rules.sonarsource.com

# C++ static code analysis: The "Rule-of-Zero" should be followed

3-4 minutes

---

Most classes should not directly handle resources, but instead, use members that perform resource handling for them:

- For memory, it can be `std::unique_ptr`, `std::shared_ptr`, `std::vector`…
- For files, it can be `std::ofstream`, `std::ifstream`…
- …

Classes that avoid directly handling resources don’t need to define any of the special member functions required to properly handle resources: destructor, copy constructor, move constructor, copy-assignment operator, move-assignment operator. That’s because the versions of those functions provided by the compiler do the right thing automatically, which is especially useful because writing these functions correctly is typically tricky and error-prone.

Omitting all of these functions from a class is known as the Rule of Zero because no special function should be defined.

In some cases, this rule takes a slightly different shape, while respecting the fact that no definition of those functions will be provided:

- For the base class of a polymorphic hierarchy, the destructor should be declared as `public` and `virtual`, and defaulted (=`default`). The copy-constructor and copy-assignment operator should be deleted. (If you want to copy classes in a polymorphic hierarchy, use the `clone` idiom.) The move operation will be automatically deleted by the compiler.
- For other kinds of base classes, the destructor should be `protected` and non-`virtual`, and defaulted (=`default`).

## Noncompliant Code Example

```
class FooPointer { // Noncompliant. The code is correct (it follows the rule of 5), but unnecessarily complex
  Foo* pFoo;
public:
  FooPointer(int initValue) {
    pFoo = new Foo(initValue);
  }
  ~FooPointer() {
    delete pFoo;
  }
  FooPointer(FooPointer const &fp) = delete;
  FooPointer const & operator=(FooPointer const &fp) = delete;
  FooPointer(FooPointer &&fp) noexcept {
    pFoo = fp.pFoo;
    fp.pFoo = nullptr;
  }
  FooPointer const & operator=(FooPointer &&fp) {
    FooPointer temp(std::move(fp));
    std::swap(temp.pFoo, pFoo);
    return *this;
  }
};
```

## Compliant Solution

```
class FooPointer { // Compliant, std::unique_ptr is use to handle memory management
  unique_ptr<Foo> pFoo;
public:
  FooPointer(int initValue) : pFoo(std::make_unique<Foo>(initValue)
{}
};
```

A polymorphic base class can look like this:

```
class Base { // Compliant, the virtual destructor is defaulted
public:
  virtual ~Base() = default;
  Base(Base const &) = delete;
  Base &operator=(Base const &) = delete;
};
```

## Exceptions

- Empty destructors are treated as though they were defaulted.
- There are several cases when this rule should not be followed. For instance, if your class is manually handling a resource, logging when being constructed/copied, maintaining some kind of counter, having non-transient data that should not be copied (like `capacity` for `std::vector`)… In that case, it should still follow the rule of 5 ({rule:cpp:S3624}). And you should consider if you can isolate this specific behavior in a base class or a dedicated member data, which would allow you to still follow the rule of 0.

## See

- {rule:cpp:S3624}
- {rule:cpp:S1235}