

## 25.7 — Pure virtual functions, abstract base classes, and interface classes

 [learncpp.com/cpp-tutorial/pure-virtual-functions-abstract-base-classes-and-interface-classes/](http://learncpp.com/cpp-tutorial/pure-virtual-functions-abstract-base-classes-and-interface-classes/)

### 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.

```
#include <string_view>

class Base
{
public:
    std::string_view sayHi() const { return "Hi"; } // a normal non-virtual function

    virtual std::string_view 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 <string_view>

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(std::string_view name)
        : m_name{ name }
    {
    }

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

    virtual ~Animal() = default;
};

class Cat: public Animal
{
public:
    Cat(std::string_view name)
        : Animal{ name }
    {
    }

    std::string_view speak() const override { return "Meow"; }
};

class Dog: public Animal
{
public:
    Dog(std::string_view name)
        : Animal{ name }
    {
    }

    std::string_view 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:

```

#include <iostream>
#include <string>
#include <string_view>

class Cow : public Animal
{
public:
    Cow(std::string_view 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:

```

#include <string>
#include <string_view>

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

public:
    Animal(std::string_view name)
        : m_name{ name }
    {
    }

    const std::string& getName() const { return m_name; }
    virtual std::string_view 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>
#include <string_view>

class Cow: public Animal
{
public:
    Cow(std::string_view 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 an error 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>
#include <string_view>

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

public:
    Animal(std::string_view name)
        : m_name{ name }
    {
    }

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

    virtual ~Animal() = default;
};

class Cow: public Animal
{
public:
    Cow(std::string_view name)
        : Animal(name)
    {
    }

    std::string_view 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.

Just like with normal virtual functions, pure virtual functions can be called using a reference (or pointer) to a base class:

```
int main()
{
    Cow cow{ "Betsy" };
    Animal& a{ cow };

    std::cout << a.speak(); // resolves to Cow::speak(), prints "Moo"

    return 0;
}
```

In the above example, `a.speak()` resolves to `Cow::speak()` via virtual function resolution.

A reminder

Any class with pure virtual functions should also have a virtual destructor.

### Pure virtual functions with definitions

It turns out that we can create pure virtual functions that have definitions:

```
#include <string>
#include <string_view>

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

public:
    Animal(std::string_view name)
        : m_name{ name }
    {
    }

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

    virtual ~Animal() = default;
};

std::string_view Animal::speak() const // even though it has a definition
{
    return "buzz";
}
```

In this case, `speak()` is still considered a pure virtual function because of the `= 0` (even though it has been given a definition) 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 definition for a pure virtual function, the definition must be provided separately (not inline).

For Visual Studio users

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

```
virtual std::string_view speak() const = 0
{
    return "buzz";
}
```

This is non-conforming with the C++ standard, 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:



```

#include <iostream>
#include <string>
#include <string_view>

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

public:
    Animal(std::string_view name)
        : m_name(name)
    {
    }

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

    virtual ~Animal() = default;
};

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

class Dragonfly: public Animal
{
public:
    Dragonfly(std::string_view name)
        : Animal{name}
    {
    }

    std::string_view 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.

A destructor can be made pure virtual, but must be given a definition so that it can be called when a derived object is destructed.

## Interface classes

An **interface class** is a class that has no member variables, and where *all* of the functions are pure virtual! 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:

```
#include <string_view>

class IErrorLog
{
public:
    virtual bool openLog(std::string_view filename) = 0;
    virtual bool closeLog() = 0;

    virtual bool writeError(std::string_view 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;
    }

    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;
    }

    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 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

For consistency, abstract classes still have virtual tables. A constructor or destructor of an abstract class can call a virtual function, and it needs to resolve to the proper function (in the same class, since the derived classes either haven't been constructed yet or have already been destroyed).

The virtual table entry for a class with 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`).