What is The Rule of Three?

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

edited Nov 15 '10 at 11:07

asked Nov 13 '10 at 13:27

fredoverflow

117k ● 36 ● 227 ● 492

25 Please **read this whole thread and the `c++-faq` tag wiki before you vote to close**. – sbi Nov 13 '10 at 14:06

7 @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 '10 at 23:02

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 '11 at 14:18

2 Related: The Law of The Big Two – Nemanja Trifunovic Jun 27 '11 at 16:39

Keep in mind that, as of C++11, I think this has been upgraded to the rule of five, or something like that. – paxdiablo Aug 19 '15 at 10:31

7 Answers

Introduction

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), age(age)
    {
    }
};

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.age)
{
}

// 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 invariants,
        // 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 to `name` to prevent memory leaks. Also, we have to protect against 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 exception :)
    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 (C++0x):

```
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.)

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.

edited Aug 30 '14 at 15:38
Sam
4,020 ● 7 ● 22 ● 44

answered Nov 13 '10 at 13:27
fredoverflow
117k ● 36 ● 227 ● 492

2 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 fineprint; 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 '10 at 14:00

5 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 '10 at 14:21

2 @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 '10 at 14:33

3 Also, I may have overread it, but you do not mention that the copy assingment operator should check for identity before doing anything. – Björn Pollex Nov 13 '10 at 14:34

26 It would be great if you update your post for C++11 (i.e. move constructor / assignment) – Alexander Malakhov Sep 13 '12 at 3:42

The Rule of Three is a rule of thumb for C++, basically saying

> If your class needs any of
>
> - a **copy constructor**,
> - an **assignment operator**,
> - or a **destructor**,
>
> defined explictly, 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 (currently usually referred to as C++0x or C++1x) adds move semantics to C++, which will likely change the Rule of Three. However, I know too little about this to write a C++1x section about the Rule of Three.)

edited Dec 31 '13 at 12:28
einpoklum
5,936 ● 5 ● 36 ● 75

answered Nov 13 '10 at 14:22
sbi
121k ● 31 ● 162 ● 328

1 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 '10 at 16:33

1 @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 '10 at 17:20