Enforcing the Rule of zero

Juan Alday (juan@greenwiresoft.com)

Abstract.

The Rule of Three, considered good practice for many years, transitioned to the Rule of Five under C++11.
A proper application of the Rule of Five and resource management makes users transition to the Rule of Zero, where the preferable option is to write classes that declare/define neither a destructor nor a copy/move constructor or copy/move assignment operator.

Introduction

The rule of three [Koenig/Moo1] is a rule of thumb coined by Marshall Cline, dating back to 1991. It states that if a class defines a destructor it should almost always define a copy constructor and an assignment operator.

In reality it is two rules:
a) If you define a destructor, you probably need to define a copy constructor and an assignment operator
b) If you defined a copy constructor or assignment operator, you probably will need both, as well as the destructor

Although considered good practice, the compiler can't enforce it: The C++ standard [N1316] mandates that all three have to be implicitly declared if not defined by the user:

> ***12.4 § 3*** *If a class has no user-declared destructor, a destructor is declared implicitly*
>
> ***12.8 § 4*** *If the class definition does not explicitly declare a copy constructor, one is declared implicitly.*
>
> ***12.8 § 10*** *If the class definition does not explicitly declare a copy assignment operator, one is declared implicitly*

This leads to code that is fundamentally broken, yet perfectly valid:

```
struct A
{
   A(const char* str) : myStr(strdup(str)) {}
   ~A() {free(myStr);}
private:
   char* myStr;
};
```

(Note: Most static analysis tools will detect these errors, but that's beyond the scope of this article)

C++11 [N3242] added wording to the Standard, deprecating the previous behavior.

> ***D.3*** *Implicit declaration of copy functions* ***[depr.impldec]***
> *The implicit definition of a copy constructor as defaulted is deprecated if the class has a user-declared copy assignment operator or a user-declared destructor. The implicit definition of a copy assignment operator as*

defaulted is deprecated if the class has a user-declared copy constructor or a user-declared destructor. In a future revision of this International Standard, these implicit definitions could become deleted

This means that compilers keep generating a defaulted copy constructor, assignment operator and destructor if no user-defined declaration is found, but at least now they might issue a warning.
Before and after the adoption of C++11 there were ongoing discussions on the need to ban, rather than deprecate this behavior.
C++14 [N3797] kept the same compromise. Amongst other reasons, there was real concern about the amount of existing code that would break. That does not mean that C++17 will not switch to a full ban [N3839] if adequate wording can be drafted, thus enforcing the need to properly enforce the Rule of Three as a standard of best practices.

C++11 introduced move operations, transforming the Rule of Three into the Rule of Five. Also, in the absence of a user-defined move constructor/move assignment operator, the compiler (under very specific rules) would generate implicit versions.

There was a lot of controversy regarding automatic generation of implicit move constructor and assignment operator [Abrahams1][Abrahams2][N3153][N3174][N3201][N3203], and the wording was adjusted to reach a compromise and tighten the rules under which they would get defaulted.

C++11 [N3242] says that move operations are ONLY implicitly declared if the following set of conditions occur:

12.8 § 9
If the definition of a class X does not explicitly declare a move constructor, one will be implicitly declared as defaulted if and only if
- *— X does not have a user-declared copy constructor,*
- *— X does not have a user-declared copy assignment operator,*
- *— X does not have a user-declared move assignment operator,*
- *— X does not have a user-declared destructor, and*
- *— The move constructor would not be implicitly defined as deleted.*

12.8 § 20
If the definition of a class X does not explicitly declare a move assignment operator, one will be implicitly declared as defaulted if and only if
- *— X does not have a user-declared copy constructor,*
- *— X does not have a user-declared move constructor,*
- *— X does not have a user-declared copy assignment operator,*
- *— X does not have a user-declared destructor, and*
- *— The move assignment operator would not be implicitly defined as deleted.*

Unlike the Rule of three, the Rule of Five is partially enforced by the compiler. You get a default move copy constructor and move assignment operator if and only if none of the other four are defined/defaulted by the user.

C++14 expanded the wording, and now an explicit declaration of a move constructor or move assignment operator marks the defaulted copy constructor and assignment operator as deleted. This means that explicit move operations disable default copying/assignment. This is as close as you get to a real enforcement of the rule of five by a compiler.

12.8 § 7

If the class definition does not explicitly declare a copy constructor, one is declared implicitly. If the class definition declares a move constructor or move assignment operator, the implicitly declared copy constructor is defined as deleted; otherwise, it is defined as defaulted (8.4). The latter case is deprecated if the class has a user-declared copy assignment operator or a user-declared destructor.
12.8 § 18
If the class definition does not explicitly declare a copy assignment operator, one is declared implicitly. If the class definition declares a move constructor or move assignment operator, the implicitly declared copy assignment operator is defined as deleted; otherwise, it is defined as defaulted (8.4). The latter case is deprecated if the class has a user-declared copy constructor or a user-declared destructor.

At this point, the Rule of Five transitions in fact to the Rule of Zero, a term coined by Peter Sommerlad [Sommerlad1]:
"Write your classes in a way that you do not need to declare/define neither a destructor, nor a copy/move constructor or copy/move assignment operator"
"Use smart pointers & standard library classes for managing resources"

Martinho Fernandes [Fernandes1] further extends the definition, tying it to the Single Responsability Principle:
"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."

There are two cases where users will want to bypass the compiler and declare them yourself:

a) Managed resources
b) polymorphic deletion and/or virtual functions

Managed resources

When possible, use a combination of standard class templates like std::unique_ptr and std::shared_ptr with custom deleters to avoid managing resources [sommerlad1]

In C++98/03, a class managing resources would look something like:

```
struct A
{
   A() : myPtr(API::InitializeData()) {}
   ~A() {API::ReleaseData(myPtr)) {}
   A(const A& rhs) : myPtr(API::CloneData(rhs.myPtr)) {}
   A& operator=(const A& rhs)
   {
      if (this != &rhs)
         myPtr = API::ReleaseAndClone(myPtr, rhs.myPtr);
      return *this;
   }
   API::Resource* myPtr;
};
```

In C++11, with move operations, it could be implemented as

```
struct A
{
   A() : myPtr(API::GiveMeStaticData()) {}
   ~A()
```

```
    {
        if (myPtr !- nullptr)
            API::ReleaseMyData(myPtr));
    }
    A(const A& rhs) : myPtr(API::CloneData(rhs.myPtr))=delete;
    A& operator=(const A& rhs)
    {
        if (this != &rhs)
            myPtr = API::ReleaseAndClone(myPtr, rhs.myPtr);
        return *this;
    }
    A(A&& rhs) : myPtr(rhs.myPtr) {rhs.myPtr = nullptr;}
    A& operator=(A&& rhs)
    {
        A tmp {std::move(rhs)};
        std::swap(myPtr, tmp.myPtr);
        return *this;
    }
private:
    API::Resource* myPtr;
};
```

Applying the Rule of Zero, the code would be more expressive:

```
struct A
{
    A() : myPtr(API::GiveMeStaticData(), &API::ReleaseMyData) {}
private:
    std::shared_ptr<API::Resource> myPtr;
};
```

std::unique_ptr and std::shared_ptr solve the issue of managing pointer types. For non-pointers, help is on its way [N3949] in the form of RAII wrappers. If the proposal gets approved, users will have a standard way of managing generic resources without the need to write their own code, combining standard containers, unique_ptr, shared_ptr, scope_guard and unique_resource.

Polymorphic deletion / virtual functions

One question on the rule of zero is what to do when we want to support polymorphic deletion, or when our classes have virtual functions

For years it has been taught that classes supporting inheritance and/or with virtual functions should usually have a virtual destructor.[Stroustrup1][Koenig/Moo2]

Note: In reality, not all base classes with virtual functions need virtual destructors. Herb Sutter's advice [Sutter1]:
If A is intended to be used as a base class, and if callers should be able to destroy polymorphically, then make A::~A public and virtual. Otherwise make it protected (and not-virtual)

So a base class with a virtual function like

```
struct A
{
        virtual void foo();
};
```

should be written, following standard practices, as:

```
struct A
{
        virtual ~A() {}
        virtual void foo();
};
```

One side effect is that now both classes are different. After declaring the destructor, A doesn't support move operations. You still get copy and assignment, but that's as far as you can go.

In C++11 the correct way to define it, in order to get move semantics is:

```
struct A
{
        virtual ~A() =default;
        A(A&&)=default;
        A& operator=(A&&)=default;

        virtual void foo();
};
```

And in C++14 we need to define all five, since otherwise we disable copy/assignment:

```
struct A
{
        virtual ~A() = default;
        A(const A&)=default;
        A& operator=(const A&)=default;
        A(A&&)=default;
        A& operator=(A&&)=default;

        virtual void foo();
};
```

In this case we have applied a consistent rule of five, defaulting all five functions due to the virtual destructor. The question is: Do we really need to do that? Why can't we apply the Rule of Zero?

The second part of the Rule of Zero (*"Use smart pointers & standard library classes for managing resources")* [Sommerlad1] gives us the answer:

Under current practice, the reason for the virtual destructor is to free resources via a pointer to base. Under the Rule of Zero we shouldn't really be managing our own resources, including instances of our classes. We should be using shared or unique pointers instead. Once we do that, our original class goes back to it's original implementation and concerns about polymorphic destruction go away:

```
struct A
```

```
{
        virtual void foo() = 0;
};
struct B : A
{
        void foo() {}
};
int main()
{
        std::shared_ptr<A> mySharedPtr = std::make_shared<B>();
        ...
}
```

Conclusion

The Rule of Zero helps you to do more by writing less. Use it as a guideline when you can and apply the Rule of Five only when you have to.

There is almost no need to manage your own resources so resist the temptation to implement your own copy/assign/move copy/move assign/destructor functions.

Managed resources can be resources inside your class definition or instances of your classes themselves. Refactoring the code around standard containers and class templates like unique_ptr or shared_ptr will make your code more readable and maintainable.

Acknowledgements

Thanks to Peter Sommerlad and Jonathan Wakely for their very helpful comments on drafts of this material.

References

[Abrahams1] Dave Abrahams, Implicit Move Must Go. http://cpp-next.com/archive/2010/10/implicit-move-must-go/
[Abrahams2] Dave Abrahams. w00t w00t nix nix http://cpp-next.com/archive/2011/02/w00t-w00t-nix-nix/
[Koenig/Moo1] Andrew Koenig/Barbara Moo. C++ Made Easier: The Rule of Three : http://www.drdobbs.com/c-made-easier-the-rule-of-three/184401400
[Koenig/Moo2] Andrew Koenig/Barbara Moo. Ruminations in C++. Destructors are special. ISBN-10 0-201-42339-1
[Fernandes1] Martinho Fernandes. Rule of Zero. http://flamingdangerzone.com/cxx11/2012/08/15/rule-of-zero.html
[N1316] Draft for C++03. Standard for Programming Language C++ http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2001/n1316/
[N3153] Dave Abrahams. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3153.htm
[N3174] Bjarne Stroustrup. To move or not to move. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3174.pdf
[N3201] Bjarne Stroustrup. Moving right along. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3201.pdf
[N3203] Jens Maurer. Tightening the conditions for generating implicit moves http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3203.htm
[N3242] Draft for C++11. Standard for Programming Language C++ http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf
[N3797] Draft for C++14. Standard for Programming Language C++ http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf
[N3839] Walter Brown . Proposing the Rule of Five, v2. http://open-std.org/JTC1/SC22/WG21/docs/papers/2014/n3839.pdf
[N3949] Peter Sommerlad/A L Sandoval. Scoped Resource.- Generic RAII wrapper for the Standard Library http://isocpp.org/files/papers/N3949.pdf
[Sommerlad1] Peter Sommerlad, Simpler C++ with C++11/14, http://wiki.hsr.ch/PeterSommerlad/files/MeetingCPP2013_SimpleC++.pdf
[Stroustrup1] Bjarne Stroustrup. The C++ Programming Language, Fourth Edition, section 17.2.5. ISBN -13 978-0-321-56384-2

[Sutter1] Herb Sutter/Andrei Alexandrescu. C++ Coding Standards. Ch 50. ISBN -13 978-0-321-11358-0