Why do we need a pure virtual destructor in C++?

Asked 12 years, 2 months ago Active 2 months ago Viewed 96k times

164

65

I understand the need for a virtual destructor. But why do we need a ***pure*** virtual destructor? In one of the C++ articles, the author has mentioned that we use pure virtual destructor when we want to make a class abstract.

But we can make a class abstract by making any of the member functions as pure virtual.

So my questions are

1. When do we really make a destructor pure virtual? Can anybody give a good real time example?
2. When we are creating abstract classes is it a good practice to make the destructor also pure virtual? If yes..then why?

c++ destructor pure-virtual

Share Edit Follow

edited Jul 21 at 1:03
allyourcode
20k ● 17 ● 73 ● 102

asked Aug 2 '09 at 19:27
Mark
1,765 ● 3 ● 12 ● 7

4 Multiple duplicates: stackoverflow.com/questions/999340/… and stackoverflow.com/questions/630950/pure-virtual-destructor-in-c being two of them – Daniel Sloof Aug 2 '09 at 19:29

17 @Daniel- The mentioned links doesn't answer my question. It answers why a pure virtual destructor should have a definition. My question is why we need a pure virtual destructor. – Mark Aug 2 '09 at 19:35

I was trying to find out the reason, but you already asked the question here. – nsivakr Aug 25 '10 at 0:00

11 Answers

Active Oldest Votes

127

1. Probably the real reason that pure virtual destructors are allowed is that to prohibit them would mean adding another rule to the language and there's no

need for this rule since no ill-effects can come from allowing a pure virtual destructor.

2. Nope, plain old virtual is enough.

If you create an object with default implementations for its virtual methods and want to make it abstract without forcing anyone to override any **specific** method, you can make the destructor pure virtual. I don't see much point in it but it's possible.

Note that since the compiler will generate an implicit destructor for derived classes, if the class's author does not do so, any derived classes will **not** be abstract. Therefore having the pure virtual destructor in the base class will not make any difference for the derived classes. It will only make the base class abstract (thanks for @kappa's comment).

One may also assume that every deriving class would probably need to have specific clean-up code and use the pure virtual destructor as a reminder to write one but this seems contrived (and unenforced).

Note: The destructor is the only method that even if it *is* pure virtual **has** to have an implementation in order to instantiate derived classes (yes pure virtual functions can have implementations).

```
struct foo {
    virtual void bar() = 0;
};

void foo::bar() { /* default implementation */ }

class foof : public foo {
    void bar() { foo::bar(); } // have to explicitly call default
implementation.
};
```

Share Edit Follow

edited May 23 '17 at 11:47
Community Bot
1 ● 1

answered Aug 2 '09 at 19:30
Motti
101k ● 44 ● 181 ● 253

14 "yes pure virtual functions can have implementations" Then it's not pure virtual. – GManNickG Aug 2 '09 at 19:37

2 If you want to make a class abstract, wouldn't it be simpler to just make all constructors protected? – bdonlan Aug 2 '09 at 19:41

80 @GMan, you're mistaken, being pure virtual means derived classes must override this method, this is orthogonal to having an implementation. Check out my code and comment out `foof::bar` if you want to see for yourself. – Motti Aug 2 '09 at 19:52

15 @GMan: the C++ FAQ lite says "Note that it is possible to provide a definition for a pure virtual function, but this usually confuses novices and is best avoided until later."

parashift.com/c++-faq-lite/abcs.html#faq-22.4 Wikipedia (that bastion of correctness) also says likewise. I believe the ISO/IEC standard uses similar terminology (unfortunately my copy is at work at the moment)... I agree that it's confusing, and I generally don't use the term without clarification when I'm providing a definition, especially around newer programmers... – leander Aug 2 '09 at 20:01

10 @Motti: What is interesting here and provides more confusion is that pure virtual destructor does NOT need to be explicitely overriden in derived (and instantiated) class. In such a case the implicit definition is used :) – kappa Aug 27 '14 at 22:50

33

All you need for an abstract class is at least one pure virtual function. Any function will do; but as it happens, the destructor is something that *any* class will have—so it's always there as a candidate. Furthermore, making the destructor pure virtual (as opposed to just virtual) has no behavioral side effects other than to make the class abstract. As such, a lot of style guides recommend that the pure virtual destuctor be used consistently to indicate that a class is abstract—if for no other reason than it provides a consistent place someone reading the code can look to see if the class is abstract.

Share Edit Follow

answered Aug 2 '09 at 22:03

Braden
1,412 ● 10 ● 11

2 but still why to provide the implementation of the pure virtaul destructor. What could possibly go wrong it I make a destructor pure virtual and doesn't provide its implementation. I assume only base classes pointers are declared and hence the destructor for abstract class is never called. – Krishna Oza Mar 10 '14 at 5:52

5 @Surfing: because a destructor of a derived class implicitly calls the destructor of its base class, even if that destructor is pure virtual. So if there is no implementation for it undefined bahavior is going to happen. – a.peganz Sep 2 '14 at 8:04

21

If you want to create an abstract base class:

- that **can't be instantiated** (yep, this is redundant with the term "abstract"!)
- but **needs virtual destructor behavior** (you intend to carry around pointers to the ABC rather than pointers to the derived types, and delete through them)
- but **does not need any other virtual dispatch** behavior for other methods (maybe there *are* no other methods? consider a simple protected "resource" container that needs a constructors/destructor/assignment but not much else)

...it's easiest to make the class abstract by making the destructor pure virtual *and* providing a definition (method body) for it.

For our hypothetical ABC:

You guarantee that it cannot be instantiated (even internal to the class itself, this is why private constructors may not be enough), you get the virtual behavior you want for the destructor, and you do not have to find and tag another method that doesn't need virtual dispatch as "virtual".

Share Edit Follow

answered Aug 2 '09 at 20:14
leander
8,337 ● 1 ● 27 ● 43

Going down the answers by score, this is the first one that is 1) correct, 2) written in a synthetical tone (as opposed to relying on examples and obiter dictum), 3) an answer to the question as it is written in the title, and 4) shows a pretty common use-case (i.e. “pure structs” with variable sizes and no methods). Kudos + upvote – DomQ Apr 8 at 14:48

8

From the answers I have read to your question, I couldn't deduce a good reason to actually use a pure virtual destructor. For example, the following reason doesn't convince me at all:

> Probably the real reason that pure virtual destructors are allowed is that to prohibit them would mean adding another rule to the language and there's no need for this rule since no ill-effects can come from allowing a pure virtual destructor.

In my opinion, pure virtual destructors can be useful. For example, assume you have two classes myClassA and myClassB in your code, and that myClassB inherits from myClassA. For the reasons mentioned by Scott Meyers in his book "More Effective C++", Item 33 "Making non-leaf classes abstract", it is better practice to actually create an abstract class myAbstractClass from which myClassA and myClassB inherit. This provides better abstraction and prevents some problems arising with, for example, object copies.

In the abstraction process (of creating class myAbstractClass), it can be that no method of myClassA or myClassB is a good candidate for being a pure virtual method (which is a prerequisite for myAbstractClass to be abstract). In this case, you define the abstract class's destructor pure virtual.

Hereafter a concrete example from some code I have myself written. I have two classes, Numerics/PhysicsParams which share common properties. I therefore let them inherit from the abstract class IParams. In this case, I had absolutely no method at hand that could be purely virtual. The setParameter method, for example, must have the same body for every subclass. The only choice that I have had was to make IParams' destructor pure virtual.

```
struct IParams
{
    IParams(const ModelConfiguration& aModelConf);
    virtual ~IParams() = 0;

    void setParameter(const N_Configuration::Parameter& aParam);

    std::map<std::string, std::string> m_Parameters;
};

struct NumericsParams : IParams
{
    NumericsParams(const ModelConfiguration& aNumericsConf);
    virtual ~NumericsParams();

    double dt() const;
    double ti() const;
    double tf() const;
};

struct PhysicsParams : IParams
{
    PhysicsParams(const N_Configuration::ModelConfiguration& aPhysicsConf);
    virtual ~PhysicsParams();

    double g()     const;
    double rho_i() const;
    double rho_w() const;
};
```

Share Edit Follow

answered Jan 11 '15 at 14:56

Laurent Michel
784 ● 1 ● 9 ● 18

1 I like this usage, but another way to "enforce" inheritance is by declaring the constructor of `IParam` to be protected, as was noted in some other comment. – rwols May 29 '15 at 7:25

6

Here I want to tell when we need **virtual destructor** and when we need **pure virtual destructor**

```
class Base
{
public:
    Base();
    virtual ~Base() = 0; // Pure virtual, now no one can create the Base Object
directly
};

Base::Base() { cout << "Base Constructor" << endl; }
Base::~Base() { cout << "Base Destructor" << endl; }
```

```
class Derived : public Base
{
public:
    Derived();
    ~Derived();
};

Derived::Derived() { cout << "Derived Constructor" << endl; }
Derived::~Derived() {   cout << "Derived Destructor" << endl; }


int _tmain(int argc, _TCHAR* argv[])
{
    Base* pBase = new Derived();
    delete pBase;

    Base* pBase2 = new Base(); // Error 1   error C2259: 'Base' : cannot
instantiate abstract class
}
```

1. When you want that no one should be able to create the object of Base class directly, use pure virtual destructor `virtual ~Base() = 0`. Usually at-least one pure virtual function is required, let's take `virtual ~Base() = 0`, as this function.
2. When you do not need above thing, only you need the safe destruction of Derived class object

   Base* pBase = new Derived(); delete pBase; pure virtual destructor is not required, only virtual destructor will do the job.

Share Edit Follow edited Sep 10 '15 at 9:46 answered Sep 10 '14 at 10:16

Anil8753
2,186 ● 2 ● 20 ● 31

5

If you want to stop instantiating of base class without making any change in your already implemented and tested derive class, you implement a pure virtual destructor in your base class.

Share Edit Follow answered Apr 21 '11 at 9:20

sukumar
51 ● 1 ● 1

2

You are getting into hypotheticals with these answers, so I will try to make a simpler, more down to earth explanation for clarity's sake.

The basic relationships of object oriented design are two: IS-A and HAS-A. I did not make those up. That is what they are called.

IS-A indicates that a particular object identifies as being of the class that is above it in a class hierarchy. A banana object is a fruit object if it is a subclass of the fruit class. This means that anywhere a fruit class can be used, a banana can be used. It is not reflexive , though. You can not substitute a base class for a specific class if that specific class is called for.

Has-a indicated that an object is part of a composite class and that there is an ownership relationship. It means in C++ that it is a member object and as such the onus is on the owning class to dispose of it or hand ownership off before destructing itself.

These two concepts are easier to realize in single-inheritance languages than in a multiple inheritance model like c++, but the rules are essentially the same. The complication comes when the class identity is ambiguous, such as passing a Banana class pointer into a function that takes a Fruit class pointer.

Virtual functions are, firstly, a run-time thing. It is part of polymorphism in that it is used to decide which function to run at the time it is called in the running program.

The virtual keyword is a compiler directive to bind functions in a certain order if there is ambiguity about the class identity. Virtual functions are always in parent classes (as far as I know) and indicate to the compiler that binding of member functions to their names should take place with the subclass function first and the parent class function after.

A Fruit class could have a virtual function color() that returns "NONE" by default. The Banana class color() function returns "YELLOW" or "BROWN".

But if the function taking a Fruit pointer calls color() on the Banana class sent to it -- which color() function gets invoked? The function would normally call Fruit::color() for a Fruit object.

That would 99% of the time not be what was intended. But if Fruit::color() was declared virtual then Banana:color() would be called for the object because the correct color() function would be bound to the Fruit pointer at the time of the call. The runtime will check what object the pointer points to because it was marked virtual in the Fruit class definition.

This is different than overriding a function in a subclass. In that case the Fruit pointer will call Fruit::color() if all it knows is that it IS-A pointer to Fruit.

So now to the idea of a "pure virtual function" comes up. It is a rather unfortunate phrase as purity has nothing to do with it. It means that it is intended that the base class method is never to be called. Indeed a pure virtual function can not be called. It must still be defined, however. A function signature must exist. Many coders make an empty implementation {} for completeness, but the compiler will generate one internally if not. In that case when the function is called even if the pointer is to Fruit , Banana::color() will be called as it is the only implementation of color() there is.

Now the final piece of the puzzle: constructors and destructors.

Pure virtual constructors are illegal, completely. That is just out.

But pure virtual destructors do work in the case that you want to forbid the creation of a base class instance. Only sub classes can be instantiated if the destructor of the base class is pure virtual. the convention is to assign it to 0.

```
virtual ~Fruit() = 0;  // pure virtual
Fruit::~Fruit(){}      // destructor implementation
```

You do have to create an implementation in this case. The compiler knows this is what you are doing and makes sure you do it right, or it complains mightily that it can not link to all the functions it needs to compile. The errors can be confusing if you are not on the right track as to how you are modeling your class hierarchy.

So you are forbidden in this case to create instances of Fruit, but allowed to create instances of Banana.

A call to delete of the Fruit pointer that points to an instance of Banana will call Banana::~Banana() first and then call Fuit::~Fruit(), always. Because no matter what, when you call a subclass destructor, the base class destructor must follow.

Is it a bad model? It is more complicated in the design phase, yes, but it can ensure that correct linking is performed at run-time and that a subclass function is performed where there is ambiguity as to exactly which subclass is being accessed.

If you write C++ so that you only pass around exact class pointers with no generic nor ambiguous pointers, then virtual functions are not really needed. But if you require run-time flexibility of types (as in Apple Banana Orange ==> Fruit) functions become easier and more versatile with less redundant code. You no longer have to write a function for each type of fruit, and you know that every fruit will respond to color() with its own correct function.

I hope this long-winded explanation solidifies the concept rather than confuses things. There are a lot of good examples out there to look at, and look at enough and actually run them and mess with them and you will get it.

Share Edit Follow   edited Apr 11 '17 at 7:46   answered Apr 11 '17 at 7:37
Chris Reid
395 ● 3 ● 7

1

This is a decade old topic :) Read last 5 paragraphs of Item #7 on "Effective C++" book for details, starts from "Occasionally it can be convenient to give a class a pure virtual destructor...."

Share Edit Follow

answered Nov 2 '17 at 8:28

J-Q

276 ●5 ●15

0

You asked for an example, and I believe the following provides a reason for a pure virtual destructor. I look forward to replies as to whether this is a *good* reason...

I do not want anyone to be able to throw the `error_base` type, but the exception types `error_oh_shucks` and `error_oh_blast` have identical functionality and I don't want to write it twice. The pImpl complexity is necessary to avoid exposing `std::string` to my clients, and the use of `std::auto_ptr` necessitates the copy constructor.

The public header contains the exception specifications that will be available to the client to distinguish different types of exception being thrown by my library:

```
// error.h

#include <exception>
#include <memory>

class exception_string;

class error_base : public std::exception {
 public:
  error_base(const char* error_message);
  error_base(const error_base& other);
  virtual ~error_base() = 0; // Not directly usable

  virtual const char* what() const;
 private:
  std::auto_ptr<exception_string> error_message_;
};

template<class error_type>
class error : public error_base {
 public:
   error(const char* error_message) : error_base(error_message) {}
   error(const error& other) : error_base(other) {}
   ~error() {}
};

// Neither should these classes be usable
class error_oh_shucks { virtual ~error_oh_shucks() = 0; }
class error_oh_blast { virtual ~error_oh_blast() = 0; }
```

And here is the shared implementation:

```
// error.cpp

#include "error.h"
#include "exception_string.h"

error_base::error_base(const char* error_message)
  : error_message_(new exception_string(error_message)) {}

error_base::error_base(const error_base& other)
  : error_message_(new exception_string(other.error_message_->get())) {}

error_base::~error_base() {}

const char* error_base::what() const {
  return error_message_->get();
}
```

The exception_string class, kept private, hides std::string from my public interface:

```
// exception_string.h

#include <string>

class exception_string {
 public:
  exception_string(const char* message) : message_(message) {}

  const char* get() const { return message_.c_str(); }
 private:
  std::string message_;
};
```

My code then throws an error as:

```
#include "error.h"

throw error<error_oh_shucks>("That didn't work");
```

The use of a template for `error` is a little gratuitous. It saves a bit of code at the expense of requiring clients to catch errors as:

```
// client.cpp

#include <error.h>

try {
} catch (const error<error_oh_shucks>&) {
} catch (const error<error_oh_blast>&) {
}
```

Share Edit Follow answered May 16 '14 at 16:58

Rai
968 ● 8 ● 20

0

Maybe there is another **REAL USE-CASE** of pure virtual destructor which I actually can't see in other answers :)

At first, I completely agree with marked answer: It is because forbidding pure virtual destructor would need an extra rule in language specification. But it's still not the use case that Mark is calling for :)

First imagine this:

```
class Printable {
  virtual void print() const = 0;
  // virtual destructor should be here, but not to confuse with another problem
};
```

and something like:

```
class Printer {
  void queDocument(unique_ptr<Printable> doc);
  void printAll();
};
```

Simply - we have interface `Printable` and some "container" holding anything with this interface. I think here it is quite clear why `print()` method is pure virtual. It could have some body but in case there is no default implementation, pure virtual is an ideal "implementation" (="must be provided by a descendant class").

And now imagine exactly the same except it is not for printing but for destruction:

```
class Destroyable {
  virtual ~Destroyable() = 0;
};
```

And also there could be a similar container:

```
class PostponedDestructor {
  // Queues an object to be destroyed later.
  void queObjectForDestruction(unique_ptr<Destroyable> obj);
  // Destroys all already queued objects.
  void destroyAll();
};
```

It's simplified use-case from my real application. The only difference here is that "special" method (destructor) was used instead of "normal" `print()`. But the reason why it is pure virtual is still the same - there is no default code for the method. A bit confusing could be the fact that there MUST be some destructor effectively and compiler actually generates an empty code for it. But from the perspective of a programmer pure virtuality still means: "I don't have any default code, it must be provided by derived classes."

I think it's no any big idea here, just more explanation that pure virtuality works really uniformly - also for destructors.

Share Edit Follow edited Jun 30 '17 at 9:40 answered Jun 30 '17 at 9:13

Jarek C
922 ● 5 ● 16

-2

we need to make destructor virtual bacause of the fact that , if we dont make the destructor virtual then compiler will only destruct the contents of base class , n all the derived classes will remain un changed , bacuse compiler will not call the destructor of any other class except the base class.

Share Edit Follow edited Sep 9 '13 at 19:00 answered Sep 9 '13 at 18:09

Brad Larson ♦
169k ● 45 ● 389 ● 564

Asad hashmi
1

-1: The question is not about why a destructor should be virtual. – Troubadour Sep 9 '13 at 19:21

Moreover, in certain situations destructors do not have to be virtual to achieve correct destruction. Virtual destructors are only needed when you end up calling `delete` on a pointer to base class when in fact it points to its derivative. – CygnusX1 Oct 31 '13 at 8:22

1 You are 100% correct. This is and has been in the past one of the number one sources of leaks and crashes in C++ programs, third only to trying to do things with null pointers and exceeding the bounds of arrays. A non-virtual base class destructor will be called on a generic pointer, bypassing the subclass destructor entirely if it is not marked virtual. If there are any dynamically-created objects belonging to the subclass, they will not get recovered by the base destructor on a call to delete. You are chugging along fine then BLUURRK! (hard to find where, too.) – Chris Reid Apr 11 '17 at 8:04