_t;
  #else
    typedef NullMutex    mutex_t;
  #endif

  MessageQueue<int, mutex_t> msgQ;
  ...
}
```

Notice in the above code the VxWorksMutex supports a far richer interface than required by our MessageQueue. This does not affect its usage in our code. In this example, the default timeout parameter is used for the calls to lock(); and the error return codes are ignored.

Since there are no interfaces or virtual functions in the template code all calls are statically-bound at compile time, meaning there is no overhead of virtual tables or v-table pointers.

Summary

Polymorphism is one of the cornerstones of building extensible, flexible software in C++. Dynamic polymorphism, via substitution, virtual functions and Interfaces provide a mechanism to enact this. However, that flexibility comes at a run-time cost.

Templates offer a similar flexibility – and in many ways even *more* flexibility – without the run-time overheads of dynamic polymorphism. The trade-off now is the fact that the flexibility is fixed at compile-time – what we might call *Compile-Time polymorphism*.

The purpose of Interfaces (and Policies) is to provide architectural flexibility in our systems by defining clear code 'seams' – points in the architecture where elements can be removed and replaced with minimal effort. Many of those seams are related to hardware or other system factors and will never be reconfigured during system operation. In such cases it makes sense to replace Interfaces with template policies to help improve the run-time performance of the code.

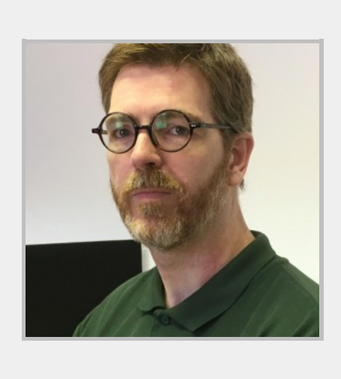
Next time, we'll explore

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 (12) | Dislike (0)



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++, Interfaces, Policy, Polymorphism, Templates. Bookmark the permalink.

← Template inheritance | Template member functions →

3 Responses to *Templates and polymorphism*

Kranti Madineni *says:*
April 21, 2018 at 12:02 pm

The best way to teach c++...covered not only templates and polymorphism but also design patterns...

Like (2) | Dislike (0)

mukesh *says:*
July 31, 2018 at 8:00 am

very nice example ....Glen

Like (0) | Dislike (0)

Greg *says:*
June 5, 2019 at 1:31 am

Explained very well...not all in this field can do both! This was helpful. Thank you.

Like (1) | Dislike (0)

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
- July 2018
- April 2018
- February 2018
- January 2018
- December 2017
- November 2017
- October 2017
- September 2017
- August 2017
- July 2017
- June 2017
- May 2017
- April 2017
- March 2017
- February 2017
- January 2017
- December 2016
- November 2016
- October 2016
- September 2016
- August 2016
- July 2016
- April 2016
- January 2016
- December 2015
- November 2015
- October 2015
- September 2015
- August 2015
- July 2015
- June 2015
- May 2015
- January 2015
- December 2014
- November 2014
- October 2014
- September 2014
- August 2014
- July 2014
- June 2014
- May 2014
- April 2014
- March 2014
- February 2014
- January 2014
- November 2013
- September 2013
- August 2013
- July 2013
- June 2013
- May 2013
- April 2013
- February 2013
- January 2013
- November 2012
- October 2012
- August 2012
- July 2012
- June 2012
- May 2012
- April 2012
- March 2012
- December 2011
- November 2011
- June 2011
- May 2011
- April 2011
- March 2011
- February 2011
- January 2011
- December 2010
- November 2010
- October 2010
- September 2010
- August 2010
- July 2010
- June 2010
- May 2010
- April 2010
- March 2010
- February 2010
- January 2010
- December 2009
- November 2009
- October 2009
- September 2009