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

ALEX AUGUST 30, 2021

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() const { return "Hi"; } // a normal non-virtual function

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

    virtual int getValue() const = 0; // a pure virtual function

    int doSomething() = 0; // Compile error: can not set non-virtual functions to 0
};

```

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 base; // We can't instantiate an abstract base class, but for the sake of example, pretend this was allowed
    base.getValue(); // what would this do?

    return 0;
}
```

Because there's no definition for getValue(), what would base.getValue() resolve to?

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

A pure virtual function example

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>
#include <utility>

class Animal
{
protected:
    std::string m_name;

    // 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(const std::string& name)
        : m_name{ name }
    {
    }

public:
    std::string getName() const { return m_name; }
    virtual const char* speak() const { return "???"; }

    virtual ~Animal() = default;
};

class Cat: public Animal
{
public:
    Cat(const std::string& name)
        : Animal{ name }
    {
    }

    const char* speak() const override { return "Meow"; }
};

class Dog: public Animal
{
public:
    Dog(const std::string& name)
        : Animal{ name }
    {
    }

    const char* speak() const override { return "Woof"; }
};
```

We've prevented people from allocating objects of type Animal by making the constructor protected. However, it is still possible to create derived classes that do not redefine function speak().

For example:

```cpp
#include <iostream>
#include <string>

class Cow : public Animal
{
public:
    Cow(const std::string& name)
        : Animal{ name }
    {
    }

    // We forgot to redefine speak
};

int main()
{
    Cow cow{"Betsy"};
    std::cout << cow.getName() << " says " << cow.speak() << '\n';

    return 0;
}
```

This will print:

```
Betsy says ???
```

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

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

```cpp
#include <string>

class Animal // This Animal is an abstract base class
{
protected:
    std::string m_name;

public:
    Animal(const std::string& name)
        : m_name{ name }
    {
    }

    const std::string& getName() const { return m_name; }
    virtual const char* speak() const = 0; // note that speak is now a pure virtual function

    virtual ~Animal() = default;
};
```

There are a couple of things to note here. First, speak() is now a pure virtual function. This means Animal is now 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:

```
#include <iostream>

class Cow: public Animal
{
public:
    Cow(const std::string& name)
        : Animal{ name }
    {
    }

    // We forgot to redefine speak
};

int main()
{
    Cow cow{ "Betsy" };
    std::cout << cow.getName() << " says " << cow.speak() <<
'\n';

    return 0;
}

```

The compiler will give us a warning because Cow is an abstract base class and we can not create instances of abstract base classes (Line numbers are wrong, because the Animal class was omitted from the above example):

```
<source>(33): error C2259: 'Cow': cannot instantiate abstract class
<source>(20): note: see declaration of 'Cow'
<source>(33): note: due to following members:
<source>(33): note: 'const char *Animal::speak(void) const': is abstract
<source>(15): note: see declaration of 'Animal::speak'
```

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:

```
#include <iostream>
#include <string>

class Cow: public Animal
{
public:
    Cow(const std::string& name)
        : Animal(name)
    {
    }

    const char* speak() const override { return "Moo"; }
};

int main()
{
    Cow cow{ "Betsy" };
    std::cout << cow.getName() << " says " << cow.speak() <<
'\n';

    return 0;
}
```

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 functions 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.

Pure virtual functions with bodies

It turns out that we can define pure virtual functions that have bodies:

```
#include <string>

class Animal // This Animal is an abstract base class
{
protected:
    std::string m_name;

public:
    Animal(const std::string& name)
        : m_name{ name }
    {
    }

    std::string getName() { return m_name; }
    virtual const char* speak() const = 0; // The = 0 means this function is pure virtual

    virtual ~Animal() = default;
};

const char* Animal::speak() const  // even though it has a body
{
    return "buzz";
}
```

In this case, speak() is still considered a pure virtual function because of the "=0" (even though it has been given a body) and Animal is still considered an abstract base class (and thus can't be instantiated). Any class that inherits from Animal needs to provide its own definition for speak() or it will also be considered an abstract base class.

When providing a body for a pure virtual function, the body must be provided separately (not inline).

For Visual Studio users

Visual Studio mistakenly allows pure virtual function declarations to be definitions, for example

```
// wrong!
virtual const char* speak() const
= 0
{
  return "buzz";
}
```

This is wrong and cannot be disabled.

This paradigm can be useful when you want your base class to provide a default implementation for a function, but still force any derived classes to provide their own implementation. However, if the derived class is happy with the default implementation provided by the base class, it can simply call the base class implementation directly. For example:

```cpp
#include <string>
#include <iostream>

class Animal // This Animal is an abstract base class
{
protected:
    std::string m_name;

public:
    Animal(const std::string& name)
        : m_name(name)
    {
    }

    const std::string& getName() const { return m_name; }
    virtual const char* speak() const = 0; // note that speak is a pure virtual function

    virtual ~Animal() = default;
};

const char* Animal::speak() const
{
    return "buzz"; // some default implementation
}

class Dragonfly: public Animal
{

public:
    Dragonfly(const std::string& name)
        : Animal{name}
    {
    }

    const char* speak() const override// this class is no longer abstract because we defined this function
    {
        return Animal::speak(); // use Animal's default implementation
    }
};

int main()
{
    Dragonfly dfly{"Sally"};
    std::cout << dfly.getName() << " says " << dfly.speak() << '\n';

    return 0;
}
```

The above code prints:

```
Sally says buzz
```

This capability isn't used very commonly.

Interface classes

An **interface class** is a class that has no member 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
{
public:
    virtual bool openLog(const char *filename) = 0;
    virtual bool closeLog() = 0;

    virtual bool writeError(const char *errorMessage) = 0;

    virtual ~IErrorLog() {} // make a virtual destructor in case we delete an IErrorLog pointer, so the proper derived destructor is called
};
```

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 the file, 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 (at least without recoding your program). For example, the following function effectively forces callers of mySqrt() to use a FileErrorLog, which may or may not be what they want.

```
#include <cmath> // for sqrt()

double mySqrt(double value, FileErrorLog &log)
{
    if (value < 0.0)
    {
        log.writeError("Tried to take square root of value less than 0");
        return 0.0;
    }
    else
    {
        return std::sqrt(value);
    }
}
```

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

```
#include <cmath> // for sqrt()
double mySqrt(double value, IErrorLog &log)
{
    if (value < 0.0)
    {
        log.writeError("Tried to take square root of value less than 0");
        return 0.0;
    }
    else
    {
        return std::sqrt(value);
    }
}
```

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 (e.g. EmailErrorLog) and use an instance of that! By using IErrorLog, your function becomes more independent and flexible.

Don’t forget to include a virtual destructor for your interface classes, so that the proper derived destructor will be called if a pointer to the interface is deleted.

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 (prior to version 8) and C# will not let you use multiple inheritance on normal classes, they will let you multiple 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.

Pure virtual functions and the virtual table

Abstract classes still have virtual tables, as these can still be used if you have a pointer or reference to the abstract class. The virtual table entry for a pure virtual function will generally either contain a null pointer, or point to a generic function that prints an error (sometimes this function is named __purecall) if no override is provided.


Next lesson
18.8 Virtual base classes


Back to table of contents


Previous lesson
18.6 The virtual table

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest