Xcpp Interfaces

David Barrett-Lennard
Cedanet Pty Ltd
Perth, Western Australia
david.barrettlennard@cedanet.com.au

March 24, 2010

Abstract

Xcpp adds support for interfaces to C++ without using virtual functions. This article presents the syntax and discusses the advantages over using abstract base classes for dynamic polymorphism.

1 Overview

The xcpp preprocessor is applied to the source code before subsequent compilation by a standard C/C++ compiler. The `$interface` directive allows for dynamic polymorphic interfaces to be defined, in a way that is more flexible and less type intrusive than the normal support for dynamic polymorphism in C++.

The conventional approach to support interface-like functionality in C++ is to use abstract base classes (ABCs). Dynamic polymorphism is typically achieved using a vtable pointer within the object that implements the interface. An alternative approach is for clients to hold a pair of pointers - i.e. a vtable pointer in addition to the pointer to the actual object. This fully decouples interface support from object implementation. The technique has been described in the literature. See for example [1] and the Boost Interfaces Library [2].

[1] points out that implementing an interface and supporting overridable methods (i.e. redefinition in derived classes) are logically distinct concepts, and therefore it is useful to separate these concerns in the programming language.

2 $interface directive

2.1 Simple example

An interface is an abstract type only containing method declarations and cannot be instantiated. The following declares an interface called `IShape`:

```
$interface IShape
{
  float64 GetArea() const;
  float64 GetPerim() const;
};
```

An interface pointer to an `IShape` is designated by `ptr<IShape>`. This wraps a pair of underlying pointers – a pointer to an object and a pointer to a vtable appropriate to the interface for that object. Given only the above interface definition it is possible to compile functions that can call the `IShape` methods on *any* given `ptr<IShape>`. For example:

```
void WriteArea(ptr<IShape> s)
{
  std::cout << "Area = " << s->GetArea()
            << std::endl;
}
```

In the literature this capability is generally referred to as dynamic polymorphism, because the one compiled implementation of `WriteArea()` is able to call the `GetArea()` method on *any* class that implements the `IShape` interface.

Now consider the following struct written in straight C++ (assuming an appropriate typedef for the `float64`):

```
struct Circle
{
  Circle() : r(3.0) {}
  float64 GetArea() const {return 3.14*r*r;}
  float64 GetPerim() const {return 2*3.14*r;}
  float64 r;
};
```

Notice that `Circle` doesn't inherit from any base classes. An instance of a `Circle` class is only 8 bytes (the size of its `float64` member `r`). There are no virtual methods so there is no vtable pointer. Despite that `Circle` is able to implement the `IShape` interface, simply because it has implemented all the `IShape` methods with the required signature and semantics.

The coercion into an interface pointer is very simple:

```
Circle c;
ptr<IShape> s = &c; // Coercion to IShape
WriteArea(s);
```

or just

```
Circle c;
WriteArea(&c);
```

If `Circle` hadn't implemented the methods in the `IShape` interface then the attempted coercion would have resulted in a *compile time* error.

Given one or more coercions from `Circle` to `IShape` within a given executable or DLL, the compiler generates a single static vtable of pointers to functions corresponding to the declared methods in the `IShape` interface. Each function takes a pointer to a `Circle` object as the first argument (i.e. in addition to the declared formal arguments). The run time overheads of coercion are minimal - only requiring initialisation of the two pointers within the `ptr<IShape>`: the address of the circle object and the vtable pointer. This may only require two `mov` machine code instructions.

The remarkable idea is that a polymorphic capability can be post-applied to an object independently of its implementation.

2.2 Reason for `ptr<>` construct

In [2] an interface like `IShape` is a value type that represents a reference to some object that implements the interface. The type `const IShape` applies constness to the reference instead of the object being referenced. Unfortunately there is no elegant way

to apply constness to the object. In [2], `const_view<IShape>` is used to represent a reference type to a `const IShape`.

This problem disappears when using the `ptr<>` syntax, because `ptr<const IShape>` associates constness with the object rather than the pointer.

2.3 Advantages

Many of the advantages of interfaces listed below are described in [1].

2.3.1 Space efficiency

Avoiding the vtable pointer in an object is particularly beneficial for applications that record large numbers of small objects in memory. The overhead of dynamic polymorphism is only paid when needed. For example, at a given point in time there could be millions of `Circle` objects in memory, and only a few are being accessed by clients through an abstract interface.

2.3.2 Compiler optimisation

Class methods are not declared virtual and therefore it is more likely that the C/C++ compiler will optimise implementations because it doesn't commit so quickly to the indirection associated with a vtable look-up.

For example, in the following function the compiler has no need to generate a virtual call to Circle::GetArea(). This also allows it to in-line the method call - one of the most effective optimisations available to a compiler.

```
float64 GetConeVolume(const Circle& c, float64
    height)
{
  return c.GetArea()*height/3;
}
```

If `Circle::GetArea()` is virtual then the compiler must perform further analysis to determine whether the method has been overridden in a derived class. Since it is always possible that a derived class is defined in a different DLL which is dynamically loaded at run time, it is generally impossible for the compiler to avoid the overhead of the virtual call without top-down type analysis.

2.3.3 Efficient multiple inheritance

Interfaces support Multiple Inheritance (MI) very effectively and efficiently. If required, interfaces can form complex MI hierarchies. Also a class can support hundreds of interfaces without any negative impact on performance. Object instances never contain any vtable pointers and vtable pointers are only defined and initialised at the point where an interface pointer is required.

By contrast, using ABCs with complex inheritance hierarchies leads to vtable pointer bloat in most C++ compilers. An example is given in section 2.6

2.3.4 Object creation efficiency

Object creation is faster because there are no vtable pointers to initialise. This has been found significant for classes using many virtual base classes with the Microsoft Visual C++ compiler.

2.3.5 Forces a clean separation of interface and implementation

Interfaces always specify a set of abstract method definitions, and never allow for implementation inheritance (either of member variables or implementation of member functions). Programmer that use ABCs have to be conscientious to avoid mixing interface and implementation inheritance.

2.3.6 Not type intrusive

An object can be made to implement an interface regardless of whether its methods are declared virtual and regardless of what bases classes it derives from. One or more of the methods could even be static!

2.3.7 Objects can implement the same interface in different ways

Objects can be made to implement the same interface in different ways for different purposes. An example is given in section 2.4 where class `CDPlayer` is able to implement interface `IButtonListener` in two distinct ways in order to distinguish the notifications it receives from its play and stop buttons.

2.3.8 Avoidance of heap allocations

As a corollary of 2.3.7, it is shown in section 2.4 that interfaces allow for objects to be interconnected or "wired up"' without the wiring itself being heap allocated.

2.4 CD Player example

Xcpp interfaces provide the means to "wire up" software components far more elegantly and efficiently than with conventional C++ or with languages like C# and Java. The normal approach is to heap allocate the wiring. For example, Java programs often use anonymous inner classes while C# programs often use delegates for this purpose. The following example shows that heap allocations are not necessary. In a complex system this could lead to significant performance gains.

This example shows how a CD Player can internally wire its play and stop buttons to its `Play()` and `Stop()` methods. Consider that the public header file Button.h defines an interface `IButtonListener`. A client is expected to implement this interface in order to receive a notification that a button has been pressed.

```
$interface IButtonListener
{
  void OnButtonPressed();
};
```

Let Button.h also contain the following definition of class `Button`:

```
class Button
{
public:
  void SetHandler(ptr<IButtonListener> h)
  {
    m_handler = h;
  }
  void OnMouseClicked()
  {
    m_handler->OnButtonPressed();
  }
private:
  ptr<IButtonListener> m_handler;
};
```

A client calls `SetHandler()` in order to set the handler that receives the notification message that the button has been pressed.

It is assumed the GUI framework calls `OnMouseClicked()` when the user clicks the mouse button while the mouse cursor is over the button. The implementation of `OnMouseClicked()` sends a notification to the client that the button has been pressed.

Now consider the implementation of CDPlayer, as follows:-

```
@import "Button.h"

class CDPlayer
{
public:
  CDPlayer()
  {
    struct B1 : public CDPlayer
    {
      void OnButtonPressed() {Play(5);}
    };
    m_playButton.SetHandler((B1*)this);

    struct B2 : public CDPlayer
    {
      void OnButtonPressed() {Stop();}
    };
    m_stopButton.SetHandler((B2*)this);
  }

  void Play(int speed);
  void Stop();

private:
  Button m_playButton;
  Button m_stopButton;
};
```

Note how the CDPlayer has two Button member variables. These need to be "wired up" so they call the Play() and Stop() methods respectively.

The interesting idea is that we can write a struct called B1 that inherits from CDPlayer. This adds member functions but no member variables (i.e. no state) to the CDPlayer class. Therefore it is reasonable to reinterpret a CDPlayer as a B1 which can in turn be coerced into an IButtonListener because it implements OnButtonPressed() as required.

The upshot is that the CDPlayer can implement the IButtonListener interface for the purposes of receiving the OnButtonPressed() notification from its play button. Similarly, the CDPlayer can implement the IButtonListener interface again (and differently) for its stop button.

This technique is unconventional but very powerful and efficient. Unlike Java anonymous inner classes or C# delegates (for example) no heap allocation is required at all to allow a button to send a message to its CDPlayer.

2.5 Interface inheritance

Interfaces can inherit from each other. Multiple inheritance is fully supported. Only DAG structures are permitted. For example

```
$interface IShape
{
  float64 GetArea() const;
  float64 GetPerim() const;
};
$interface IColouredShape : IShape
{
  int32 GetColour() const;
};
$interface ICircle : IShape
{
  float64 GetRadius() const;
};
$interface IColouredCircle :
  IColouredShape, ICircle
{
};
```

Now consider the following struct in straight C++

```
struct ColouredCircle
{
  ColouredCircle() : r(3.0), c(17) {}
  float64 GetArea() const {return 3.14*r*r;}
  float64 GetPerim() const {return 2*3.14*r;}
  float64 GetRadius() const {return r;}
  int32 GetColour() const {return c;}

  float64 r;
  int32 c;
};
```

This can be coerced into an IColouredCircle as follows

```
ColouredCircle c;
ptr<IColouredCircle> p = &c;
```

The pointer p can be implicit upcast to a ptr<IShape>, ptr<IColouredShape> or ptr<ICircle>.

Let a *subinterface* refer to either a directly or indirectly inherited interface. For example, the subinterfaces of IColouredCircle are IShape, IColouredShape, ICircle.

An interface inherits all the methods from its subinterfaces. A pointer to an interface can be implicit upcast to a pointer to any subinterface.

Sideways or downwards casting of interface pointers requires a qicast<> but is only supported for interfaces that inherit from IObject (this is described below).

Repeated inheritance is supported, but only indirectly. The following is not permitted

```
$interface Ix {};
$interface Iy : Ix, Ix {}   // compiler error
```

2.6 Coloured circle using ABCs

For the purpose of comparison, consider the following code using ABCs, where it has been assumed that virtual inheritance should always be used for interface inheritance:

```
struct IShape
{
  virtual float64 GetArea() const = 0;
  virtual float64 GetPerim() const = 0;
};
struct IColouredShape : public virtual IShape
{
  virtual int32 GetColour() const = 0;
};
struct ICircle : public virtual IShape
{
  virtual float64 GetRadius() const = 0;
};
struct IColouredCircle :
    public virtual IColouredShape,
    public virtual ICircle
{
};
struct ColouredCircle : public IColouredCircle
{
  ColouredCircle() : r(3.0), c(17) {}
  float64 GetArea() const {return 3.14*r*r;}
  float64 GetPerim() const {return 2*3.14*r;}
  float64 GetRadius() const {return r;}
  int32 GetColour() const {return c;}

  float64 r;
  int32 c;
};
```

For the Microsoft Visual C++ compiler (VC 2008), the size of ColouredCircle is 56 bytes. That is an enormous overhead considering the member variables only take up 12 bytes.

2.7 The IObject interface

There is a special interface called IObject defined as follows

```
$interface IObject
{
  AnyInterface QueryInterface(
    const ReflectedInterface& ri);
  const ReflectedClass&
    GetReflectedClass() const;
  void VisitObjects(IObjectVisitor& v) const;
  void OnGarbageCollect();
  void Destroy();
  ObjSysState& GetSysState();
};
```

Xcpp allows for the keyword `isa` to be used to specify one or more interfaces that are implemented by a given class/struct. Each of the interfaces must inherit directly or indirectly from `IObject`.

Xcpp will generate code to implement this interface implicitly. For example

```
$struct X isa IObject {};
```

will compile successfully even though the methods of `IObject` haven't been explicitly implemented (nor should they).

Note that global variables, frame variables or member variables of classes are allowed to implement `IObject`. i.e. it shouldn't be assumed that only heap allocated objects are allowed to implement `IObject`, even though some of the functionality is specific to support for the `CSpace` garbage collector.

2.8 QueryInterface

The method `QueryInterface()` is similar in purpose to the function of the same name in the Microsoft COM interface `IUnknown`. It allows a client to cast interface pointers.

Consider the following interface definitions

```
$interface Ix : IObject {};
$interface Iy : IObject {};
$interface Iz : Ix,Iy {};
$interface Iw : IObject {};
```

Now suppose struct X implements Iz as follows

```
$struct X isa Iz {};
```

then the following code shows how we may cast between the different interface pointers using `qicast<>`.

```
X x;
ptr<Ix> px = &x;

// qicast<> can be used to cast to any other
// supported interface
ptr<Iy> py = qicast<Iy>(px);

// Casting to an interface that is not
// implemented by the object returns null.
assert(qicast<Iw>(px) == null);
```

For a given object we could define a binary relation between interfaces according to whether `qicast` will successfully take us from one interface to another interface. It is guaranteed that this relation cannot change while a process is running, and the relation will be reflexive, symmetric and transitive.

References

[1] Christopher Diggins, *C++ with Interfaces*, Dr. Dobb's, September 01,2004, `http://www.ddj.com/cpp/184401848`.

[2] Jonathan Turkanis, *Boost.Interfaces*, `http://www.coderage.com/interfaces/`