Rule-of-Three becomes Rule-of-Five with C++11?

So, after watching this wonderful lecture on rvalue references, I thought that every class would benefit of such a "move constructor", `template<class T> MyClass(T&& other)` ***edit*** and of course a "move assignment operator", `template<class T> MyClass& operator=(T&& other)` as Philipp points out in his answer, if it has dynamically allocated members, or generally stores pointers. Just like you *should* have a copy-ctor, assignment operator and destructor if the points mentioned before apply. Thoughts?

c++ constructor c++11 rvalue-reference rule-of-three

edited Dec 19 '11 at 16:00
Flexo ♦
64.3k ● 19 ● 131 ● 205

asked Jan 24 '11 at 13:51
Xeo
98.1k ● 33 ● 238 ● 342

Some additional information can be found here brookspatola.com/the-rule-of-five-c11 (just copying from deleted answer) – Aleks Jul 13 '16 at 13:55

Cripey. Your link to the "wonderful" lecture.... epic. I never saw this series before. – kevinarpe Aug 23 '16 at 14:56

8 Answers

I'd say the Rule of Three becomes the Rule of Three, Four and Five:

> Each class should explicitly define exactly one of the following set of special member functions:
>
> - None
> - Destructor, copy constructor, copy assignment operator
>
> In addition, each class that explicitly defines a destructor may explicitly define a move constructor and/or a move assignment operator.
>
> Usually, one of the following sets of special member functions is sensible:
>
> - None (for many simple classes where the implicitly generated special member functions are correct and fast)
> - Destructor, copy constructor, copy assignment operator (in this case the class will not be movable)
> - Destructor, move constructor, move assignment operator (in this case the class will not be copyable, useful for resource-managing classes where the underlying resource is not copyable)
> - Destructor, copy constructor, copy assignment operator, move constructor (because of copy elision, there is no overhead if the copy assignment operator takes its argument by value)
> - Destructor, copy constructor, copy assignment operator, move constructor, move assignment operator

Note that move constructor and move assignment operator won't be generated for a class that explicitly declares any of the other special member functions, that copy constructor and copy assignment operator won't be generated for a class that explicitly declares a move constructor or move assignment operator, and that a class with a explicitly declared destructor and implicitly defined copy constructor or implicitly defined copy assignment operator is considered deprecated. In particular, the following perfectly valid C++03 polymorphic base class

```
class C {
  virtual ~C() { }   // allow subtype polymorphism
};
```

should be rewritten as follows:

```
class C {
  C(const C&) = default;              // Copy constructor
  C(C&&) = default;                   // Move constructor
  C& operator=(const C&) & = default; // Copy assignment operator
  C& operator=(C&&) & = default;      // Move assignment operator
  virtual ~C() { }                    // Destructor
};
```

A bit annoying, but probably better than the alternative (automatic generation of all special member functions).

In contrast to the Rule of the Big Three, where failing to adhere to the rule can cause serious damage, not explicitly declaring the move constructor and move assignment operator is generally fine but often suboptimal with respect to efficiency. As mentioned above, move constructor and move assignment operators are only generated if there is no explicitly declared copy constructor, copy assignment operator or destructor. This is not symmetric to the traditional C++03 behavior with respect to auto-generation of copy constructor and copy assignment operator, but is much safer. So the possibility to define move constructors and move assignment operators is very useful and creates new possibilities (purely movable classes), but classes that adhere to the C++03 Rule of the Big Three will still be fine.

For resource-managing classes you can define the copy constructor and copy assignment operator as deleted (which counts as definition) if the underlying resource cannot be copied. Often you still want move constructor and move assignment operator. Copy and move assignment operators will often be implemented using `swap`, as in C++03. If you have a move constructor and move assignment operator, specializing `std::swap` will become unimportant because the generic `std::swap` uses the move constructor and move assignment operator if available, and that should be fast enough.

Classes that are not meant for resource management (i.e., no non-empty destructor) or subtype polymorphism (i.e., no virtual destructor) should declare none of the five special member functions; they will all be auto-generated and behave correct and fast.

edited Jan 3 '15 at 0:40

CoryKramer
67.1k ● 11 ● 69 ● 124

answered Jan 24 '11 at 14:10

Philipp
34.4k ● 10 ● 68 ● 90

1 @Philipp: Hm, right... a pass-by-value assignment operators 'other' would be move-constructed if you just implement the move-ctor if I got that right? And the remaining pointer copies and assignments would just be optimized by the compiler I think... – Xeo Jan 24 '11 at 14:33

2 @Xeo: I believe that if the class is not copyable, then you cannot pass instances of it by value even if the copy can be elided. In that case you should declare a true move assignment operator using a rvalue reference (an assignment operator that takes its argument by value is a copy assignment operator by §12.8/19, which you wouldn't want if the class is not copyable). For copyable and movable classes, the compiler should use copy elision or call the move constructor. – Philipp Jan 24 '11 at 14:42

1 @Omni: A virtual function cannot be explicitly defaulted on declaration (§8.4.2/2 and the last example in §8.4.2/5). – Philipp Feb 8 '11 at 19:50

9 Have the rules changed ever since C++11 was passed? I believe `struct C { virtual ~C() = default; };` is now allowed and the most concise option. The prohibition ("- it shall not be virtual") from n3242 is not present anymore in n3290 and GCC allows it while previously it didn't. – Luc Danton Sep 21 '11 at 18:18

2 @ВЈовић No, it's not a typo. Here is a good explanation for it: stackoverflow.com/a/12306344/1174378 – Mihai Todor Sep 6 '12 at 19:00

I can't believe that nobody linked to this.

Basically article argues for "Rule of Zero". It is not appropriate for me to quote entire article but I believe this is the main point:

> Classes that have custom destructors, copy/move constructors or copy/move assignment operators should deal exclusively with ownership. Other classes should not have custom destructors, copy/move constructors or copy/move assignment operators.

Also this bit is IMHO important:

> Common "ownership-in-a-package" classes are included in the standard library: `std::unique_ptr` and `std::shared_ptr`. Through the use of custom deleter objects, both have been made flexible enough to manage virtually any kind of resource.

edited Jul 4 '17 at 3:08

MultiplyByZer0
2,480 ● 2 ● 18 ● 36

answered Dec 19 '12 at 11:30

NoSenseEtAl
6,328 ● 10 ● 67 ● 169

2 See here and here for my thoughts on this whole matter. :) – Xeo Dec 19 '12 at 12:33

5 The link moved to http://flamingdangerzone.com/cxx11/rule-of-zero/ – Fabian Jun 30 '15 at 4:56

I don't think so, the rule of three is a rule of thumb that states that a class that implements one of the following but not them all is probably buggy.

1. Copy constructor
2. Assignment operator
3. Destructor

However leaving out the move constructor or move assignment operator does not imply a bug. It *may* be a missed opportunity at optimization (in most cases) or that move semantics aren't relevant for this class but this isn't a bug.

While it may be best practice to define a move constructor when relevant, it isn't mandatory. There are many cases in which a move constructor isn't relevant for a class (e.g. `std::complex`) and all classes that behave correctly in C++03 will continue to behave correctly in C++0x even if they don't define a move constructor.

answered Jan 24 '11 at 14:56

Motti
65.2k ● 29 ● 148 ● 217

Yes, I think it would be nice to provide a move constructor for such classes, but remember that:

- It's only an optimization.

  Implementing only one or two of the copy constructor, assignment operator or destructor will probably lead to bugs, while not having a move constructor will just potentially reduce performance.

- Move constructor cannot always be applied without modifications.

  Some classes always have their pointers allocated, and thus such classes always delete their pointers in the destructor. In these cases you'll need to add extra checks to say whether their pointers are allocated or have been moved away (are now null).

edited Feb 1 '13 at 13:29

cmh
6,675 ● 1 ● 21 ● 38

answered Jan 24 '11 at 14:00

peoro
16.6k ● 14 ● 66 ● 124

20 It's not just an optimization, move semantics are important in perfect forwarding and some classes (`unique_ptr`) cannot be implemented without move semantics. – Puppy Jan 24 '11 at 14:17

@DeadMG: in general you're right, but in this context move semantics is just an optimization. Here I'm talking about already existing classes which respect the rule of three; `unique_ptr` and perfect forwarding are some special cases... – peoro Jan 24 '11 at 14:21

1 @peoro: I think that a C++03 class that declares private copy constructor and copy assignment operators (or inherits from `boost::noncopyable`) can be called to obey the Rule of Three. (Otherwise we have to introduce different terminology, e.g. "Rule of the Big One and the Small Two"). – Philipp Jan 24 '11 at 14:26

1 @DeadMG: `unique_ptr` does respect the Rule of Five. – Philipp Jan 24 '11 at 14:27

4 `some classes have always their pointers allocated...` in this case a move is usually implemented as a swap. Just as simple and fast. (Actually faster since it moves deallocation to the destructor of the rvalue) – Mooing Duck Aug 26 '11 at 19:11

Here's a short update on the current status and related developments since Jan 24 '11.

According to the C++11 Standard (see Annex D's [depr.impldec]):

> The implicit declaration of a copy constructor is deprecated if the class has a user-declared copy assignment operator or a user-declared destructor. The implicit declaration of a copy assignment operator is deprecated if the class has a user-declared copy constructor or a user-declared destructor.

It was actually proposed to obsolete the deprecated behavior *giving C++14 a true "rule of five" instead of the traditional "rule of three".* In 2013 the EWG voted against this proposal to be implemented in C++2014. The major rationale for the decision on the proposal had to do with general concerns about breaking existing code.

Recently, it has been proposed again to adapt the C++11 wording so as to achieve the informal Rule of Five, namely that

> **no copy function, move function, or destructor be compiler-generated if any of these functions is user-provided.**

If approved by the EWG, the "rule" is likely to be adopted for C++17.

answered Jan 11 '15 at 19:17

Andrey Rekalo
186 ● 2 ● 6

1 Thanks for the update. As some of these C++ questions get old, it's helpful to see how question and/or answers

are affected by newer language versions. – cb4 Apr 2 '16 at 15:26

Basically, it's like this: If you don't declare any move operations, you should respect the rule of three. If you declare a move operation, there is no harm in "violating" the rule of three as the generation of compiler-generated operations has gotten very restrictive. Even if you don't declare move operations and violate the rule of three, a C++0x compiler is expected to give you a warning in case one special function was user-declared and other special functions have been auto-generated due to a now deprecated "C++03 compatibility rule".

I think it's safe to say that this rule becomes a little less significant. The real problem in C++03 is that implementing different copy semantics required you to user-declare *all* related special functions so that none of them is compiler-generated (which would otherwise do the wrong thing). But C++0x changes the rules about special member function generation. If the user declares just one of these functions to change the copy semantics it'll prevent the compiler from auto-generating the remaining special functions. This is good because a missing declaration turns a runtime error into a compilation error now (or at least a warning). As a C++03 compatibility measure some operations are still generated but this generation is deemed deprecated and should at least produce a warning in C++0x mode.

Due to the rather restrictive rules about compiler-generated special functions and the C++03 compatibility, the rule of three stays the rule of three.

Here are some exaples that should be fine with newest C++0x rules:

```
template<class T>
class unique_ptr
{
   T* ptr;
public:
   explicit unique_ptr(T* p=0) : ptr(p) {}
   ~unique_ptr();
   unique_ptr(unique_ptr&&);
   unique_ptr& operator=(unique_ptr&&);
};
```

In the above example, there is no need to declare any of the other special functions as deleted. They simply won't be generated due to the restrictive rules. The presence of a user-declared move operations disables compiler-generated copy operations. But in a case like this:

```
template<class T>
class scoped_ptr
{
   T* ptr;
public:
   explicit scoped_ptr(T* p=0) : ptr(p) {}
   ~scoped_ptr();
};
```

a C++0x compiler is now expected to produce a warning about possibly compiler-generated copy operations that might do the wrong thing. Here, the rule of three matters and should be respected. A warning in this case is totally appropriate and gives the user the chance to handle the bug. We can get rid of the issue via deleted functions:

```
template<class T>
class scoped_ptr
{
   T* ptr;
public:
   explicit scoped_ptr(T* p=0) : ptr(p) {}
   ~scoped_ptr();
   scoped_ptr(scoped_ptr const&) = delete;
   scoped_ptr& operator=(scoped_ptr const&) = delete;
};
```

So, the rule of three still applies here simply because of the C++03 compatibility.

edited Jan 24 '11 at 20:14

answered Jan 24 '11 at 19:55

sellibitze

20.8k ● 2 ● 54 ● 82

In fact, N3126 does define the copy constructor and copy assignment operator of `unique_ptr` as deleted—anybody knows why? – Philipp Jan 24 '11 at 20:49

@Philipp: The restrictive rules are newer than N3126. However, N3225 still declares the copy operations of unique_ptr as deleted. This is not necessary anymore, but it's also not wrong. So, there is no real need to change the spec of unique_ptr. – sellibitze Jan 24 '11 at 21:58

N3126 had the less strict rules that a copy constructor would not be implicitly declared if there is a user-declared move constructor, and that a copy assignment operator would not be implicitly declared if there is a user-declared move assignment operator. `unique_ptr` has both user-declared move constructor and move assignment operator, so I think the user-declared copy constructor and copy assignment operator wouldn't be necessary even when applying the N3126 rules. Not really important, but since the conventions used by the standard library classes might be interpreted as being best – Philipp Jan 24 '11 at 22:44

practices, it would be nice to know whether the explicitly declared copy constructor and copy assignment operator are intentional. – Philipp Jan 24 '11 at 22:45

We cannot say that rule of 3 becomes rule of 4 (or 5) now without breaking all existing code that does enforce rule of 3 and does not implement any form of move semantics.

Rule of 3 means if you implement one you must implement all 3.

Also not aware there will be any auto-generated move. The purpose of "rule of 3" is because they automatically exist and if you implement one, it is most likely the default implementation of the other two is wrong.

answered Jan 24 '11 at 14:07

CashCow
25.6k ● 3 ● 45 ● 77

In the general case, then yes, the rule of three just became the of five, with the move assignment operator and move constructor added in. However, not *all* classes are copyable and movable, some are just movable, some are just copyable.

answered Jan 24 '11 at 14:16

Puppy
119k ● 24 ● 183 ● 390

1 I believe even if a class is not copyable, you want to define copy constructor and assignment operator (as deleted). So a movable resource-managing class should define all five, too. – Philipp Jan 24 '11 at 14:23

1 @Philipp, I strongly disagree, many classes don't support move semantics and it makes no sense to define two redundant functions just for some sense of aesthetics. Why should `std::complex` care about rvalue references? – Motti Jan 24 '11 at 15:25

@Motti: Why does it define regular copy semantics? Virtually all resources that can be copied can be moved. – Puppy Jan 24 '11 at 15:33

@Motti: Philipp said they should be defined *as deleted*! So you should explicitly adverse the fact that they don't support the operation. – Konrad Rudolph Jan 24 '11 at 15:36

@Konrad this seems overly verbose to me, once a cctor is defined the mctor will not be defined (as I understand the current draft). Would you also define your default constructor as deleted for every class the defines a custom constructor? – Motti Jan 24 '11 at 15:51