C++ INTERFACE CLASSES – NOISE REDUCTION

*Interface class*es are a principle mechanism for separating a class' interface from its implementation in C++. I wrote an introduction to interface classes in a previous article [Radford04], and Alan Griffiths and I included the technique in our survey of techniques for separating interface and implementation in C++ [Griffiths05].

In this article, I intend to explore interface classes – and their *implementation classes* – further. The topics I plan to cover are:

- How interface and implementation classes can be designed into the code in such a way as to reduce implementation "noise"
- How factory functions can be used to facilitate the above
- A way of managing instance lifecycles when factory functions are used to encapsulate different memory allocation mechanisms

An Example Class

In [Griffiths05] Alan and I used `telephone_list` – a telephone address book class – in order to illustrate several C++ interface/implementation separation techniques. Here I will again use (a slightly modified version of) that example.

The `telephone_list` interface class looks like this:

```
class telephone_list
{
public:
  virtual ~telephone_list()   {}

  virtual std::string name() const = 0;

  virtual std::pair<bool, std::string>
  number(const std::string& person) const = 0;

  virtual telephone_list&
  add_entry(const std::string& name,
            const std::string& number) = 0;

protected:
  telephone_list() {}
  telephone_list(const telephone_list& rhs) {}
private:
  telephone_list& operator=(const telephone_list& rhs);
};
```

In order for this to have functionality, and in order for instances to be created, an implementation class is needed – I'm going to call it `telephone_list_imp`:

```
class telephone_list_imp : public telephone_list
{
public:
  telephone_list_imp(const std::string& list_name);

private:

  virtual ~telephone_list_imp();

  virtual std::string name() const;

  virtual std::pair<bool, std::string>
  number(const std::string& person) const;

  virtual telephone_list&
  add_entry(const std::string& name,
            const std::string& number);

  typedef std::map<std::string, std::string> dictionary_t;

  std::string  name_rep;
  dictionary_t dictionary_rep;

  telephone_list_imp(const telephone_list_imp& rhs);
  telephone_list_imp& operator=(const telephone_list_imp& rhs);
};
```

In [Radford04] I also described *implementation only* classes, and this is an approach I have applied here. Apart from the constructors, all member functions have been made private. This strengthens the separation of interface from implementation by making it possible to create instances of `telephone_list_imp`, while usage must be via pointers and/or references to `telephone_list`.

Hiding the Implementation and Creating Instances

This whole design is geared up to functionality being used through pointers/references to `telephone_list`. Therefore, the only reason to make the definition of `telephone_list_imp` visible to client code is so that instances can be created. It follows that client code has to carry a certain amount of "noise" – in the form of the publicly visible definition of `telephone_list_imp` – just so instances can be created.

Further, C++ has the problem of physical dependencies between header files, and the consequent recompilations that result from modifications being made to them. This is a consequence of the file inclusion model inherited from C. Let's say for the sake of an example, that one day `telephone_list_imp` is modified, abandoning the `std::map` implementation in favour of a different container. The fact that client code – which has no dependency on the modified implementation detail – needs to recompile, emphasises the fact that `telephone_list_imp` is just noise to the client code.

The two issues discussed above add up to the fact that it would be better if `telephone_list_imp`'s definition could be kept out of client code altogether. Ideally, the best place for the definition of `telephone_list_imp` is in an implementation (typically .cpp) file. This leads to another problem of how clients can create instances, but this is straightforward to solve: in the header file, provide a factory function for creating instances of `telephone_list_imp`. The header file *telephone_list_imp.h* (with include "guards" removed for brevity) now looks like this:

```
#include "telephone_list.h"
#include <string>

telephone_list* create_telephone_list(const std::string& list_name);
```

Note that there is no mention of the implementation class – the `telephone_list` interface class is all that's needed. In fact, only a forward declaration of `telephone_list`

is needed – however, the header file has been included because users might reasonably expect that when they write "`#include "telephone_list_imp.h"`" in client code, the base class' definition will be made available.

The fragment of the implementation file containing the `telephone_list_imp` and factory function definition looks like this:

```
class telephone_list_imp : public telephone_list
{
public:
  telephone_list_imp(const std::string& list_name);

private:

  virtual ~telephone_list_imp();

  virtual std::string name() const;

  virtual std::pair<bool, std::string>
  number(const std::string& person) const;

  virtual telephone_list&
  add_entry(const std::string& name,
            const std::string& number);

  typedef std::map<std::string, std::string> container;

  std::string  name_rep;
  container dictionary_rep;

  telephone_list_imp(const telephone_list_imp& rhs);
  telephone_list_imp& operator=(const telephone_list_imp& rhs);
};

telephone_list* create_telephone_list(const std::string& list_name)
{
  return new telephone_list_imp(list_name);
}
...
```

At this point in the exercise the aim of removing `telephone_list`'s implementation from having visibility in client code has been achieved. Clients deal with pointers/references to `telephone_list`s, while `telephone_list_imp` remains buried safely in its implementation file. All should be well, but the solution to one problem has created another…

How are Instances Deleted?

There are two observations to make about the ("naïve") implementation of `create_telephone_list()`:

- The mechanism used to create instances is now encapsulated and hidden from public view
- So far, the return type is a simple pointer to `telephone_list`

This means clients can apply the `delete` operator to pointers returned from `create_telephone_list()`. However, they have to rely on documentation to know they must do this. There is no way it can be made clear in the code, and clients can't assume it because using the delete operator is not compatible with all mechanisms for allocating class instances on the heap in C++. A solution to the problem (not the only one) is, rather than return a simple pointer, to return a smart pointer such as Boost's `shared_ptr` (see [Boost]). The (fragmented form of the) header file *telephone_list_imp.h* now looks like this:

```
#include "telephone_list.h"
#include "boost/shared_ptr.hpp"
#include <string>

boost::shared_ptr<telephone_list> create_telephone_list(const std::string& list_name);
```

While the implementation of `create_telephone_list()` now looks like this:

```
boost::shared_ptr<telephone_list> create_telephone_list(const std::string& list_name)
{
  telephone_list* p = new telephone_list_imp(list_name);
  return boost::shared_ptr<telephone_list>(p);
}
```

In passing note the avoidance of the expression:

```
return boost::shared_ptr<telephone_list>(new telephone_list_imp(list_name));
```

This is because `boost::shared_ptr` "remembers" the concrete type created, and uses it when the instance is deleted – i.e. `telephone_list_imp` having a private destructor means `boost::shared_ptr`'s attempt to `delete` via it causes a compile error. The mechanism used ensures that the "remembered" type is `telephone_list`, and thus avoids compilation problems. Another option is simply to make `telephone_list_imp`'s destructor public. I chose the option in the code fragment because it adheres to the principle of all usage being through the interface class.

The above approach solves the problem of deleting the instance that had its creation mechanism encapsulated. The cost of achieving this is the hard-wiring of a specific smart pointer into the code. Further, there is a remaining problem that it doesn't solve…

Different Allocation Mechanisms

The memory allocation scheme used so far is not the only one available in C++. For example, the placement form of `new` could be used to construct instances in conjunction with using `malloc()`to allocate the memory. However, if `create_telephone_list()` returns a simple pointer and relies on the client code to apply the `delete` operator, then there's no way its implementation can ever be changed to use an alternative allocation mechanism.

In some design scenarios, as well as having a factory function to create instances, it is possible to have a disposal function to delete them. However in the design scenario under consideration, there is a serious drawback to this approach. The implementation class `telephone_list_imp` is implemented in a way that results in particular complexity characteristics – i.e. those associated with its implementation container `std::map`. Imagine that there arises a need for a second implementation with different characteristics. Why this may be so is outside the scope of this article – suffice to say that if this is done, `telephone_list_imp` ceases to be the only implementation of `telephone_list` in town. Getting back to disposing of instances, it is not hard to see that in order for clients to pass instances to a disposal function, either instances of each implementation class would need to use the same memory allocation mechanism, or disposal functions would need some way of recovering the implementation class from a pointer/reference to the interface class.

The analysis of the complexities and tradeoffs involved in using disposal function may at some point be the subject of another article, but in this one I want to look at a different approach. The approach I want to look at involves associating a disposal function with class instances at the time their factory function creates them. Here, just for illustration's sake, is a fragment of a home grown smart pointer that achieves this:

```
template <typename T> class ref_counted_ptr
{
public:
  ref_counted_ptr(T* p, void (*delete_fn)(T*))
    : pointee(p),
      del(delete_fn)
  {
     ...
  }

  ~ref_counted_ptr() { del(pointee); }

  T* operator->()
  {
    return pointee;
  }
...
private:
  T* pointee;
  void (*del)(T*);
...
};
```

Using `ref_counted_ptr`, the declaration of the factory function now looks like this:

```
ref_counted_ptr<telephone_list> create_telephone_list(const std::string& list_name);
```

Its implementation now looks like this:

```
ref_counted_ptr<telephone_list> create_telephone_list(const std::string& list_name)
{
  telephone_list* mem =
    static_cast<telephone_list*>(std::malloc(sizeof telephone_list_imp));

  if (!mem)
    throw std::bad_alloc();

  telephone_list* pobj = new (mem) telephone_list_imp(list_name);

  return ref_counted_ptr<telephone_list>(pobj, del_telephone_list);
}
```

As if by magic, a function called `del_telephone_list()` has appeared in the above code fragment – it looks like this:

```
void del_telephone_list(telephone_list* p)
{
  p->~telephone_list();
  std::free(p);
}
```

However, as I said, `ref_counted_ptr` is for illustration only. There is actually no need to write a custom smart pointer just to associate a disposal function with a class instance, because `boost::shared_ptr` has a mechanism for accommodating a disposal function. Actually, `boost::shared_ptr` has a somewhat more sophisticated mechanism that allows the disposal function to be either a pointer to a function, or a function object. For this article, I'll stick to the approach already used – that of using `del_telephone_list()` as shown above. The factory function implementation now looks like this:

```
boost::shared_ptr<telephone_list> create_telephone_list(const std::string& list_name)
{
  telephone_list* mem =
    static_cast<telephone_list*>(std::malloc(sizeof telephone_list_imp));

  if (!mem)
    throw std::bad_alloc();

  telephone_list* pobj = new (mem) telephone_list_imp(list_name);

  return boost::shared_ptr<telephone_list>(pobj, del_telephone_list);
}
```

In passing I should mention that there are various tradeoffs in possible implementations of reference counted smart pointers, and `boost::shared_ptr` addresses only one set of

tradeoffs. I just thought it best to point that out; sorry, but I'm not going into any more detail on that topic. The reader is referred to the Boost documentation (see [Boost]).

Finally

Cases where solutions to problems are *the* solutions are rare – usually there are alternatives that come with their own sets of tradeoffs. I hope I have succeeded in making the tradeoffs clear. This article has covered the three points set out in the introduction, having followed one train of thought. Others have been alluded to in passing but not covered, but perhaps in future articles …

References

[Boost] www.boost.org

[Griffiths05] Alan Griffiths and Mark Radford, *Separating Interface and Implementation in C++*, (Overload, and also available at http://www.twonine.co.uk/articles/SeparatingInterfaceAndImplementation.pdf)

[Radford04] Mark Radford, *C++ Interface Classes – An Introduction* (Overload 62, and also available from http://www.twonine.co.uk/articles/CPPInterfaceClassesIntro.pdf)

Mark Radford

(mark@twonine.co.uk)