When do we need to define destructors? [duplicate]

Asked 7 years, 3 months ago Active 3 years, 8 months ago Viewed 51k times

33

18

This question already has answers here:

What is The Rule of Three? (8 answers)

Closed 7 years ago.

I read that destructors need to be defined when we have pointer members and when we define a base class, but I am not sure if I completely understand. One of the things I am not sure about is whether or not defining a default constructor is useless or not, since we are always given a default constructor by default. Also, I am not sure if we need to define default constructor to implement the RAII principle (do we just need to put resource allocation in a constructor and not define any destructor?).

```
class A
{

public:
    ~Account()
    {
        delete [] brandname;
        delete b;

        //do we need to define it?

    };

    something(){} =0; //virtual function (reason #1: base class)

private:
    char *brandname; //c-style string, which is a pointer member (reason #2: has a
pointer member)
    B* b; //instance of class B, which is a pointer member (reason #2)
    vector<B*> vec; //what about this?



}

class B: public A
{
    public something()
    {
    cout << "nothing" << endl;
    }

    //in all other cases we don't need to define the destructor, nor declare it?
}
```

c++ destructor

669 2 7 12

7 Though the answer might be relevant the questions are not the same. Not a duplicate. I think that this is a good question and would like to hear the answer myself. – nonsensickle Mar 18 '14 at 21:16

3 Your 2nd sentence is a bit confusing. I think you meant destructor where you wrote constructor? – Filipe Gonçalves Mar 18 '14 at 21:27

5 Answers

Active | Oldest | Votes

34

The rule of Three and The Rule of Zero

The good ol' way of handling resources was with the Rule of Three (now Rule of Five due to move semantic), but recently another rule is taking over: the Rule of Zero.

The idea, but you should really read the article, is that resource management should be left to other specific classes.

On this regard the standard library provides a nice set of tools like: `std::vector`, `std::string`, `std::unique_ptr` and `std::shared_ptr`, effectively removing the need for custom destructors, move/copy constructors, move/copy assignment and default constructors.

How to apply it to your code

In your code you have a lot of different resources, and this makes for a great example.

The string

If you notice `brandname` is effectively a "dynamic string", the standard library not only saves you from C-style string, but automatically manages the memory of the string with `std::string`.

The dynamically allocated B

The second resource appears to be a dynamically allocated `B`. If you are dynamically allocating for other reasons other than "I want an optional member" you should definitely use `std::unique_ptr` that will take care of the resource (deallocating when appropriate) automatically. On the other hand, if you want it to be an *optional member* you can use `std::optional` instead.

The collection of Bs

The last resource is just an array of `B`s. That is easily managed with an `std::vector`. The standard library allows you to choose from a variety of different containers for your different needs; Just to mention some of them: `std::deque`, `std::list` and `std::array`.

```
class A {
private:
    std::string brandname;
    std::unique_ptr<B> b;
    std::vector<B> vec;
public:
    virtual void something(){} = 0;
};
```

Which is both safe and readable.

Share Improve this answer Follow

edited Jun 20 '20 at 9:12
Community ♦
1 1

answered Mar 18 '14 at 21:29
Shoe
70.5k 30 151 251

15 Ok, but this hardly answers the question. Q: "When would I define a destructor?" A: "Use a `vector` ." Huh? – Ed S. Mar 18 '14 at 21:45

6 @EdS., The answer is implicly: "Never, use a `vector` ". :) – Shoe Mar 18 '14 at 21:46

4 Well I don't think that's a very good answer. Understanding is never a bad thing, and you can't really believe that no one but the standard library implementors will ever need to define their own destructors. – Ed S. Mar 18 '14 at 21:48

3 I think that the answer is in understanding *The Rule of Zero* and *The Rule of Three* correctly. Hence your answer and @Claudiordgz complement each other nicely. The rest is just a matter of philosophy in my opinion. Both +1. – nonsensickle Mar 18 '14 at 21:57

3 @Jeffrey That rule of zero is awesome man, thank you very much, I hadn't hear of it before – Claudiordgz Mar 18 '14 at 22:02

12

As @nonsensickle points out, the questions is too broad... so I'm gonna try to tackle it with everything I know...

The first reason to re define the destructor would be in **The Rule of Three** which is on part the **item 6** in Scott Meyers Effective C++ but not entirely. The rule of three says that if you re defined the destructor, copy constructor, or copy assignment operations then that means you should rewrite all three of them. The reason is that if you had to rewrite your own version for one, then the compiler defaults will no longer be valid for the rest.

Another example would be the one pointed out by Scott Meyers in Effective C++

> When you try to delete a derived class object through a base class pointer and the base class has a non virtual destructor, the results are undefined.

And then he continues

> If a class does not contain any virtual functions, that is often an indication that it is