

What is The Rule of Three?

Asked 14 years, 1 month ago Modified 2 years ago Viewed 393k times



2549



- What does *copying an object* mean?
- What are the *copy constructor* and the *copy assignment operator*?
- When do I need to declare them myself?
- How can I prevent my objects from being copied?

c++

copy-constructor

assignment-operator

c++-faq

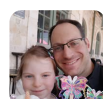
rule-of-three

Share

Improve this question

Follow

edited May 26, 2018 at 12:07



Rann Lifshitz

4,090 ● 4 ● 26 ● 43

asked Nov 13, 2010 at 13:27



fredoverflow

263k ● 99 ● 400 ● 670

66 Please read [this whole thread](#) and [the c++ - faq tag wiki](#) before you vote to close. – [sbi](#) Nov 13, 2010 at 14:06

19 @Binary: At least take the time to read the comment discussion *before* you cast a vote. The text used to be much simpler, but Fred was asked to expand on it. Also, while that's four questions *grammatically*, it really is just one

question with several aspects to it. (If you disagree to that, then prove your POV by answering each of those questions on its own and let us vote on the results.) – [sbi](#) Nov 15, 2010 at 23:02

1 Fred, here's an interesting addition to your answer regarding C++1x: stackoverflow.com/questions/4782757/.... How do we deal with this? – [sbi](#) Jan 25, 2011 at 14:18

7 Related: [The Law of The Big Two](#) – [Nemanja Trifunovic](#) Jun 27, 2011 at 16:39

3 @paxdiablo [The Rule of Zero](#) to be exact. – [rubenvb](#) Sep 25, 2015 at 11:08

8 Answers

Sorted by:

Highest score (default)



Introduction

2140

C++ treats variables of user-defined types with *value semantics*. This means that objects are implicitly copied in various contexts, and we should understand what "copying an object" actually means.



Let us consider a simple example:



```
class person
{
    std::string name;
    int age;

public:
    person(const std::string& name, int age) : name(name)
    {
    }
}
```

```
};

int main()
{
    person a("Bjarne Stroustrup", 60);
    person b(a);    // What happens here?
    b = a;          // And here?
}
```

(If you are puzzled by the `name(name), age(age)` part, this is called a [member initializer list](#).)

Special member functions

What does it mean to copy a `person` object? The `main` function shows two distinct copying scenarios. The initialization `person b(a);` is performed by the *copy constructor*. Its job is to construct a fresh object based on the state of an existing object. The assignment `b = a` is performed by the *copy assignment operator*. Its job is generally a little more complicated because the target object is already in some valid state that needs to be dealt with.

Since we declared neither the copy constructor nor the assignment operator (nor the destructor) ourselves, these are implicitly defined for us. Quote from the standard:

The [...] copy constructor and copy assignment operator, [...] and destructor are special member functions. [*Note: The implementation will implicitly declare these member functions for*

some class types when the program does not explicitly declare them. The implementation will implicitly define them if they are used. [...] *end note*] [n3126.pdf section 12 §1]

By default, copying an object means copying its members:

The implicitly-defined copy constructor for a non-union class X performs a memberwise copy of its subobjects. [n3126.pdf section 12.8 §16]

The implicitly-defined copy assignment operator for a non-union class X performs memberwise copy assignment of its subobjects. [n3126.pdf section 12.8 §30]

Implicit definitions

The implicitly-defined special member functions for `person` look like this:

```
// 1. copy constructor
person(const person& that) : name(that.name), age(that
{
}

// 2. copy assignment operator
person& operator=(const person& that)
{
    name = that.name;
```

```
        age = that.age;
        return *this;
    }

    // 3. destructor
    ~person()
    {
    }
}
```

Memberwise copying is exactly what we want in this case: `name` and `age` are copied, so we get a self-contained, independent `person` object. The implicitly-defined destructor is always empty. This is also fine in this case since we did not acquire any resources in the constructor. The members' destructors are implicitly called after the `person` destructor is finished:

After executing the body of the destructor and destroying any automatic objects allocated within the body, a destructor for class X calls the destructors for X's direct [...] members
[n3126.pdf 12.4 §6]

Managing resources

So when should we declare those special member functions explicitly? When our class *manages a resource*, that is, when an object of the class is *responsible* for that resource. That usually means the resource is *acquired* in the constructor (or passed into the constructor) and *released* in the destructor.

Let us go back in time to pre-standard C++. There was no such thing as `std::string`, and programmers were in love with pointers. The `person` class might have looked like this:

```
class person
{
    char* name;
    int age;

public:

    // the constructor acquires a resource:
    // in this case, dynamic memory obtained via new[]
    person(const char* the_name, int the_age)
    {
        name = new char[strlen(the_name) + 1];
        strcpy(name, the_name);
        age = the_age;
    }

    // the destructor must release this resource via delete[]
    ~person()
    {
        delete[] name;
    }
};
```

Even today, people still write classes in this style and get into trouble: *"I pushed a person into a vector and now I get crazy memory errors!"* Remember that by default, copying an object means copying its members, but copying the `name` member merely copies a pointer, *not* the character array it points to! This has several unpleasant effects:

1. Changes via `a` can be observed via `b`.

2. Once `b` is destroyed, `a.name` is a dangling pointer.
3. If `a` is destroyed, deleting the dangling pointer yields [undefined behavior](#).
4. Since the assignment does not take into account what `name` pointed to before the assignment, sooner or later you will get memory leaks all over the place.

Explicit definitions

Since memberwise copying does not have the desired effect, we must define the copy constructor and the copy assignment operator explicitly to make deep copies of the character array:

```
// 1. copy constructor
person(const person& that)
{
    name = new char[strlen(that.name) + 1];
    strcpy(name, that.name);
    age = that.age;
}

// 2. copy assignment operator
person& operator=(const person& that)
{
    if (this != &that)
    {
        delete[] name;
        // This is a dangerous point in the flow of execution
        // We have temporarily invalidated the class invariant
        // and the next statement might throw an exception
        // leaving the object in an invalid state :(
        name = new char[strlen(that.name) + 1];
        strcpy(name, that.name);
        age = that.age;
    }
}
```

```
        return *this;
    }
```

Note the difference between initialization and assignment: we must tear down the old state before assigning it to `name` to prevent memory leaks. Also, we have to protect against the self-assignment of the form `x = x`. Without that check, `delete[] name` would delete the array containing the *source* string, because when you write `x = x`, both `this->name` and `that.name` contain the same pointer.

Exception safety

Unfortunately, this solution will fail if `new char[...]` throws an exception due to memory exhaustion. One possible solution is to introduce a local variable and reorder the statements:

```
// 2. copy assignment operator
person& operator=(const person& that)
{
    char* local_name = new char[strlen(that.name) + 1]
    // If the above statement throws,
    // the object is still in the same state as before
    // None of the following statements will throw an
    strcpy(local_name, that.name);
    delete[] name;
    name = local_name;
    age = that.age;
    return *this;
}
```


This also takes care of self-assignment without an explicit check. An even more robust solution to this problem is the [copy-and-swap idiom](#), but I will not go into the details of exception safety here. I only mentioned exceptions to make the following point: **Writing classes that manage resources is hard.**

Noncopyable resources

Some resources cannot or should not be copied, such as file handles or mutexes. In that case, simply declare the copy constructor and copy assignment operator as `private` without giving a definition:

```
private:  
  
    person(const person& that);  
    person& operator=(const person& that);
```

Alternatively, you can inherit from `boost::noncopyable` or declare them as deleted (in C++11 and above):

```
    person(const person& that) = delete;  
    person& operator=(const person& that) = delete;
```

The rule of three

Sometimes you need to implement a class that manages a resource. (Never manage multiple resources in a single

class, this will only lead to pain.) In that case, remember the **rule of three**:

If you need to explicitly declare either the destructor, copy constructor or copy assignment operator yourself, you probably need to explicitly declare all three of them.

(Unfortunately, this "rule" is not enforced by the C++ standard or any compiler I am aware of.)

The rule of five

From C++11 on, an object has 2 extra special member functions: the move constructor and move assignment. The rule of five states to implement these functions as well.

An example with the signatures:

```
class person
{
    std::string name;
    int age;

public:
    person(const std::string& name, int age);
    person(const person &) = default;
    person(person &&) noexcept = default;
    person& operator=(const person &) = default;
    person& operator=(person &&) noexcept = default;
    ~person() noexcept = default;
};
```

The rule of zero

The rule of 3/5 is also referred to as the rule of 0/3/5. The zero part of the rule states that you are allowed to not write any of the special member functions when creating your class.

Advice

Most of the time, you do not need to manage a resource yourself, because an existing class such as `std::string` already does it for you. Just compare the simple code using a `std::string` member to the convoluted and error-prone alternative using a `char*` and you should be convinced. As long as you stay away from raw pointer members, the rule of three is unlikely to concern your own code.

Share Improve this answer

Follow

edited Nov 28, 2022 at 12:38



Rohan Bari

7,716 ● 3 ● 16 ● 38


answered Nov 13, 2010 at 13:27




fredoverflow

263k ● 99 ● 400 ● 670

-
- 4 Fred, I'd feel better about my up-vote if (A) you wouldn't spell out badly implemented assignment in copyable code and add a note saying it's wrong and look elsewhere in the fingerprint; either use c&s in the code or just skip over implementing all these members (B) you would shorten the

first half, which has little to do with the RoT; (C) you would discuss the introduction of move semantics and what that means for the RoT. – [sbi](#) Nov 13, 2010 at 14:00 

7 But then the post should be made C/W, I think. I like that you keep the terms mostly accurate (i.e that you say "copy assignment operator", and that you don't tap into the common trap that assignment couldn't imply a copy).
– [Johannes Schaub - litb](#) Nov 13, 2010 at 14:21 

5 @Prasoon: I don't think cutting out half of the answer would be seen as "fair editing" of a non-CW answer. – [sbi](#) Nov 13, 2010 at 14:33

77 It would be great if you update your post for C++11 (i.e. move constructor / assignment) – [Alexander Malakhov](#) Sep 13, 2012 at 3:42

7 @solalito Anything you must release after use: concurrency locks, file handles, database connections, network sockets, heap memory... – [fredoverflow](#) Nov 18, 2015 at 10:13



The [Rule of Three](#) is a rule of thumb for C++, basically saying

560



If your class needs any of

- a **copy constructor**,
- an **assignment operator**,
- or a **destructor**,

defined explicitly, then it is likely to need **all three of them**.



The reasons for this is that all three of them are usually used to manage a resource, and if your class manages a resource, it usually needs to manage copying as well as freeing.

If there is no good semantic for copying the resource your class manages, then consider to forbid copying by declaring (not [defining](#)) the copy constructor and assignment operator as `private`.

(Note that the forthcoming new version of the C++ standard (which is C++11) adds move semantics to C++, which will likely change the Rule of Three. However, I know too little about this to write a C++11 section about the Rule of Three.)

Share Improve this answer

Follow

edited May 23, 2017 at 12:03



Community Bot

1 • 1

answered Nov 13, 2010 at 14:22



sbi

224k • 46 • 264 • 447

3 Another solution to prevent copying is to inherit (privately) from a class that cannot be copied (like `boost::noncopyable`). It can also be much clearer. I think that C++0x and the possibility to "delete" functions could help here, but forgot the syntax :/ – [Matthieu M.](#) Nov 13, 2010 at 16:33

2 @Matthieu: Yep, that works, too. But unless `noncopyable` is part of the std lib, I don't consider it much of an improvement. (Oh, and if you forgot the deletion syntax, you

forgot mor ethan I ever knew. :)) – [sbi](#) Nov 13, 2010 at 17:20

4 @Daan: See [this answer](#). However, I'd recommend to stick to [Martinho's Rule of Zero](#). To me, this is one of the most important rules of thumb for C++ coined in the last decade.
– [sbi](#) Jun 4, 2014 at 17:46

4 Martinho's Rule of Zero now better (without apparent adware takeover) located on [archive.org](#) – [Nathan Kidd](#) Jun 27, 2018 at 18:31



The law of the big three is as specified above.

173

An easy example, in plain English, of the kind of problem it solves:



Non default destructor



You allocated memory in your constructor and so you need to write a destructor to delete it. Otherwise you will cause a memory leak.



You might think that this is job done.

The problem will be, if a copy is made of your object, then the copy will point to the same memory as the original object.

Once, one of these deletes the memory in its destructor, the other will have a pointer to invalid memory (this is called a dangling pointer) when it tries to use it things are going to get hairy.

Therefore, you write a copy constructor so that it allocates new objects their own pieces of memory to destroy.

Assignment operator and copy constructor

You allocated memory in your constructor to a member pointer of your class. When you copy an object of this class the default assignment operator and copy constructor will copy the value of this member pointer to the new object.

This means that the new object and the old object will be pointing at the same piece of memory so when you change it in one object it will be changed for the other object too. If one object deletes this memory the other will carry on trying to use it - eek.

To resolve this you write your own version of the copy constructor and assignment operator. Your versions allocate separate memory to the new objects and copy across the values that the first pointer is pointing to rather than its address.

Share Improve this answer

Follow

edited Jan 9, 2018 at 18:27



rmobis


27k ● 8 ● 69 ● 67

answered May 14, 2012 at 14:22



Stefan

3,859 ● 3 ● 34 ● 46

-
- 4 So If we use a copy constructor then the copy is made but at a different memory location altogether and if we do not use copy constructor then copy is made but it points to the same memory location. is that what you are trying to say? So a copy without copy constructor means that a new pointer will be there but pointing to the same memory location however if we have copy constructor explicitly defined by user then we will have a separate pointer pointing to a different memory location but having the data. – [Unbreakable](#) Jan 4, 2015 at 14:01 
-
- 4 Sorry, I replied to this ages ago but my reply does not seem to still be here :-(Basically, yes - you get it :-) – [Stefan](#) Jul 27, 2016 at 13:35
-



48



Basically if you have a destructor (not the default destructor) it means that the class that you defined has some memory allocation. Suppose that the class is used outside by some client code or by you.

```
MyClass x(a, b);  
MyClass y(c, d);  
x = y; // This is a shallow copy if assignment operator is not defined
```

If MyClass has only some primitive typed members a default assignment operator would work but if it has some pointer members and objects that do not have assignment operators the result would be unpredictable. Therefore we can say that if there is something to delete in destructor of a class, we might need a deep copy operator which means we should provide a copy constructor and assignment operator.

Share Improve this answer

edited Sep 11, 2015 at 11:39

Follow



Shakti Malik

2,407 ● 27 ● 33

answered Dec 31, 2012 at 19:29



fatma.ekici

2,827 ● 5 ● 31 ● 31



36



What does copying an object mean? There are a few ways you can copy objects--let's talk about the 2 kinds you're most likely referring to--deep copy and shallow copy.

Since we're in an object-oriented language (or at least are assuming so), let's say you have a piece of memory allocated. Since it's an OO-language, we can easily refer to chunks of memory we allocate because they are usually primitive variables (ints, chars, bytes) or classes we defined that are made of our own types and primitives. So let's say we have a class of Car as follows:

```
class Car //A very simple class just to demonstrate wh
mean.
//It's pseudocode C++/Javaish, I assume strings do not
{
    private String sPrintColor;
    private String sModel;
    private String sMake;

    public changePaint(String newColor)
    {
        this.sPrintColor = newColor;
    }
}
```

```

public Car(String model, String make, String color) //
{
    this.sPrintColor = color;
    this.sModel = model;
    this.sMake = make;
}

public ~Car() //Destructor
{
    //Because we did not create any custom types, we aren't
    //Anytime your object goes out of scope / program coll
    guy gets called + all other related destructors.
    //Since we did not use anything but strings, we have n
    handle.
    //The assumption is being made that the 3 strings will
    destructor and that it is being called automatically--
    you would need to do it here.
}

public Car(const Car &other) // Copy Constructor
{
    this.sPrintColor = other.sPrintColor;
    this.sModel = other.sModel;
    this.sMake = other.sMake;
}

public Car &operator =(const Car &other) // Assignment
{
    if(this != &other)
    {
        this.sPrintColor = other.sPrintColor;
        this.sModel = other.sModel;
        this.sMake = other.sMake;
    }
    return *this;
}

}

```

A deep copy is if we declare an object and then create a completely separate copy of the object...we end up with 2 objects in 2 completely sets of memory.

```
Car car1 = new Car("mustang", "ford", "red");
Car car2 = car1; //Call the copy constructor
car2.changePaint("green");
//car2 is now green but car1 is still red.
```

Now let's do something strange. Let's say car2 is either programmed wrong or purposely meant to share the actual memory that car1 is made of. (It's usually a mistake to do this and in classes is usually the blanket it's discussed under.) Pretend that anytime you ask about car2, you're really resolving a pointer to car1's memory space...that's more or less what a shallow copy is.

```
//Shallow copy example
//Assume we're in C++ because it's standard behavior if
if you do not have a constructor written for an operator
//Now let's assume I do not have any code for the assignment
like I do above...with those now gone, C++ will use the default
to resolve the address of where car2 exists and delete the memory
the memory associated with your car.*/
Car car1 = new Car("ford", "mustang", "red");
Car car2 = car1;
car2.changePaint("green");//car1 is also now green
delete car2; /*I get rid of my car which is also really
no longer allocated to the program.*/
```

So regardless of what language you're writing in, be very careful about what you mean when it comes to copying objects because most of the time you want a deep copy.

What are the copy constructor and the copy assignment operator? I have already used them above. The copy

constructor is called when you type code such as `Car car2 = car1;` Essentially if you declare a variable and assign it in one line, that's when the copy constructor is called. The assignment operator is what happens when you use an equal sign-- `car2 = car1;`. Notice `car2` isn't declared in the same statement. The two chunks of code you write for these operations are likely very similar. In fact the typical design pattern has another function you call to set everything once you're satisfied the initial copy/assignment is legitimate--if you look at the longhand code I wrote, the functions are nearly identical.

When do I need to declare them myself? If you are not writing code that is to be shared or for production in some manner, you really only need to declare them when you need them. You do need to be aware of what your program language does if you choose to use it 'by accident' and didn't make one--i.e. you get the compiler default. I rarely use copy constructors for instance, but assignment operator overrides are very common. Did you know you can override what addition, subtraction, etc. mean as well?

How can I prevent my objects from being copied? Override all of the ways you're allowed to allocate memory for your object with a private function is a reasonable start. If you really don't want people copying them, you could make it public and alert the programmer by throwing an exception and also not copying the object.

Follow



David Szalai

2,529 ● 31 ● 48

answered Oct 17, 2012 at 16:37



user1701047

737 ● 5 ● 7

7 The question was tagged C++. This pseudo-code exposition does little to clarify anything about the well-defined "Rule Of Three" at best, and just spreads confusion at worst. – [sehe](#)
Jun 11, 2014 at 22:55



When do I need to declare them myself?

29



The Rule of Three states that if you declare any of a

1. copy constructor
2. copy assignment operator
3. destructor




then you should declare all three. It grew out of the observation that the need to take over the meaning of a copy operation almost always stemmed from the class performing some kind of resource management, and that almost always implied that

- whatever resource management was being done in one copy operation probably needed to be done in the other copy operation and

- the class destructor would also be participating in management of the resource (usually releasing it). The classic resource to be managed was memory, and this is why all Standard Library classes that manage memory (e.g., the STL containers that perform dynamic memory management) all declare “the big three”: both copy operations and a destructor.

A consequence of the Rule of Three is that the presence of a user-declared destructor indicates that simple member wise copy is unlikely to be appropriate for the copying operations in the class. That, in turn, suggests that if a class declares a destructor, the copy operations probably shouldn't be automatically generated, because they wouldn't do the right thing. At the time C++98 was adopted, the significance of this line of reasoning was not fully appreciated, so in C++98, the existence of a user declared destructor had no impact on compilers' willingness to generate copy operations. That continues to be the case in C++11, but only because restricting the conditions under which the copy operations are generated would break too much legacy code.



How can I prevent my objects from being copied?

Declare copy constructor & copy assignment operator as private access specifier.

```

class MemoryBlock
{
public:

//code here

private:
MemoryBlock(const MemoryBlock& other)
{
    cout<<"copy constructor"<<endl;
}

// Copy assignment operator.
MemoryBlock& operator=(const MemoryBlock& other)
{
    return *this;
}
};

int main()
{
    MemoryBlock a;
    MemoryBlock b(a);
}

```

In C++11 onwards you can also declare copy constructor & assignment operator deleted

```

class MemoryBlock
{
public:
MemoryBlock(const MemoryBlock& other) = delete

// Copy assignment operator.
MemoryBlock& operator=(const MemoryBlock& other) =delete
};

int main()
{
    MemoryBlock a;
}

```

```
MemoryBlock b(a);  
}
```

Share Improve this answer

edited Jan 12, 2016 at 10:19

Follow

answered Jan 12, 2016 at 9:54



Ajay yadav

4,481 ● 4 ● 33 ● 42



19

Many of the existing answers already touch the copy constructor, assignment operator and destructor.

However, in post C++11, the introduction of move semantic may expand this beyond 3.



Recently Michael Claisse gave a talk that touches this topic: <http://channel9.msdn.com/events/CPP/C-PP-Con-2014/The-Canonical-Class>

Share Improve this answer

answered Jan 7, 2015 at 5:38

Follow



xyz

910 ● 2 ● 8 ● 16



12

Rule of three in C++ is a fundamental principle of the design and the development of three requirements that if there is clear definition in one of the following member function, then the programmer should define the other two members functions together. Namely the following three member functions are indispensable: destructor, copy constructor, copy assignment operator.





Copy constructor in C++ is a special constructor. It is used to build a new object, which is the new object equivalent to a copy of an existing object.

Copy assignment operator is a special assignment operator that is usually used to specify an existing object to others of the same type of object.

There are quick examples:

```
// default constructor
My_Class a;

// copy constructor
My_Class b(a);

// copy constructor
My_Class c = a;

// copy assignment operator
b = a;
```

Share Improve this answer

Follow

edited Oct 16, 2016 at 4:57



ReinstateMonica316704
0

792 ● 9 ● 30

answered Aug 12, 2014 at 4:27



Marcus Thornton

6,193 ● 8 ● 51 ● 53

-
- 7 Hi, your answer doesn't add anything new. The others cover the subject in much more depths, and more accurately - your answer is approximate and in fact wrong in some places (namely there is no "must" here; it's "very probably should").

It'd really not worth your while posting this sort of answer to questions that have been thoroughly answered already. Unless you have new things to add. – [Mat](#) Aug 15, 2014 at 16:26

- 1 Also, there are *four* quick examples, which are *somehow* related to *two* of the *three* that the Rule of Three is talking about. Too much confusion. – [anatolyg](#) Nov 3, 2014 at 15:41
-



Highly active question. Earn 10 reputation (not counting the [association bonus](#)) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.