12.6 — Pure virtual functions, abstract base classes, and interface classes

BY ALEX ON FEBRUARY 13TH, 2008 | LAST MODIFIED BY ALEX ON FEBRUARY 13TH, 2008

Finally, we arrive at the end of our long journey through inheritance! This is the last topic we will cover on the subject. So congratulations in advance on making it through the hardest part of the language!

Pure virtual (abstract) functions and abstract base classes

So far, all of the virtual functions we have written have a body (a definition). However, C++ allows you to create a special kind of virtual function called a **pure virtual function** (or **abstract function**) that has no body at all! A pure virtual function simply acts as a placeholder that is meant to be redefined by derived classes.

To create a pure virtual function, rather than define a body for the function, we simply assign the function the value 0.

```
class Base
{
public:
    const char* SayHi() { return "Hi"; } // a normal non-virtual function

    virtual const char* GetName() { return "Base"; } // a normal virtual function

    virtual int GetValue() = 0; // a pure virtual function
};
```

When we add a pure virtual function to our class, we are effectively saying, “it is up to the derived classes to implement this function”.

Using a pure virtual function has two main consequences: First, any class with one or more pure virtual functions becomes an **abstract base class**, which means that it can not be instantiated! Consider what would happen if we could create an instance of Base:

```
int main()
{
    Base cBase; // pretend this was legal
    cBase.GetValue(); // what would this do?
}
```

Second, any derived class must define a body for this function, or that derived class will be considered an abstract base class as well.

Let’s take a look at an example of a pure virtual function in action. In a previous lesson, we wrote a simple Animal base class and derived a Cat and a Dog class from it. Here’s the code as we left it:

```
#include <string>
class Animal
{
protected:
    std::string m_strName;

    // We're making this constructor protected because
    // we don't want people creating Animal objects directly,
    // but we still want derived classes to be able to use it.
    Animal(std::string strName)
        : m_strName(strName)
    {
    }

public:
    std::string GetName() { return m_strName; }
```

```
    virtual const char* Speak() { return "???"; }
};

class Cat: public Animal
{
public:
    Cat(std::string strName)
        : Animal(strName)
    {
    }

    virtual const char* Speak() { return "Meow"; }
};

class Dog: public Animal
{
public:
    Dog(std::string strName)
        : Animal(strName)
    {
    }

    virtual const char* Speak() { return "Woof"; }
};
```

We've prevented people from allocating objects of type Animal by making the constructor protected. However, there's one problem that has not been addressed. It is still possible to create derived classes that do not redefine Speak(). For example:

```
class Cow: public Animal
{
public:
    Cow(std::string strName)
        : Animal(strName)
    {
    }

    // We forgot to redefine Speak
};

int main()
{
    Cow cCow("Betsy");
    cout << cCow.GetName() << " says " << cCow.Speak() << endl;
}
```

This will print:

```
Betsy says ???
```

What happened? We forgot to redefine Speak, so cCow.Speak() resolved to Animal.Speak(), which isn't what we wanted.

A better solution to this problem is to use a pure virtual function:

```
#include <string>
class Animal
{
protected:
    std::string m_strName;

public:
    Animal(std::string strName)
        : m_strName(strName)
    {
    }

```

```
    std::string GetName() { return m_strName; }
    virtual const char* Speak() = 0; // pure virtual function
};
```

There are a couple of things to note here. First, Speak() is now a pure virtual function. This means Animal is an abstract base class, and can not be instantiated. Consequently, we do not need to make the constructor protected any longer (though it doesn't hurt). Second, because our Cow class was derived from Animal, but we did not define Cow::Speak(), Cow is also an abstract base class. Now when we try to compile this code:

```
class Cow: public Animal
{
public:
    Cow(std::string strName)
        : Animal(strName)
    {
    }

    // We forgot to redefine Speak
};

int main()
{
    Cow cCow("Betsy");
    cout << cCow.GetName() << " says " << cCow.Speak() << endl;
}
```

The compiler will give us a warning because Cow is an abstract base class and we can not create instances of abstract base classes:

```
C:\Test.cpp(141) : error C2259: 'Cow' : cannot instantiate abstract class due to following
        C:Test.cpp(128) : see declaration of 'Cow'
C:\Test.cpp(141) : warning C4259: 'const char *__thiscall Animal::Speak(void)' : pure virt
```

This tells us that we will only be able to instantiate Cow if Cow provides a body for Speak().

Let's go ahead and do that:

```
class Cow: public Animal
{
public:
    Cow(std::string strName)
        : Animal(strName)
    {
    }

    virtual const char* Speak() { return "Moo"; }
};

int main()
{
    Cow cCow("Betsy");
    cout << cCow.GetName() << " says " << cCow.Speak() << endl;
}
```

Now this program will compile and print:

```
Betsy says Moo
```

A pure virtual function is useful when we have a function that we want to put in the base class, but only the derived classes know what it should return. A pure virtual function makes it so the base class can not be instantiated, and the derived classes are forced to define these function before they can be instantiated. This helps ensure the derived classes do not

forget to redefine functions that the base class was expecting them to.

Interface classes

An **interface class** is a class that has no members variables, and where all of the functions are pure virtual! In other words, the class is purely a definition, and has no actual implementation. Interfaces are useful when you want to define the functionality that derived classes must implement, but leave the details of how the derived class implements that functionality entirely up to the derived class.

Interface classes are often named beginning with an I. Here's a sample interface class:

```
class IErrorLog
{
    virtual bool OpenLog(const char *strFilename) = 0;
    virtual bool CloseLog() = 0;

    virtual bool WriteError(const char *strErrorMessage) = 0;
};
```

Any class inheriting from IErrorLog must provide implementations for all three functions in order to be instantiated. You could derive a class named FileErrorLog, where OpenLog() opens a file on disk, CloseLog() closes it, and WriteError() writes the message to the file. You could derive another class called ScreenErrorLog, where OpenLog() and CloseLog() do nothing, and WriteError() prints the message in a pop-up message box on the screen.

Now, let's say you need to write some code that uses an error log. If you write your code so it includes FileErrorLog or ScreenErrorLog directly, then you're effectively stuck using that kind of error log. For example, the following function effectively forces callers of MySqrt() to use a FileErrorLog, which may or may not be what they want.

```
double MySqrt(double dValue, FileErrorLog &cLog)
{
    if (dValue < 0.0)
    {
        cLog.WriteError("Tried to take square root of value less than 0");
        return 0.0;
    }
    else
        return dValue;
}
```

A much better way to implement this function is to use IErrorLog instead:

```
double MySqrt(double dValue, IErrorLog &cLog)
{
    if (dValue < 0.0)
    {
        cLog.WriteError("Tried to take square root of value less than 0");
        return 0.0;
    }
    else
        return dValue;
}
```

Now the caller can pass in any class that conforms to the IErrorLog interface. If they want the error to go to a file, they can pass in an instance of FileErrorLog. If they want it to go to the screen, they can pass in an instance of ScreenErrorLog. Or if they want to do something you haven't even thought of, such as sending an email to someone when there's an error, they can derive a new class from IErrorLog (eg. EmailErrorLog) and use an instance of that! By using IErrorLog, your function becomes more independent and flexible.

Interface classes have become extremely popular because they are easy to use, easy to extend, and easy to maintain. In fact, some modern languages, such as Java and C#, have added an "interface" keyword that allows programmers to directly define an interface class without having to explicitly mark all of the member functions as abstract. Furthermore, although Java and C# will not let you use multiple inheritance on normal classes, they will let you multiply inherit as many interfaces as you like. Because interfaces have no data and no function bodies, they avoid a lot of the traditional problems with multiple

inheritance while still providing much of the flexibility.

13.1 -- Input and output (I/O) streams

Index

12.5 -- The virtual table

Share this:

C++ TUTORIAL | PRINT THIS POST

35 comments to 12.6 — Pure virtual functions, abstract base classes, and interface classes

Susan
April 14, 2008 at 12:58 am · Reply

Can you please answer why a pure virtual function should be equated to 0 and not to any other number.

Alex
April 15, 2008 at 10:18 am · Reply

As far as I know, =0 is just a syntactic convention for letting the compiler know that this is a pure virtual function.

You can think about it this way: normal functions have a body of code that needs to be executed when they are called. This code has to live somewhere in memory, so the function's name is essentially a pointer to that code.

On the other hand, pure virtual functions have no body, and thus, do not have need for an address at all. 0 is the NULL address.