The rule of three/five/zero

Rule of three

If a class requires a user-defined destructor, a user-defined copy constructor, or a user-defined copy assignment operator, it almost certainly requires all three.

Because C++ copies and copy-assigns objects of user-defined types in various situations (passing/returning by value, manipulating a container, etc), these special member functions will be called, if accessible, and if they are not user-defined, they are implicitly-defined by the compiler.

The implicitly-defined special member functions are typically incorrect if the class is managing a resource whose handle is an object of non-class type (raw pointer, POSIX file descriptor, etc), whose destructor does nothing and copy constructor/assignment operator performs a "shallow copy" (copy the value of the handle, without duplicating the underlying resource).

```
class rule_of_three
{
    char* cstring; // raw pointer used as a handle to a dynamically-allocated memory block
 public:
    rule_of_three(const char* arg)
    : cstring(new char[std::strlen(arg)+1]) // allocate
    {
        std::strcpy(cstring, arg); // populate
    }
    ~rule_of_three()
    {
        delete[] cstring;  // deallocate
    }
    rule_of_three(const rule_of_three& other) // copy constructor
    {
        cstring = new char[std::strlen(other.cstring) + 1];
        std::strcpy(cstring, other.cstring);
    }
    rule_of_three& operator=(const rule_of_three& other) // copy assignment
    {
        char* tmp_cstring = new char[std::strlen(other.cstring) + 1];
        std::strcpy(tmp_cstring, other.cstring);
        delete[] cstring;
        cstring = tmp_cstring;
        return *this;
    }
// alternatively, reuse destructor and copy ctor
//  rule_of_three& operator=(rule_of_three other)
//  {
//      std::swap(cstring, other.cstring);
//      return *this;
//  }
};
```

Classes that manage non-copyable resources through copyable handles may have to declare copy assignment and copy constructor private and not provide their definitions or define them as deleted. This is another application of the rule of three: deleting one and leaving the other to be implicitly-defined will most likely result in errors.

Rule of five

Because the presence of a user-defined destructor, copy-constructor, or copy-assignment operator prevents implicit definition of the move constructor and the move assignment operator, any class for which move semantics are desirable, has to declare all five special member functions:

```
class rule_of_five
{
    char* cstring; // raw pointer used as a handle to a dynamically-allocated memory block
 public:
    rule_of_five(const char* arg)
    : cstring(new char[std::strlen(arg)+1]) // allocate
    {
        std::strcpy(cstring, arg); // populate
    }
    ~rule_of_five()
    {
        delete[] cstring;  // deallocate
    }
    rule_of_five(const rule_of_five& other) // copy constructor
    {
        cstring = new char[std::strlen(other.cstring) + 1];
        std::strcpy(cstring, other.cstring);
    }
    rule_of_five(rule_of_five&& other) : cstring(other.cstring) // move constructor
    {
        other.cstring = nullptr;
    }
    rule_of_five& operator=(const rule_of_five& other) // copy assignment
    {
        char* tmp_cstring = new char[std::strlen(other.cstring) + 1];
        std::strcpy(tmp_cstring, other.cstring);
        delete[] cstring;
        cstring = tmp_cstring;
        return *this;
    }
    rule_of_five& operator=(rule_of_five&& other) // move assignment
    {
        if(this!=&other) // prevent self-move
        {
            delete[] cstring;
            cstring = other.cstring;
            other.cstring = nullptr;
        }
        return *this;
    }
// alternatively, replace both assignment operators with
//  rule_of_five& operator=(rule_of_five other)
//  {
//      std::swap(cstring, other.cstring);
//      return *this;
//  }
};
```

Unlike Rule of Three, failing to provide move constructor and move assignment is usually not an error, but a missed optimization opportunity.

Rule of zero

Classes that have custom destructors, copy/move constructors or copy/move assignment operators should deal exclusively with ownership (which follows from the Single Responsibility Principle). Other classes should not have custom destructors, copy/move constructors or copy/move assignment operators.[1]

```
class rule_of_zero
{
    std::string cppstring;
 public:
    rule_of_zero(const std::string& arg) : cppstring(arg) {}
};
```

When a base class is intended for polymorphic use, its destructor may have to be declared public and virtual. This blocks implicit moves (and deprecates implicit copies), and so the special member functions have to be declared as defaulted[2]

```
class base_of_five_defaults
{
 public:
    base_of_five_defaults(const base_of_five_defaults&) = default;
    base_of_five_defaults(base_of_five_defaults&&) = default;
    base_of_five_defaults& operator=(const base_of_five_defaults&) = default;
    base_of_five_defaults& operator=(base_of_five_defaults&&) = default;
    virtual ~base_of_five_defaults() = default;
};
```

however, this can be avoided if the objects of the derived class are not dynamically allocated, or are dynamically allocated only to be stored in a `std::shared_ptr` (such as by `std::make_shared`): shared pointers invoke the derived class destructor even after casting to `std::shared_ptr`<Base>.

References

1. ↑ "Rule of Zero", R. Martinho Fernandes 8/15/2012 (https://rmf.io/cxx11/rule-of-zero)
2. ↑ "A Concern about the Rule of Zero", Scott Meyers, 3/13/2014 (http://scottmeyers.blogspot.fr/2014/03/a-concern-about-rule-of-zero.html)

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/language/rule_of_three&oldid=95257"