25.4 — Virtual destructors, virtual assignment, and overriding virtualization

ALEX[1] OCTOBER 18, 2024

Virtual destructors

Although C++ provides a default destructor for your classes if you do not provide one yourself, it is sometimes the case that you will want to provide your own destructor (particularly if the class needs to deallocate memory). You should *always* make your destructors virtual if you're dealing with inheritance. Consider the following example:

```
#include <iostream>
class Base
{
public:
    ~Base() // note: not virtual
    {
        std::cout << "Calling ~Base()\n";
    }
};

class Derived: public Base
{
private:
    int* m_array {};

public:
    Derived(int length)
      : m_array{ new int[length] }
    {
    }

    ~Derived() // note: not virtual (your compiler may warn you about this)
    {
        std::cout << "Calling ~Derived()\n";
        delete[] m_array;
    }
};

int main()
{
    Derived* derived { new Derived(5) };
    Base* base { derived };

    delete base;

    return 0;
}
```

Note: If you compile the above example, your compiler may warn you about the non-virtual destructor (which is intentional for this example). You may need to disable the compiler flag that treats warnings as errors to proceed.

Because base is a Base pointer, when base is deleted, the program looks to see if the Base destructor is virtual. It's not, so it assumes it only needs to call the Base destructor. We can see this in the fact that the above example prints:

```
Calling ~Base()
```

However, we really want the delete function to call Derived's destructor (which will call Base's destructor in turn), otherwise m_array will not be deleted. We do this by making Base's destructor virtual:

```
#include <iostream>
class Base
{
public:
    virtual ~Base() // note: virtual
    {
        std::cout << "Calling ~Base()\n";
    }
};

class Derived: public Base
{
private:
    int* m_array {};

public:
    Derived(int length)
      : m_array{ new int[length] }
    {
    }

    virtual ~Derived() // note: virtual
    {
        std::cout << "Calling ~Derived()\n";
        delete[] m_array;
    }
};

int main()
{
    Derived* derived { new Derived(5) };
    Base* base { derived };

    delete base;

    return 0;
}
```

Now this program produces the following result:

```
Calling ~Derived()
Calling ~Base()
```

Rule

Whenever you are dealing with inheritance, you should make any explicit destructors virtual.

As with normal virtual member functions, if a base class function is virtual, all derived overrides will be considered virtual regardless of whether they are specified as such. It is not necessary to create an empty

derived class destructor just to mark it as virtual.

Note that if you want your base class to have a virtual destructor that is otherwise empty, you can define your destructor this way:

```
virtual ~Base() = default; // generate a virtual default destructor
```

Virtual assignment

It is possible to make the assignment operator virtual. However, unlike the destructor case where virtualization is always a good idea, virtualizing the assignment operator really opens up a bag full of worms and gets into some advanced topics outside of the scope of this tutorial. Consequently, we are going to recommend you leave your assignments non-virtual for now, in the interest of simplicity.

Ignoring virtualization

Very rarely you may want to ignore the virtualization of a function. For example, consider the following code:

```
#include <string_view>
class Base
{
public:
    virtual ~Base() = default;
    virtual std::string_view getName() const { return "Base"; }
};

class Derived: public Base
{
public:
    virtual std::string_view getName() const { return "Derived"; }
};
```

There may be cases where you want a Base pointer to a Derived object to call Base::getName() instead of Derived::getName(). To do so, simply use the scope resolution operator:

```
#include <iostream>
int main()
{
    Derived derived {};
    const Base& base { derived };

    // Calls Base::getName() instead of the virtualized Derived::getName()
    std::cout << base.Base::getName() << '\n';

    return 0;
}
```

You probably won't use this very often, but it's good to know it's at least possible.

Should we make all destructors virtual?

This is a common question asked by new programmers. As noted in the top example, if the base class destructor isn't marked as virtual, then the program is at risk for leaking memory if a programmer later deletes a base class pointer that is pointing to a derived object. One way to avoid this is to mark all your destructors as virtual. But should you?

It's easy to say yes, so that way you can later use any class as a base class -- but there's a performance penalty for doing so (a virtual pointer added to every instance of your class). So you have to balance that cost, as well as your intent.

We'd suggest the following: If a class isn't explicitly designed to be a base class, then it's generally better to have no virtual members and no virtual destructor. The class can still be used via composition. If a class is designed to be used as a base class and/or has any virtual functions, then it should always have a virtual destructor.

If the decision is made to have a class not be inheritable, then the next question is whether it's possible to enforce this.

Conventional wisdom (as initially put forth by Herb Sutter, a highly regarded C++ guru) has suggested avoiding the non-virtual destructor memory leak situation as follows, "A base class destructor should be either public and virtual, or protected and non-virtual." A base class with a protected destructor can't be deleted using a base class pointer, which prevents deleting a derived class object through a base class pointer.

Unfortunately, this also prevents *any* use of the base class destructor by the public. That means:

- We shouldn't dynamically allocate base class objects by we have no conventional way to delete them (there are non-conventional workarounds, but yuck).
- We can't even statically allocate base class objects because the destructor isn't accessible when they go out of scope.

In other words, using this method, to make the derived class safe, we have to make the base class practically unusable by itself.

Now that the `final` specifier has been introduced into the language, our recommendations are as follows:

- If you intend your class to be inherited from, make sure your destructor is virtual and public.
- If you do not intend your class to be inherited from, mark your class as final. This will prevent other classes from inheriting from it in the first place, without imposing any other use restrictions on the class itself.

Next lesson

25.5 Early binding and late binding

2

Back to table of contents

3

Previous lesson

25.3 The override and final specifiers, and covariant return types

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.

200 COMMENTS

Newest

EmtyC

December 28, 2024 11:18 am PST

Could `final` be bypassed in anyway ? like the access specifiers are possibly bypassed through pointers under some conditions

0 Reply

Alex Author

Reply to EmtyC [8] January 3, 2025 5:56 pm PST

Not that I'm aware of, but I also haven't researched the topic more than cursorily.

1 Reply

EmtyC

Reply to Alex [9] January 4, 2025 4:15 am PST

I did some research (you sparked my curiosity enough :D), and it seems it is by the use of macros, `#define final` before any class or header that uses final and boom, disabled. The funny thing is this could be used to modify deeper implementation of the STL, like calls to the CRL, or even sdtout and sdtin (although quite useless and you may be better off without doing so)

One cool thing is `#define private public` and `#define protected public` if you want to explore the implementation of a class:

```cpp
#define private public

#define protected public

#include <iostream>

int main() {
    std::streambuf b{};
}
```

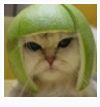
Normally not possible because the default constructor of `std::streambuf` is protected.
When you try to compile it, you get hit with:
`error C1189: #error: The C++ Standard Library forbids macroizing the keyword "private". Enable warning C4005 to find the forbidden define.`
(hmmm, he he he they really thought about this)
But still, for MSVC STL `#define _XKEYCHECK_H` solves this issue (thank you header guards >:D)
/I figured this out myself/

```cpp
#define private public

#define protected public

#define _XKEYCHECK_H

#include <iostream>

#include <streambuf>

int main() {

    std::streambuf c{};
}
```

Although sadly this throws a linker error (I couldn't solve it because no one seem to care to answer this question, so neither the net nor chatgpt have any idea /lol chatgpt gets high when it doesn't know the answer, it literally said std::streambuf is an abstract base class. So much for the expected end-er of software engineers/ so to solve this a great knowledge of the implementation is needed):
`error LNK2019: unresolved external symbol "__declspec(dllimport) public: __cdecl std::basic_streambuf<char,struct std::char_traits<char> >::basic_streambuf<char,struct std::char_traits<char> >(void)" (__imp_??0?$basic_streambuf@DU?$char_traits@D@std@@@std@@QEAA@XZ)`

Last edited 6 months ago by EmtyC

0 Reply



Alex Author

Reply to EmtyC [10] January 17, 2025 8:19 pm PST

It compiled and linked on GCC! I'm guessing MSVC defines the constructor in an anonymous namespace so it ends up with internal linkage...

1 Reply

EmtyC

Reply to Alex [11] January 18, 2025 7:17 am PST

Interesting choice from them :>

Last edited 5 months ago by EmtyC

0 Reply

mori

October 16, 2024 7:21 am PDT

When I click the chapter 25.3
The override and final specifiers, and covariant return types

Error establishing a database connection happens.
the page doesn't show up.

all the other chapters(pages) opens fine.

I opened it on chrome browser using m1 mac

0 Reply

mori

Reply to mori [12] October 16, 2024 8:19 am PDT

And this is an question for text about Should we make all destructors virtual?

in the text, there's a statement " Unfortunately, this also means the base class can’t be deleted through a base class pointer, which essentially means the class can’t be dynamically allocated or deleted except by a derived class."

but I think because we only concern about the destructor in here , isn't it possible for the class to be dynamically allocated (if we assume the constructor is in the public section)?

BTW, Thanks for this great tutorial Alex!

0 Reply

Alex Author

Reply to mori [13] October 18, 2024 3:10 pm PDT

Yes. I'll clarify in the text. Thanks for pointing this out.

Last edited 8 months ago by Alex

0 Reply

wangzhichao

July 24, 2024 9:00 pm PDT

Could you please explain the output of the following code for me?

```cpp
#include <iostream>

class Base
{
public:
    ~Base()
    {
        std::cout << "Calling ~Base()\n";
    }
};

class Derived: public Base
{
private:
    int* m_array {};

public:
    Derived(int length)
        : m_array { new int[length] }
    {
    }

    ~Derived()
    {
        std::cout << "Calling ~Derived\n";
        delete[] m_array;
    }
};

int main()
{
    Derived d { 8 };
    Base& b { d };

    return 0;
}

/*
g++ (Ubuntu 13.1.0-8ubuntu1~20.04.2) 13.1.0

Output:
Calling ~Derived
Calling ~Base()
*/
```

I expect that the output is: `Calling ~Base()`

Last edited 11 months ago by wangzhichao

0 Reply

sandersan

Reply to wangzhichao [14] April 20, 2025 2:15 am PDT

In your case, the `d` is an object allocated on the stack, which will be cleaned up at the moment when the program exits, no matter what you do.

In the case above from the lesson, the object is dynamically allocated on the heap, rather than on the stack, which means you are responsible for the cleanup.

This is my own opinion, dunno if it's correct.

0 Reply

tri lolis

Reply to wangzhichao [14] November 6, 2024 7:30 am PST

Derived::~Derived called the Base constructor:
~Derived()
{
00007FF72ADF2070 mov qword ptr [rsp+8],rcx
00007FF72ADF2075 push rbp
00007FF72ADF2076 push rdi
00007FF72ADF2077 sub rsp,108h
00007FF72ADF207E lea rbp,[rsp+20h]
00007FF72ADF2083 lea rcx,[__BC18738A_main@cpp (07FF72AE0506Dh)]
00007FF72ADF208A call __CheckForDebuggerJustMyCode (07FF72ADF1479h)
00007FF72ADF208F nop
std::cout << "Calling ~Derived\n";
00007FF72ADF2090 lea rdx,[string "Calling ~Derived\n" (07FF72ADFBC10h)]
00007FF72ADF2097 mov rcx,qword ptr [__imp_std::cout (07FF72AE031B0h)]
00007FF72ADF209E call std::operator<<<std::char_traits<char> > (07FF72ADF1091h)
00007FF72ADF20A3 nop
delete[] m_array;
00007FF72ADF20A4 mov rax,qword ptr [this]
00007FF72ADF20AB mov rax,qword ptr [rax]
00007FF72ADF20AE mov qword ptr [rbp+0C8h],rax
00007FF72ADF20B5 mov rcx,qword ptr [rbp+0C8h]
00007FF72ADF20BC call operator delete[] (07FF72ADF150Fh)
00007FF72ADF20C1 nop
}
00007FF72ADF20C2 mov rcx,qword ptr [this]
00007FF72ADF20C9 call Base::~Base (07FF72ADF1528h)
00007FF72ADF20CE nop
00007FF72ADF20CF lea rsp,[rbp+0E8h]
00007FF72ADF20D6 pop rdi
00007FF72ADF20D7 pop rbp
00007FF72ADF20D8 ret

0 Reply

Ajit Mali

July 19, 2024 11:32 pm PDT

> virtualizing the assignment operator really opens up a bag full of worms and gets into some advanced topics outside of the scope of this tutorial

what are these advanced topic that even this website can't explain

1 Reply

Alex Author

Reply to Ajit Mali [15] July 22, 2024 12:13 pm PDT

If I explained it to you then it wouldn't be outside the scope of this tutorial. :)

Stack overflow has some decent threads on the topic.

2 Reply

bigbear

Reply to Alex [16] August 5, 2024 11:28 am PDT

i love ur jokes lol

0 Reply

Ajit Mali

Reply to Alex [16] July 22, 2024 7:50 pm PDT

lol :D

ok I will check there

0 Reply

rafal

July 18, 2024 8:46 am PDT

I did some testing with virtual destructors :

First I simulated what would happen when it's marked not virutal, Obviously we get only ~Base() call.

```cpp
#include <iostream>
class Base
{
public:
    ~Base() // note: Not virtual
    {
        std::cout << "Calling ~Base()\n";
    }
};

class Derived: public Base
{
private:
    int* m_array {};

public:
    Derived(int length)
      : m_array{ new int[length] }
    {
    }

    ~Derived() // note: override
    {
        std::cout << "Calling ~Derived()\n";
        delete[] m_array;
    }

    Derived(const Derived&) = delete;
    void operator=(const Derived&) = delete;
};

int main()
{
    Derived* derived { new Derived(5) };
    Base* base { derived };

    delete base;

    return 0;
}
```

Then I changed derived destructor to virtual, obviously still only ~Base()

But then I wanted to archive following:

If derived has to dallocate memory or do something else so that destructor has to be called. I though what about override and tried that and got what excepted that is compile error. Using override makes code not compile when base class doesn't have virtual destructor

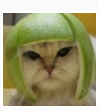
```cpp
#include <iostream>
class Base
{
public:
    ~Base() // note: Not virtual
    {
        std::cout << "Calling ~Base()\n";
    }
};

class Derived: public Base
{
private:
    int* m_array {};

public:
    Derived(int length)
      : m_array{ new int[length] }
    {
    }

//main.cpp:22:5: error: ‘Derived::~Derived()’ marked ‘override’, but does not override
//   22 |     ~Derived() override // note: override
//      |     ^
//
    ~Derived() override // note: override
    {
        std::cout << "Calling ~Derived()\n";
        delete[] m_array;
    }

    Derived(const Derived&) = delete;
    void operator=(const Derived&) = delete;
};

int main()
{
    Derived* derived { new Derived(5) };
    Base* base { derived };

    delete base;

    return 0;
}
```

Compile error makes it easy to realize that `~Base()` should be virtual, making IMO this better approach than to mark destructor as public virtual or making class unusable with pointers.

And output is as excepted:

Calling ~Derived()
Calling ~Base()

0 Reply



Alex Author

Reply to rafal [17] July 18, 2024 10:03 pm PDT

The issue is that we may write a class not intending it to be inherited from (and thus making the destructor non-virtual). Then we come along later and inherit from it, and have issues.

Ideally, the base class should be set up to either allow inheritance properly, or not at all, so there's nothing that a derived class has to remember to do or check. That's why we recommend that the base class should either have a virtual destructor OR be marked as final.

1 Reply

rafal

July 18, 2024 6:39 am PDT

In first example we have

```
Derived(const Derived&) = delete;
void operator=(const Derived&) = delete;
```

Why

```
Derived(Derived&&) = default;
void operator=(Derived&&) = default;
```

is not there too? (Will it need default constructor to work?)

0 Reply

Alex Author

Reply to rafal [18] July 18, 2024 9:53 pm PDT

The deleted copy constructors aren't needed here, so I have removed them.

1 Reply

Dck

April 6, 2024 7:27 pm PDT

Hi Alex, what is the rationale behind dynamically creating the `Derived` object in this snippet:

```
int main()
{
    Derived* derived { new Derived(5) };
    Base* base { derived };

    delete base;

    return 0;
}
```

instead of doing this:

```
int main()
{
    Derived derived(5);
    Base* base { &derived };

    return 0;
}
```

Thank you!

1 Reply

Alex Author

Reply to Dck [19] April 8, 2024 3:54 pm PDT

In the top snippet, we're dynamically allocating a Derived and then deleting it through a Base pointer. This only works properly if the Base destructor is virtual.

In the bottom snippet, Derived is statically allocated so the Derived gets automatically deleted (which calls the Derived destructor).

In this particular example, the latter works, but in future lessons we'll see examples where the latter isn't possible so we need to understand how to deal with the former.

3 Reply

Dck

Reply to Alex [20] April 8, 2024 4:12 pm PDT

Thank you!

0 Reply

Roman

March 26, 2024 4:11 am PDT

IMO it's great that you included "Ignoring virtualization" section. I like the comprehension you approaching topics with!!

As for having-virtual-assignment-op-bad-idea, just for myself for now I stick with the following explanation: following "the rule of 3/5" we keep consistent logic in our ctors and operators=. Making operator= virtual would motivate us to make ctors virtual which doesn't make sense.

Also out of this discussion (https://stackoverflow.com/questions/669818/virtual-assignment-operator-c)[21] I brought that since virtual function must have the same signature, we would end up with long list of overloaded operators= in given derived class (one for each parent in the inheritance chain as right-hand-side param). Not to mention that operator= should be implemented via base class operator= (to copy base's private members) which has it own complications because base's operator= is virtual (to put it briefly).
Could you, please, verify is this explanation ok to stick with for now?

0 Reply

Alex Author

Reply to Roman [22] March 28, 2024 12:54 pm PDT

Yep.

0 Reply

Ryan

January 23, 2024 12:14 pm PST

While my compiler didn't complain about the lack of a virtual destructor in the first example, it *did* complain about the implicit sign conversion when initializing the array length at line 18. I guess it's up to you if you want to do something about that.

0 Reply

Kirill

January 18, 2024 3:51 pm PST

>A class with a protected destructor can’t be deleted via a pointer, thus preventing the accidental deleting of a derived class through a base pointer when the base class has a non-virtual destructor. Unfortunately, this also means the base class can’t be deleted through a base class pointer, which essentially means the class can’t be dynamically allocated or deleted except by a derived class. This also precludes using smart pointers (such as std::unique_ptr and std::shared_ptr) for such classes, which limits the usefulness of that rule. It also means the base class can’t be allocated on the stack. That’s a pretty heavy set of penalties.
So basically we can't use Base class at any way?
We can't do this since it's compile error:

```
1 | Base base{}; //compile error
```

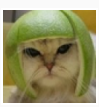
We can allocate it dynamically:

```
1 | Base base{new Base()};
```

However we can't deallocate it, so it's no point to dynamically allocate at first place:

```
1 | delete base; //compile error
```

1 Reply



Alex Author

Reply to Kirill [23] January 19, 2024 9:23 am PST

Correct. However, you can allocate a Derived (that inherits from Base) on the stack, or dynamically allocate/delete a Derived (though a Derived*).

3 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/early-binding-and-late-binding/
3. https://www.learncpp.com/
4. https://www.learncpp.com/cpp-tutorial/the-override-and-final-specifiers-and-covariant-return-types/
5. https://www.learncpp.com/virtual-destructors-virtual-assignment-and-overriding-virtualization/
6. https://www.learncpp.com/cpp-tutorial/virtual-functions/
7. https://gravatar.com/
8. https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/#comment-605819
9. https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/#comment-606187
10. https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/#comment-606216
11. https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/#comment-606689
12. https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/#comment-603187
13. https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/#comment-603190
14. https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/#comment-600055
15. https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/#comment-599868
16. https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/#comment-599945
17. https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/#comment-599780
18. https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/#comment-599776
19. https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/#comment-595495
20. https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/#comment-595546
21. https://stackoverflow.com/questions/669818/virtual-assignment-operator-c
22. https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/#comment-595122
23. https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/#comment-592544
24. https://g.ezoic.net/privacy/learncpp.com