# C++ INTERFACE CLASSES – AN INTRODUCTION

Class hierarchies that have run-time polymorphism as one of their prominent characteristics are a common design feature in C++ programs, and with good design, it should not be necessary for users of a class to be concerned with its implementation details. One of the mechanisms for achieving this objective is the separation of a class' interface from its implementation. Some programming languages, e.g. Java, have a mechanism available in the language for doing this. In Java, an *interface* can contain only method signatures. In C++ however, there is no such first class language feature, and the mechanisms already in the language must be used to emulate interfaces as best as can be achieved. To this end, an *interface class* is a class used to *hoist* the polymorphic interface – i.e. pure virtual function declarations – into a base class. The programmer using a class hierarchy can then do so via a base class that communicates only the interface of classes in the hierarchy.

## Example Hierarchy

The much used *shape* hierarchy example serves well here. Let's assume for the sake of illustration, that we have two kinds of shape: `arc` and `line`. The hierarchy therefore, contains three *abstractions*: the `arc` and `line` concrete classes, and the generalisation `shape`. From now on, I'll talk mainly about `shape` and `line` only – the latter serving as an illustration of an implementation. These two classes, in fragment form, look like this:

```
class shape
{
public:
  virtual ~shape();

  virtual void move_x(distance x) = 0;
  virtual void move_y(distance y) = 0;
  virtual void rotate(angle rotation) = 0;

//...
};

class line : public shape
{
public:
  virtual ~line();

  virtual void move_x(distance x);
  virtual void move_y(distance y);
  virtual void rotate(angle rotation);
private:
  point end_point_1, end_point_2;
//...
};
```

The `shape` abstraction is expressed here as an *interface class* – it contains nothing but pure virtual function declarations. This is as close as we can get in C++ to expressing an interface. Adding to the terminology, classes such as `line` (and `arc`) are known as *implementation classes.*

Now let's assume this hierarchy is to be used in a two dimensional drawing package. It seems reasonable to suggest that in this package, `drawing` may be another useful abstraction. `drawing` could be expressed as an interface class, like in this fragment:

```
class drawing
{
public:
    virtual ~drawing();

    virtual void add(shape* additional_shape) = 0;

//...
};
```

Besides the virtual destructor, only one member function of drawing – the `add()` virtual function – is shown. Note that `drawing` does not collaborate with any implementation of `shape`, but only with the interface class `shape`. This is sometimes known as abstract coupling – `drawing` can talk to any class that supports the `shape` interface.

## Benefits

Having explained the technique of *hoisting* a class' interface, I need to explain why developers should be interested in doing this. There are three points:

- Hoisting the (common) interface of classes in a run-time polymorphic hierarchy affords a clear separation of interface from implementation. Further, doing so helps to underpin the use of abstraction, because the interface class expresses only the capabilities of the abstracted entity.
- It follows on from the above, that new implementations can be added without changing existing code. For example, it is most likely that drawing will initially have only one implementation class, but because other code is dependent only on its interface class, new implementations *can* easily be added in the future.
- Consider the physical structure of C++ code with regard to the interface class, its implementation classes, and classes (such as `drawing`) that use it. Assuming common C++ practice is followed, the definition of `shape` will have a header file – let's assume it's called `shape.hpp` – all to itself, as will `drawing` (i.e. `drawing.hpp`, using the same convention). Now, owing to the physical structure of C++ (that is, the structure it inherited from C), if anything in the `shape.hpp` header file is changed, anything that depends on it – such as `drawing.hpp` – must recompile. In large systems where build times are measured in hours (or even days), this can be a significant overhead. However, because `shape` is an interface class, `drawing` (for example) has no physical dependency on any of the implementation detail, and it is in the implementation detail that change is likely to occur (assuming some thought has been put into the design of `shape`'s interface).

## Strengthening the Separation

Returning to the first point above for a moment, there is a way by which we can strengthen the logical separation further: we can make `shape`'s implementation classes into *implementation only* classes. This means that in the implementation classes, all the virtual member functions are made *private*, leaving only their constructors publicly accessible. The `line` class then looks like this:

```
class line : public shape
{
public:
  line(point end_point_1, point end_point_2);
//...
private:
  virtual ~line();
  virtual void move_x(distance x);
  virtual void move_y(distance y);
  virtual void rotate(angle rotation);

//...
};
```

Now, the only thing users can do with `line` is create instances of it. All usage must be via its interface – i.e. `shape`, thus enforcing a stronger interface/implementation separation. Before leaving this topic, it is important to get something straight: the point of enforcing the interface/implementation separation is *not* to tell users what to do. Rather, the objective is to underpin the logical separation – the *code* now explains that the key abstraction is `shape`, and that `line` serves to provide an implementation of `shape`.

## Mixin Interfaces

As a general design principle, all classes should have responsibilities that represent a primary design role played by the class. However, sometimes a class must also express functionality representing responsibilities that fall outside its primary design role. In such cases, the need for partitioning of this functionality is pressing, and interface classes have a part to play.

A class that expresses this kind of extra functionality is called a *mixin*. For example, it is easy to imagine there might be a requirement to store and retrieve the state of `shape` objects. However, storage and retrieval functionality is not a responsibility of `shape` in the application domain model. Therefore, a feasible design is as follows:

```
class serialisable
{
public:
  virtual void load(istream& in) = 0;
  virtual void save(ostream& out) = 0;
protected:
    ~serialisable();
};


class shape : public serialisable
{
public:
  virtual ~shape();

  virtual void move_x(distance x) = 0;
  virtual void move_y(distance y) = 0;
  virtual void rotate(angle rotation) = 0;

// No declarations of load() or save() in this class
...
};
```

```
class line : public shape
{
public:
  line(point end_point_1, point end_point_2);
//...
private:
  virtual ~line();
  virtual void move_x(distance x);
  virtual void move_y(distance y);
  virtual void rotate(angle rotation);

  virtual void load(istream& in);
  virtual void save(ostream& out);

//...
};
```

This approach is intrusive to a degree because `serialisable`'s virtual member functions must be declared in `line`'s interface. However, at least there is a separation in that `serialisable` is kept separate from the crucial `shape` abstraction.

Note that `serialisable` does not have a public virtual destructor – its destructor is protected and non-virtual. `serialisable` is not intended that pointers to `serialisable` are held and passed around in a program – i.e. it is not a *usage* type, that's the role of the `shape` class. Making the destructor non-virtual and not publicly accessible allows the code to state this explicitly, without recourse to any further documentation.

Often mixin functionality is added to a class using multiple inheritance. Here there is an analogy with Java, in which there is direct language support for interfaces. In Java, a class can inherit from one other class, but can implement as many interfaces as desired. The same thing can be emulated in C++ using interface classes, but in C++ there is an added twist – C++ has *private* inheritance to offer. This approach comes in handy particularly when the usage type is outside the control of the programmer – for example, because it is part of a third party API. For example, consider a small framework where notifications are sent out by objects of type `notifier`, and received by classes supporting an interface defined by `notifiable`. The two interface classes (or fragment of, in the case of `notifier`) are defined as follows:

```
class notifiable
{
public:
    virtual void update() = 0;
protected:
    ~ notifiable();
};
```

```
class notifier
{
public:
  virtual void register_client(notifiable* o)=0;
    ...
};
```

Now consider using a GUI toolkit that provides a base class called `window`, from which all window classes are to be derived. The programmer wishes to write a class called `my_window` that receives notifications from objects of type `notifier` – such a class could look like this:

```
class my_window : public window, private notifiable
{
public:
    void register_for_notifications(notifier& n)
    { n.register_client(this); }
    ...
};
```

Using private inheritance has rendered the `notifiable` interface inaccessible to clients, but allows `my_window` use of it, because like anything else that's private to `my_window`, its private base classes are accessible in its member functions. This approach helps to strengthen the separation of concerns which the use of mixin functionality seeks to promote.

## Interface Class Emulation Issues

The fact that we have to consider emulation issues at all is owing to the fact that interfaces are being *emulated* rather than being a first class language feature – all part of the fun of using C++! I think there are issues in two areas, i.e. those concerned with:

- An interface class' interface
- Deriving from an interface class

### An interface class' interface

Consider the interface class `shape`:

```
class shape
{
public:
  virtual ~shape();

  virtual void move_x(distance x) = 0;
  virtual void move_y(distance y) = 0;
  virtual void rotate(angle rotation) = 0;
  // other virtual function declarations...
};
```

If we write only the above, the compiler will step in and provide: a copy assignment operator, a default constructor, and a copy constructor. I think we can safely say that, an interface class' run time polymorphic behaviour points to assignment semantics being inappropriate and irrelevant. Therefore, the assignment operator should be private and not implemented:

```
class shape
{
public:
  ...
private:
  shape& operator=(const shape&);
};
```

Interface classes are stateless by their nature, so allowing assignment is harmless, but prohibiting it is a simple contribution to avoiding errors.

What about the default constructor and a copy constructor? Here we should just thank the compiler and take what is on offer, as this is the easiest way to avoid any complications. Note that declaration of constructors by the programmer has potential pitfalls. For example, if a copy constructor only is declared, then the compiler will not generate a default constructor.

### Deriving from an interface class

Consider the following fragment that shows `line` being derived from `shape` (as one would expect):

```
class line : public shape
{
public:
  line(int in_x1, int in_y1, int in_x2, int in_y2)
  : x1(in_x1), y1(in_y1), x2(in_x2), y2(in_y2)
  {}
  ...
  private:
    int x1, y1, x2, y2;
};
```

The programmer has declared a constructor that initialises `line`'s state, but not specified which of `shape`'s constructors is to be called. As a result the compiler generates a call to

`shape`'s default constructor. So far this is fine. Because `shape` is stateless it doesn't matter how it gets initialised.

However, that's not the end of the story …

It is a common design re-factoring in C++ (and several other languages), to hoist common state out of concrete classes, and place it in a base class. So if common implementation is found between `shape`'s derived classes `line` and `arc`, rather than have a two tier hierarchy, it is reasonable to have a three tier hierarchy. For the sake of an example, let's assume that it is necessary for all `shape`s to maintain a *proximity rectangle* – i.e. if a point falls within the rectangle, the point is considered to be in close proximity to the `shape`. This functionality can then, for example, be used to determine if a `shape` object should be selected when the user click's the mouse near by.

I'm going to assume a suitable `rectangle` class is in scope, and introduce `shape_impl` to contain the common implementation.

```
class shape_impl : public shape
{
private:
  virtual ~shape_impl() = 0;

  virtual void move_x(distance x);
  virtual void move_y(distance y);
  virtual void rotate(angle rotation);
  //...
protected:
  shape_impl();
  shape_impl(const rectangle& initial_proximity);
  //...
private:
  rectangle proximity;
  // ...
};
```

The implementation class `shape_impl` is abstract, as shown by the pure virtual destructor. As a brief digression, it is also an *implementation only* class – its implementation of `shape`'s interface has been declared as private so clients can create instances, but can't call any of the member functions.

Now look what happens if `line`'s base class is changed, but changing the constructor used to initialise the base class gets forgotten about.

```
class line : public shape_impl
{
public:
  line(int in_x1, int in_y1, int in_x2, int in_y2)
  : x1(in_x1), y1(in_y1), x2(in_x2), y2(in_y2)
  {}
  ...
  private:
    int x1, y1, x2, y2;
};
```

This will compile, and fail at run time. However, if in the first place the programmer had written:

```
class line : public shape
{
public:
  line(int in_x1, int in_y1, int in_x2, int in_y2)
  : shape(), x1(in_x1), y1(in_y1), x2(in_x2), y2(in_y2)
  {}
  ...
};
```

In the latter case, changing the base class to `shape_impl` would cause a compile error, because `shape` is no longer the immediate base class. This leads me to make the following recommendation: *always call an interface class' constructor explicitly*.

## Finally

Interface classes are fundamental to programming with run time polymorphism in C++. Despite this, I'm all too frequently surprised by how little they are known about by the C++ programmers out there.

This article doesn't cover everything: for example, the use of virtual inheritance when deriving from mixins is something I hope to get around to covering in a future article. However, I hope this article serves as a reasonable introduction.

## Acknowledgements

Many thanks to Phil Bass, Thaddaeus Frogley and Alan Griffiths for their feedback.