thinkbottomup.com.au

C++ Mixins - Reuse through inheritance is good... when done the right way

15-19 minutes

Conventional Wisdom

Conventional wisdom tells us that *code reuse through inheritance is evil*. Go ahead and Google it if you're not clear on the stance in the community. The majority of articles that I've read encourage the programmer to favour *composition and delegation* (for example, by using the strategy pattern) over inheritance in order to attain code reuse. In general this advice is good as code-reuse-through-inheritance (as it is commonly implemented) does suffer from many problems. However, composition-and-delegation suffers from its own set of problems... So is there a better approach?

In this article I introduce the concept of the C++ Mixin. It will be shown that code reuse through inheritance in C++ mixins is not evil - in fact, it has so many advantages that it should be a technique commonly employed by C++ programmers.

Tutorials on C++ mixins tend to be a bit confusing and use contrived examples that make you wonder why you'd bother with this obsure technique. The intention with this article is to provide a easy to understand introduction to Mixins, their implementation and a comparison to other techniques for code reuse.

The example

Throughout this article a single example will be used. Consider that there is a Task Manager framework into which we can enqueue tasks for asynchronous execution. An interface for a task is defined as follows and must be implemented by all tasks that are to be executed by the framework:

```
1. struct ITask
2. {
3.     virtual ~ITask() = 0;
4.     virtual std::string GetName() = 0;
5.     virtual void Execute() = 0;
6. };
```

It is expected that there will be some common bits of functionality that should be reused across task implementations. I have selected *task execution timing* and *logging of start and completion messages* to serve as examples.

Approach 1: Code reuse through inheritance (evil)

First we will take the approach of *stick-the-reusable-stuff-in-a-base-class* and see how we fare. Let's first attack the case of logging start and completion messages:

1. `// abstract base class which implements Execute() and provides logging feature. Derived classes must implement OnExecute().`

2. `class BaseLoggingTask : public ITask`

3. `{`

4. `public:`

5. `    virtual void Execute()`

6. `    {`

7. `        std::cout << "LOG: The task is starting - " << GetName().c_str() << std::endl;`

8. `        OnExecute();`

9. `        std::cout << "LOG: The task has completed - " << GetName().c_str() << std::endl;`

10. `    }`

11.

```
12.        virtual void OnExecute() = 0;

13. };

14.

15. // Concrete Task implementation that reuses
    the logging code of the BaseLoggingTask

16. class MyTask : public BaseLoggingTask

17. {

18. public:

19.     virtual void OnExecute()

20.     {

21.         std::cout << "...This is where the
    task is executed..." << std::endl;

22.     }

23.

24.     virtual std::string GetName()

25.     {

26.         return "My task name";

27.     }

28. };
```

On the face of it, the above code looks reasonable - we have pushed the reusable code into the base class. The code is quite concise and simple to read. We can do the same thing

for the task timing code:

```
1.  // abstract base class which implements
    Execute() and provides task timing feature.
    Derived classes must implement OnExecute().

2.  class BaseTimingTask : public ITask

3.  {

4.      Timer timer_;

5.

6.  public:

7.      virtual void Execute()

8.      {

9.          timer_.Reset();

10.         OnExecute();

11.         double t =
    timer_.GetElapsedTimeSecs();

12.         std::cout << "Task Duration: " << t <<
    " seconds" << std::endl;

13.     }

14.

15.     virtual void OnExecute() = 0;

16. };

17.
```

```
18. // Concrete Task implementation that reuses
    the timing code of the BaseTimingTask

19. class MyTask : public BaseTimingTask

20. {

21. public:

22.     virtual void OnExecute()

23.     {

24.         std::cout << "...This is where the
    task is executed..." << std::endl;

25.     }

26.

27.     virtual std::string GetName()

28.     {

29.         return "My task name";

30.     }

31. };
```

In the second case, MyTask has reused task timing code instead of task logging code. Great! But there is a serious problem...

The problem occurs when we want to write a class that reuses both the timing and logging code - it quite simply can not be done and this shows up a major limitation of the *stick-the-*

reusable-stuff-in-a-base-class approach. At some point you will discover that you can not combine the reusable parts in the way you want. What we would really like is an approach whereby we can reuse code in various combinations (like reuse both the logging and timing code in one class).

Another problem with this approach is the use of virtual functions. We have virtual functions calling virtual functions when we are trying to something relatively simple! It should be noted that the compiler can not generally inline virtual functions and there is some overhead in calling a virtual function compared to calling a non-virtual function. This runtime hit seems unreasonable, but how can we overcome it?

Approach 2: Code reuse through composition and delegation (still evil, just not as much)

Let's revisit the problem by exploring composition and delegation. In implementing an interface, an object can delegate part of the implementation to another object that it contains.

We can rewrite the reusable logging and timing code to work with composition rather than inheritance. Let's look at the LoggingTask first:

```
1. class LoggingTask : public ITask
2. {
```

```
  3.        ITask* task_;

  4.  public:

  5.        LoggingTask( ITask* task ) : task_( task )

  6.        {

  7.        }

  8.

  9.        ~LoggingTask()

 10.        {

 11.            delete task_;

 12.        }

 13.

 14.        virtual void Execute()

 15.        {

 16.            std::cout << "LOG: The task is
    starting - " << GetName().c_str() <<
    std::endl;

 17.            if( task_ ) task_->Execute();

 18.            std::cout << "LOG: The task has
    completed - " << GetName().c_str() <<
    std::endl;

 19.        }

 20.
```

```
21.       virtual std::string GetName()
22.       {
23.           if( task_ )
24.           {
25.               return task_->GetName();
26.           }
27.           else
28.           {
29.               return "Unbound LoggingTask";
30.           }
31.       }
32. };
```

In principal this is quite simple - the LoggingTask implements ITask and and is passed a pointer to an ITask in its constructor - let's call that the child task. The LoggingTask delegates its implementation of GetName() to the child task. It also delegates Execute() to the child task but performs logging before and after the delegation.

The timing task is implemented in a similar manner:

```
1. class TimingTask : public ITask
2. {
3.     ITask* task_;
```

```
4.      Timer timer_;

5.  public:

6.      TimingTask( ITask* task ) : task_( task )

7.      {

8.      }

9.

10.     ~TimingTask()

11.     {

12.         delete task_;

13.     }

14.

15.     virtual void Execute()

16.     {

17.         timer_.Reset();

18.         if( task_ ) task_->Execute();

19.         double t =
    timer_.GetElapsedTimeSecs();

20.         std::cout << "Task Duration: " << t <<
    " seconds" << std::endl;

21.     }

22.
```

```
23.     virtual std::string GetName()
24.     {
25.         if( task_ )
26.         {
27.             return task_->GetName();
28.         }
29.         else
30.         {
31.             return "Unbound TimingTask";
32.         }
33.     }
34. };
```

The above classes can be used in combination to add timing and logging to your task as follows:

```
1. // MyTask is written with no coupling to the
   reusable logging and timing task
2. class MyTask : public ITask
3. {
4. public:
5.     virtual void Execute()
6.     {
```

```
 7.          std::cout << "...This is where the
     task is executed..." << std::endl;

 8.      }

 9.

10.     virtual std::string GetName()

11.     {

12.          return "My task name";

13.     }

14. };

15.

16. // Add timing and logging to MyTask through
    composition and delegation

17. ITask* t = new LoggingTask(

18.                    new TimingTask(

19.                        new MyTask() ) );

20. t->Execute();

21. delete t;
```

While the principal may be simple, that looks a lot more complex than it was in our first example! There are some reasons for this:

- We need to think about the lifetime of the child task. In my design I have decided that the LoggingTask adopts ownership

of the child task and will delete the child task in its destructor.

- Since a null pointer may be passed into the constructor, we need to perform runtime checks before dereferencing the child pointer.

- In the event of a null pointer to the child task, the LoggingTask has to return a dummy name from GetName(). This is horrible! The Logging task doesn't really want to provide any logic in its GetName() method, but it is forced to!

The problems continue. Did you notice that we're forced to use heap allocations? Since the parent deletes the child, the child must be heap allocated. So, to add to the performance issues noted in the first approach, we now have heap allocations and runtime checks for null pointers. This seems like a whole lot of backward steps.

I shouldn't be so pessimistic; there are some positive aspects of this approach. It has overcome the issue of reusing both the logging and timing task in one class and it's quite straightforward to compose the reusable parts. The way that each part is decoupled is also very good - unlike the stick-the-reusable-stuff-in-a-base-class approach where the MyTask was tightly coupled to the reusable part through inheritance. Another positive is that each component (TimingTask, LoggingTask and MyTask) all directly implement the ITask interface. There is no special "OnExecute()" virtual method or other odd things going on.

With low coupling and high cohesion, we'd have to call this a

good approach to attaining code reuse, right? Well, no - not in my book. I just can't get past it's flaws:

- The heap allocations and runtime checks are surely unnecessary, but the design pattern forces this upon us.

- Classes like the LoggingTask don't really want to provide an implementation of GetName(), but are forced to provide an implementation. Imagine if the ITask interface had a dozen methods and your class wants to simply delegate 11 of them; you'd end up writing more plumbing code than real logic!

These problems make this approach *still evil* in my mind. Let's try to find a technique that overcomes these problems without introducing problems like we saw in the stick-the-reusable-stuff-in-a-base-class approach.

Approach 3: Code reuse through inheritance - revisted (Clayton's reuse)

In this approach we turn the idea of stick-the-reusable-stuff-in-a-base-class on its head and instead stick-the-reusable-stuff-in-a-derived-class. I call this Clayton's reuse; the reuse you have when you're not having reuse. While this section might seem silly, it's a stepping stone to understanding C++ mixins, so please bear with me.

Consider our simple task that we've used as an example so far:

```
1. class MyTask : public ITask
```

```
2.  {

3.  public:

4.      virtual void Execute()

5.      {

6.          std::cout << "...This is where the
    task is executed..." << std::endl;

7.      }

8.

9.      virtual std::string GetName()

10.     {

11.         return "My task name";

12.     }

13. };
```

We can add timing to this class by creating a subclass:

```
1.  class TimingTask : public MyTask

2.  {

3.  protected:

4.      virtual void Execute()

5.      {

6.          std::cout << "(start timer)" <<
    std::endl;
```

7.   MyTask::Execute();

8.   std::cout << "(end timer)" << std::endl;

9.   }

10. };

And we can add logging to the TimingTask:

1. class LoggingTask : public TimingTask

2. {

3. protected:

4.   void Execute()

5.   {

6.   std::cout << "LOG: The task is starting: " << GetName().c_str() << std::endl;

7.   TimingTask::Execute();

8.   std::cout << "LOG: The task has completed: " << GetName().c_str() << std::endl;

9.   }

10. };

Ok, so what have we achieved? The logging code is coupled to the timing code which is coupled to MyTask and none of it is particularly reusable. So what's the point?

The point is that other than the coupling and lack of reuse, we have overcome some problems:

- We are not forced into using heap allocations

- There are no dubious runtime checks

- No object lifetime management decision to be made

- No superfluous virtual functions defined

- No superfluous virtual function calls

- The compiler has greater opportunity for optimisation, such as the ability to inline the Execute() calls defined in base classes

With all these positives, this is worth pursuing - all we need to do is work out how to decouple the classes and allow for reuse. It turns out that this silly looking example is just one small step away from greatness!

Approach 4: Code reuse through mixins

Although Mixins have a fairly broad definition, I think it's quite well understood that in C++ circles a Mixin refers to the following idiom - a template class that is parameterised on its base class:

```
1. template< class T >
2. class MyMixin : public T
3. {
4. };
```

If this is the first time you've seen this construct, you probably need to pause and think about what this means for a moment... A class's base class is supplied by a template parameter... A template class that has a different base class for each template instantiation... It turns out that this is the key to fixing the problems in our Clayton's reuse example!

Let's revist the LoggingTask:

```
1.  class LoggingTask : public TimingTask
2.  {
3.  protected:
4.      void Execute()
5.      {
6.          std::cout << "LOG: The task is starting: " << GetName().c_str() << std::endl;
7.          TimingTask::Execute();
8.          std::cout << "LOG: The task has completed: " << GetName().c_str() << std::endl;
9.      }
10. };
```

The problem with the LoggingTask is that it is coupled to the TimingTask even though logging and timing are orthogonal concepts. The decoupling is simple - we just supply the base

class of the LoggingTask as a template parameter:

1. `template< class T >`

2. `class LoggingTask : public T`

3. `{`

4. `public:`

5. `    void Execute()`

6. `    {`

7. `        std::cout << "LOG: The task is starting - " << GetName().c_str() << std::endl;`

8. `        T::Execute();`

9. `        std::cout << "LOG: The task has completed - " << GetName().c_str() << std::endl;`

10. `    }`

11. `};`

Hey, that's a Mixin!

So long as type T provides an Execute() method (as LoggingTask calls T::Execute()) then this code will compile and run as expected. Let's rewrite our example using mixins:

1. `template< class T >`

2. `class LoggingTask : public T`

```
3.  {
4.  public:
5.      void Execute()
6.      {
7.          std::cout << "LOG: The task is
    starting - " << GetName().c_str() <<
    std::endl;
8.          T::Execute();
9.          std::cout << "LOG: The task has
    completed - " << GetName().c_str() <<
    std::endl;
10.     }
11. };
12.
13. template< class T >
14. class TimingTask : public T
15. {
16.     Timer timer_;
17. public:
18.     void Execute()
19.     {
20.         timer_.Reset();
```

```
21.            T::Execute();

22.            double t =
     timer_.GetElapsedTimeSecs();

23.            std::cout << "Task Duration: " << t <<
     " seconds" << std::endl;

24.        }

25. };

26.

27. class MyTask

28. {

29. public:

30.     void Execute()

31.     {

32.          std::cout << "...This is where the
     task is executed..." << std::endl;

33.     }

34.

35.     std::string GetName()

36.     {

37.          return "My task name";

38.     }
```

39. `};`

In the above, the TimingTask and the LoggingTask are written as Mixins while MyTask is not. This is to be expected as MyTask is not reusable and represents a concrete concept rather than the abstract concepts embodied in the Mixins.

Building up a class using the Mixins is quite easy. Some examples are shown below:

```
1.  // A plain old MyTask

2.  MyTask t1;

3.  t1.execute();

4.

5.  // A MyTask with added timing:

6.  TimingTask< MyTask > t2;

7.  t2.Execute();

8.

9.  // A MyTask with added logging and timing:

10. LoggingTask< TimingTask< MyTask > > t3;

11. t3.Execute();

12.

13. // A MyTask with added logging and timing
    written to look a bit like the composition
    example
```

```
14. typedef LoggingTask<

15.            TimingTask<

16.                MyTask > > Task;

17. Task t4;

18. t4.Execute();
```

But hang on a minute, none of this helps us plug into our Task Manager framework as the classes do not implement the ITask interface. This is where one final Mixin helps - a Mixin which introduces the ITask interface into the inheritance hierarchy, acting as an adapter between some type T and the ITask interface:

```
1. template< class T >

2. class TaskAdapter : public ITask, public T

3. {

4. public:

5.     virtual void Execute()

6.     {

7.         T::Execute();

8.     }

9.

10.    virtual std::string GetName()

11.    {
```

```
12.            return T::GetName();
13.      }
14. };
```

Using the TaskAdapter is simple - it's just another link in the chain of mixins.

```
1. // typedef for our final class, inlcuding the
   TaskAdapter<> mixin

2. typedef public TaskAdapter<

3.                      LoggingTask<

4.                        TimingTask<

5.                          MyTask > > > task;

6.

7. // instance of our task - note that we are not
   forced into any heap allocations!

8. task t;

9.

10. // implicit conversion to ITask* thanks to the
    TaskAdapter<>

11. ITask* it = &t;

12. it->Execute();
```

Conclusion

Looking back at the problems with the previous approaches, it should be clear that Mixins avoid the problems of Composition and Delegation while preserving the advantages of Clayton's reuse and, once you understand the pattern, looks as simple as stick-it-in-the-base-class reuse.

The Visual Studio 2008 solution attached to this post contains all of the examples from this article.