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

ALEX[1] OCTOBER 1, 2024

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:

```cpp
#include <iostream>
#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 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:

```cpp
#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>
#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 }
    {
    }

    // 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:

```
prog.cc:35:9: error: variable type 'Cow' is an abstract class
   35 |     Cow cow{ "Betsy" };
      |         ^
prog.cc:17:30: note: unimplemented pure virtual method 'speak' in 'Cow'
   17 |     virtual std::string_view speak() const = 0; // note that speak is now
      |                              ^
```

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:

```cpp
#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:

```cpp
#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.

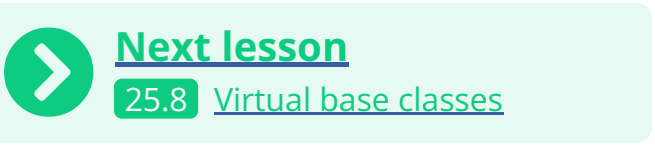
Don't forget to include a virtual destructor for your interface classes, so that the proper derived destructor will be called if a pointer to the interface is deleted.

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

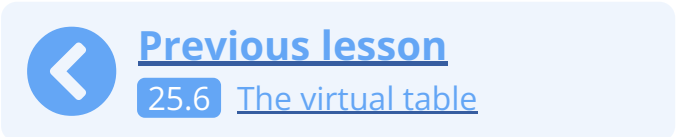
For consistency, abstract classes still have virtual tables. A constructor or destructor of an abstract class can call a virtual function, and it needs to resolve to the proper function (in the same class, since the derived classes either haven't been constructed yet or have already been destroyed).

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).



2

3



4

5

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[7] are connected to your provided email address.

230 COMMENTS

Newest

polymorphic

June 20, 2025 11:20 am PDT

Previously there were exercises that asked about the order of construction given a derived class instantiation. Example: D d{} would lead to A, B, C, then D being constructed and destruction is reversed. If A was abstract, for example, then would the construction skip A since it is abstract? Like B, C, D?

0 Reply

polymorphic

Reply to polymorphic[8] June 20, 2025 11:28 am PDT

Well I just tested and it still goes A, B, ... for construction and ..., B, A for destruction despite the base class A being abstract.

0 Reply

redshift

May 22, 2025 4:39 pm PDT

So, if a class has pure virtual functions with definitions, it can't be classified as an interface, is that right?

0 Reply

PollyWantsACracker

Reply to redshift[9] June 24, 2025 5:18 am PDT

i think it can becouse goal of interface is only to make sure you implement all methods but i might be

wrong

0 Reply

Nidhi Gupta
May 6, 2025 9:17 am PDT

A pure virtual function is a virtual function that has no implementation in the base class and is marked with = 0. It must be overridden by derived classes.

virtual ReturnType FunctionName() const = 0;
Though rare, a pure virtual function can have a definition—but only outside the class declaration.

virtual std::string_view speak() const = 0;

std::string_view Animal::speak() const { return "buzz"; }
Still makes the class abstract.

Derived classes can call this default using Base::speak().

0 Reply

Robert Reimann
March 31, 2025 7:32 am PDT

> "Any class with pure virtual functions should also have a virtual destructor."
Why? I find that this is not explained. Or maybe virtual destructors are explained somewhere else in an upcoming lesson?

As far as I understand, the virtual keyword is used for member functions whenever they are supposed to be overriden. What happens with virtual destrucotrs though? Because constrcutors are not to be made virtual.

1 Reply

Block3r
Reply to Robert Reimann [10] May 4, 2025 6:02 am PDT

In one of the previous lessons (https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/)[11] there is an information that if you delete a derived object through a pointer or reference to the base class, if the destructor is not virtual, the program will not call the derived class destructor. It will just go into the base class destructor, see that it is not virtual - so it will think that "hmm, the base class destructor is not virtual, so I do not need to search the virtual table to resolve better destructor to call". Therefore only the base portion of the derived class object will be destroyed, effectively causing a memory leak if the derived class object managed a dynamically allocated resource.

1 Reply

Seb

January 18, 2025 8:44 am PST

Since interface classes can't contain variables, is there any way to enforce that a class derived from an interface class contain certain member variables (in the same way an interface enforces the derived class having certain functions)?

0 Reply

Davi Bergamin

Reply to Seb [12] April 4, 2025 1:40 pm PDT

There is actually nothing stopping you from putting variables in the class, this rule is more of convention. The C++ doesnt have formal interfaces like other languages, in practice they are classes that we decided to use as interfaces. If you use a language like java that does have proper interfaces you could declare a getter in the interface like `virutal int getValue()`. Now, it is debateble if any of this is a good practice.

0 Reply

EmtyC

December 29, 2024 1:40 pm PST

Oh man, interface classes and polymorphic inheritance just opened a wide range of possibilities (mostly possibilities to reduce dev time)

2 Reply

Tvishy

April 26, 2024 5:32 am PDT

Hi Alex,
From the example above.
Say I got the ScreenError class derived from IErrorLog and I don't want to fiddle with implemening opening and closing file virtual functions, but just interested in writing to the console. Is it commonly allowed to delete those virtual functions in the derived class in such a way like bool closeLog() delete?
Doing so triggers the compiler to hint me with message "cannot override non-deleted function"

0 Reply

Alex Author

Reply to Tvishy [13] April 26, 2024 3:36 pm PDT

A deleted function can't override a non-deleted function, nor vice-versa.

0 Reply

abdel
April 12, 2024 5:32 am PDT

Hi Alex,
I have a couple of questions:
1/ Since we can provide a "dummy" definition for pure virtual functions, can we use this functionality for documentation purposes by maybe giving an example to developers of how the function should be implemented in derived classes?
2/ "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". Is it the same for a pure virtual destructor of a base class, since the destructor is required to have a body for derived objects to be destructed ?
3/ Out of curiosity, "Interface"class is an "Abstract" class. Which means, conceptually, "Interface" is derived from "Abstract". Does the C++ language uses a real inheritance to implement these two concepts in the language?

thanks in advance,

0 Reply

Alex Author
Reply to abdel [14] April 14, 2024 4:21 pm PDT

1. You can do whatever you want. Generally though the body of a pure virtual function should be code that you expect to be called explicitly from a derived class.
2. I'm not sure, since this is an implementation detail, but I assume so.
3. No. The terms "interface" and "abstract" are just words we use to describe classes that have a certain set of properties. The language does not "implement" these things, but rather provides tools for us to implement classes that we would then label with these terms.

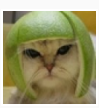
3 Reply

BLANK28
January 22, 2024 3:11 pm PST

Hey Alex,
"Abstract classes still have virtual tables, as these can still be used if you have a pointer or reference to the abstract class."

There wouldn't be any objects of an abstract class right so why do we need a virtual function table for this class?

0 Reply



Alex Author
Reply to BLANK28 [15] January 23, 2024 10:28 am PST

A few reasons that I can think of:

1. It's more consistent to do so.
2. Constructors and destructors can call virtual functions.

3. I think dynamic_cast is typically implemented via the virtual table.

1 Reply

Dck

Reply to Alex [16] April 6, 2024 10:37 pm PDT

Hi everyone, just following up on this thread, I'm confused about how `these can still be used if you have a pointer or reference to the abstract class.` since if there is a pointer or reference to the `Base` class then the `*__vptr` has to point to some derived class' virtual table, so there is no need for a virtual table for the `Base` class. Please let me know if I'm wrong!

0 Reply

Alex Author

Reply to Dck [17] April 8, 2024 4:07 pm PDT

I rewrote that sentence in the lesson to be more reflective of the answer in the prior comment.

0 Reply

...

October 10, 2023 2:58 pm PDT

I have a question about

```
#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);
    }
}
```

is it possible to pass any type that inherit from IErrorLog interface because any object derived from the class have a base portion of it is that right ?

its like the assignment of base{derived} derived upcast to base

sry if my question not clear.

Last edited 1 year ago by ...

0 Reply

Alex Author

Reply to ... [18] October 13, 2023 12:58 pm PDT

Yes, you can pass any class derived from IErrorLog (because it has an IErrorLog portion) and virtual function resolution will resolve any function calls to the correct derived function.

0 Reply

...

Reply to Alex [19] October 13, 2023 4:27 pm PDT

thx :)

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/virtual-base-classes/
3. https://www.learncpp.com/
4. https://www.learncpp.com/cpp-tutorial/the-virtual-table/
5. https://www.learncpp.com/pure-virtual-functions-abstract-base-classes-and-interface-classes/
6. https://www.learncpp.com/cpp-tutorial/command-line-arguments/
7. https://gravatar.com/
8. https://www.learncpp.com/cpp-tutorial/pure-virtual-functions-abstract-base-classes-and-interface-classes/#comment-611054
9. https://www.learncpp.com/cpp-tutorial/pure-virtual-functions-abstract-base-classes-and-interface-classes/#comment-610319
10. https://www.learncpp.com/cpp-tutorial/pure-virtual-functions-abstract-base-classes-and-interface-classes/#comment-608931
11. https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/
12. https://www.learncpp.com/cpp-tutorial/pure-virtual-functions-abstract-base-classes-and-interface-classes/#comment-606721
13. https://www.learncpp.com/cpp-tutorial/pure-virtual-functions-abstract-base-classes-and-interface-classes/#comment-596223
14. https://www.learncpp.com/cpp-tutorial/pure-virtual-functions-abstract-base-classes-and-interface-classes/#comment-595683
15. https://www.learncpp.com/cpp-tutorial/pure-virtual-functions-abstract-base-classes-and-interface-classes/#comment-592694
16. https://www.learncpp.com/cpp-tutorial/pure-virtual-functions-abstract-base-classes-and-interface-classes/#comment-592747
17. https://www.learncpp.com/cpp-tutorial/pure-virtual-functions-abstract-base-classes-and-interface-classes/#comment-595498

18. https://www.learncpp.com/cpp-tutorial/pure-virtual-functions-abstract-base-classes-and-interface-classes/#comment-588489
19. https://www.learncpp.com/cpp-tutorial/pure-virtual-functions-abstract-base-classes-and-interface-classes/#comment-588769
20. https://g.ezoic.net/privacy/learncpp.com