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)
    {
    }
```

```
12
13      std::string GetName() { return m_strName; }
14      virtual const char* Speak() = 0; // pure virtual function
15  };
```

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:

```
1   class Cow: public Animal
2   {
3   public:
4       Cow(std::string strName)
5           : Animal(strName)
6       {
7       }
8
9       // We forgot to redefine Speak
10  };
11
12  int main()
13  {
14      Cow cCow("Betsy");
15      cout << cCow.GetName() << " says " << cCow.Speak() << endl;
16  }
```

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:

```
1   class Cow: public Animal
2   {
3   public:
4       Cow(std::string strName)
5           : Animal(strName)
6       {
7       }
8
9       virtual const char* Speak() { return "Moo"; }
10  };
11
12  int main()
13  {
14      Cow cCow("Betsy");
15      cout << cCow.GetName() << " says " << cCow.Speak() << endl;
16  }
```

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

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.

Jim
December 1, 2008 at 4:33 am · Reply

I read this today, they didn't want to add any more reserved words such as "pure" and it was moving too far from the C language, so this hotchpotch =0 stuff was used instead

w0rkbean
December 24, 2008 at 12:56 am · Reply

As someone said earlier, it's a syntactic convention. I think the reason for that convetion can be explained as " as we know the function name itself is a pointer to that function just like array name is a pointer to the first element of that array. So if you are equating that pointer to 0 which is a NULL, makes it a NULL pointer". So the essence is equating the function to 0 is like defing a NULL ponter which may point to a a valid address later on"

Sean
July 30, 2008 at 4:15 am · Reply

Great tutorial for interface class,
Thanks a lot!

Ben
August 7, 2008 at 10:55 am · Reply

Hi Alex,
in your IErrorLog class all functions are private. As i tried to compile your code, with the MySqrt() function, i discoverd, that the compiler complains about that, so i defined a class inheriting IErrorLong and changing the scope of the functions to public; still the compiler insists not to compile the code. So you might want to change the code. Besides that, can you imagine why a virtual class should ever have private member functions?
Ben

Grant
September 3, 2008 at 4:39 am · Reply

Hi Ben,

I agree with you, the functions need to be public. My code did compile - the only thing
I needed to be sure about was that the IErrorLog parameter to MySqrt was a pointer or reference.
If it's passed by value the compiler tries to instantiate an IErrorLog object which it can't do for an abstract class.

Grant

Tunc
September 10, 2009 at 12:51 am · Reply

why compile give me an error "IErrorLog cannot access private member declared in class "IErrorLog"

Balaji
September 19, 2008 at 10:35 pm · Reply

Very good tutorial for beginers,

Eugene Wee
February 13, 2009 at 9:38 am · Reply

A pure virtual function can be defined (as in have a body). One case where that is useful is when the base class implementer wishes to allow derived classes access to some default implementation, but require that concrete derived classes override that member function nonetheless.

[Thanks, I wasn't aware of that. -Alex]

tvraju
February 20, 2009 at 10:59 am · Reply

"First, any class with one or more virtual functions becomes an abstract base class,",
the text should be one or more "pure virtual functions" instead of virtual functions.

[Thanks! It's fixed now. -Alex]

E-man_96@live.com
April 3, 2009 at 6:50 pm · Reply

I have read 2 other tutorials and this one is the one that has "everything".

sudheer
June 10, 2009 at 5:19 am · Reply

Wonderful tutorial on Pure Virtual functions. Kudos Alex

Garrett
December 11, 2009 at 8:32 am · Reply

Quick nitpick on the HTML in this page - the top of the page with next/previous post lists 7.13 as the next topic instead of 13.1.

P.S.: Excellent website for learning C++. Thank you very much for providing this resource. I'll be sure to refer people here when appropriate!

LJ
February 4, 2010 at 12:02 pm · Reply

this is definitely the best tutorial i've found so far on c++. great work!
another quick nitpick: the mySqrt function does not take the square root 🙂

sachin kumar
July 29, 2010 at 8:41 am · Reply

verry good website

ricky
August 26, 2010 at 11:25 pm · Reply

just a simple program..

```
#include "stdafx.h"
#include "iostream"
#include "conio.h"
using namespace std;

class base
{
    public :
    int x;
    virtual void display();
};
void base::display()
{
    x =100;
    cout<<x<<" Display Base"<<endl;
}
class derived : public base
{
    void display();
};
void derived::display()
{
    x =50;cout<<x<<" Display Derived";
}
int _tmain(int argc, _TCHAR* argv[])
{
    base B1;
    derived d1;
    base *b1ptr;
    b1ptr = &B1;
    b1ptr->display();

    b1ptr = &d1;

    b1ptr->display();
    //d1.base::display();
    getch();
    return 0;
}
```

)

my doubt:

How am I able to access the private member of derived class function (**which is private) using base class pointer??

Kathy
November 13, 2010 at 8:54 pm · Reply

petrfect tutorial~~

I have learned a lot…

gans
February 1, 2011 at 3:46 am · Reply

Hi,
As reply to Ricky's question:

In C++ there is also a concept called “Late binding yet static typing”. That is
When you have a pointer to object and object is derived from the type of class of type pointer, there are two types,1) type of the pointer: static 2) type of pointed-to-object: dynamic.
Here static typing means, the legality of the function call is determined by the compiler at compile time, since pointer is Base and Base has public virtual function. compiler validates the function calling. but at run time, since you have assigned the derived object address, it resolves the derived function only.

Hope this helps u.

spartan
January 30, 2012 at 8:43 pm · Reply

Simply awesome

saurabh
February 5, 2012 at 4:16 pm · Reply

I learned a lot from your tutorials and kudos to you

It will be great if you can add two more topics to this chapter

1) We can define body for a pure virtual function so please add when, why and how ?
2) Please add virtual table view for abstract class why compiler not letting us to instantiate abstract class or i should say class having and pure virtual function

Here is my views:
we can not make object of abstract class because, in the vtable the vtable entry for the abstract class functions will be NULL, even if there is a single pure virtual function in the class the class becomes as abstract class.Even if there is a single NULL entry in the function table the compiler does not allow to create the object.

bantoo011
February 13, 2012 at 9:55 am · Reply

How is the vtable for abstract & interface class looks like ?

warzix
October 2, 2012 at 8:04 pm · Reply

Great article, although I should add one comment about the Interface class definition:
“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.”

Effectively, my comment is after the example below:

Suppose you define the following Interface

```
class ISomething {
public:
virtual void doStuff1() = 0;
//… bunch of other definitions
};
```

and you have a class that implements it:

```
class Impl : public ISomething {
private:
char* myCharPtr;
public:
```

```
void doStuff1() { /* do 'stuff' */ };
// implement all other stuff
/* Constructor that allocates memory */
Impl() { myCharPtr = new char[10]; }
/* Destructor that deallocates memory */
~Impl() { delete[] myCharPtr; }
};

void main()
{
ISomething* ptr = new Impl; //upcasting. ISomething pointer points to Impl object…

delete ptr; //cleaning up memory…
// This effectivelly invokes the default destructor on ISomething which is not virtual and therefore would not resolve to ~Impl()
// therefore this can lead to memory leaks (as shown in the example) or any other problems related to missing destructor calls.
}
```

I think it is worth a comment about that right after the phrase I've quoted such as:
"The only exception is the destructor which should be virtual and implemented if the Interface is supposed to be used as a "placeholder" for pointers to derived classes' objects…".

Best Regards

Mariya
January 1, 2015 at 5:48 am · Reply

How can we call a function with an argument of type reference to an object of an interface class if interface classes could not be instantiated? (I mean the example you give with the reference to IErrorLog.)

shani khan
February 17, 2015 at 9:23 am · Reply

for what purpose why we use abstract classes ????

Monic
June 17, 2015 at 7:57 am · Reply

Is it okay if we do not write the pure virtual function for let's say function draw in figure base class, but we still write the draw function in the derived class?
If so, then what is the advantage of writing the pure virtual function is base class if we can just write the function in every derived class?

Peter
August 20, 2015 at 7:53 am · Reply

An issue I noticed on your last example is you are defining a function called MySqrt but you are not doing anything with it, you are just returning the value you received.
I have put your example into a working program.

```
1  #include <string>
2  #include <cmath>
3  #include <iostream>
4
5  class IErrorLog
6  {
7  public:
```

```
8       virtual bool OpenLog(const char *strFilename) = 0;
9       virtual bool CloseLog() = 0;
10
11      virtual bool WriteError(const char *strErrorMessage) = 0;
12  };
13
14  class WriteErrorMessage: public IErrorLog
15  {
16  public:
17      virtual bool OpenLog(const char *strFilenname)
18      {
19      };
20
21      virtual bool CloseLog()
22      {
23      };
24
25      virtual bool WriteError(const char *strErrorMessage)
26      {
27          std::cout << "Error: " << strErrorMessage << std::endl;
28      }
29  };
30
31  double MySqrt(double dValue, IErrorLog &cLog)
32  {
33      if (dValue < 0.0)
34      {
35          cLog.WriteError("Tried to take square root of value less than 0");
36          return 0.0;
37      }
38      else
39          return sqrt(dValue);
40  }
41
42  int main()
43  {
44      double myNumber1(-1.0);
45      double myNumber2(9.0);
46      double myReturn(0.0);
47
48      WriteErrorMessage myMessage;
49      IErrorLog &myOutput = myMessage;
50
51      myReturn = MySqrt(myNumber1, myOutput);
52      if (myReturn != 0)
53          std::cout << "Square root of " << myNumber1 << " is " << myReturn << std::endl;
54
55      myReturn = MySqrt(myNumber2, myOutput);
56      if (myReturn != 0)
57          std::cout << "Square root of " << myNumber2 << " is " << myReturn << std::endl;
58
59      return 0;
60  }
```

vish
September 24, 2015 at 11:15 pm · Reply

how to remove ambuiguity here? or clear me if this is a bad coding
#include<iostream>
using namespace std;
class animal{
protected:
string mname;
public:

```
    animal(string a):mname(a){}
    const char* getspeak(){return "what";}
    };

class cow:virtual public animal{
    public:
    cow(string a):animal(a){}
    virtual const char* getspeak(){animal::getspeak();return  "moo";}


};
class cat:virtual public animal {
    public:
    cat(string a):animal(a){}
    virtual const char* getspeak(){return "meow";}

};
class pet:public cow,public cat{
    public:
    pet(string a,string b,string c):cow(b),cat(a),animal(c){}


};
int main(){
pet a("cow","cat","no");
cout<<a.getspeak();


}
```

Alex
September 27, 2015 at 2:45 pm · Reply

Your class hierarchy doesn't make sense. You're saying:
* A cow is an animal -- okay, that's fine.
* A cat is an animal -- okay, that's fine too.
* A pet is both a cow and a cat simultaneously -- that's not correct!

There are two ways to do what you want:
1) Have both Animal and Pet both be base classes. Then Cat can inherit from Animal and Pet, and Cow can inherit from Animal (probably not Pet, since Cows usually aren't kept as pets).

2) Since pet doesn't really do anything, it's probably better to make pet a virtual function: bool isPet(), that evaluates to true for Cat and false for Cow. That way you can avoid the multiple inheritance altogether.

Also, Animal::getspeak() should be made virtual.

Steiner
February 14, 2016 at 11:44 am · Reply

I'm just trying to give a technical answer on your question about removing ambiguity. Alex already gave an answer about redesigning this code.

To remove the ambiguity, you have to use the scope resolution operator and pick from which derived class do you want to invoke getspeak().

```
1 cout << a.cat::getspeak();
```

or

```
cout << a.cow::getspeak();
```

Devashish
October 1, 2015 at 9:27 pm · Reply

“any class with one or more pure virtual functions becomes an abstract base class, which means that it can not be instantiated”

If we can’t instantiate a base class that has a pure virtual function,

```
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;
}
```

isn’t cLog an instance of interface class IErrorLog???

Alex
October 3, 2015 at 9:50 am · Reply

Yes, this confusion is probably somewhat due to imprecise working on my part. You can’t _directly_ instantiate any class that has pure virtual functions (which includes interface classes). You _can_ indirectly instantiate that class by instantiating a derived class that provides definitions for all the virtual functions in the base class.

So yes, cLog is an instance of IErrorLog, but you can only pass objects of the derived class to it since there’s no way to instantiate an object that is just an IErrorLog.

Make sense?

Devashish
October 3, 2015 at 11:02 am · Reply

Yes…thanks 🙂

JaSoN
October 6, 2015 at 2:30 am · Reply

Dear Alex, I still not clear about the Interface class, can you make another example? Thank for your tutorial

Alex
October 6, 2015 at 4:06 pm · Reply

Interface classes are just classes where all functions are pure virtual (abstract). Any derived class must provide implementations for all of the pure virtual functions in the interface class.

```
class Shape {
  virtual int getSides() = 0;
```

```
    virtual void move(double x, double y, double z) = 0;
    virtual void rotate(double rotationDegrees) = 0;
};
```