rules.sonarsource.com

C++ static code analysis: When the "Rule-of-Zero" is not applicable, the "Rule-of-Five" should be followed

3-4 minutes

In C++, you should not directly manipulate resources (a database transaction, a network connection, a mutex lock), but encapsulate them in RAII wrapper classes that will allow to manipulate them safely. When defining one of those wrapper classes, you cannot rely on the compiler-generated special member functions to manage the class' resources for you (see the Rule-of-Zero, {rule:cpp:S4963}). You must define those functions yourself to make sure the class' resources are properly copied, moved, and destroyed.

In that case, make sure you consider what should be done for all five special functions (all three of them if your compiler is pre-C++11):

- The destructor, to release the resource when the wrapper is destroyed
- The copy constructor and the copy-assignment operator, to handle what should happen to the resource when the wrapper is copied (a valid option is to disable those operations with `=delete`)
- The move constructor and the move-assignment operator, to handle what should happen to the resource when the wrapper is moved (since C++11). If you cannot find a way to implement them more efficiently than the copy operations, as an exception to this rule, you can just leave out these operations: the compiler will not generate them and will use the copy operations as a fallback.

Those operations work together, and letting the compiler automatically generate some of them, but not all, means that when one of those functions is called, the integrity of the resource will probably be compromised (for instance, it might lead to double release of a resource when the wrapper is copied).

Noncompliant Code Example

```
class FooPointer { // Noncompliant, missing copy constructor and copy-assignment operator
  Foo* pFoo;
public:
  FooPointer(int initValue) {
    pFoo = new Foo(initValue);
  }
  ~FooPointer() {
    delete pFoo;
  }
};

int main() {
  FooPointer a(5);
  FooPointer b = a; // implicit copy constructor gives rise to double free memory error
  return 0;
}
```

Compliant Solution

```
class FooPointer { // Compliant, although it's usually better to reuse an existing wrapper for memory
  Foo* pFoo;
public:
  FooPointer(int initValue) {
    pFoo = new Foo(initValue);
```

```
}
FooPointer(FooPointer& other) {
  pFoo = new Foo(other.pFoo->value);
}
FooPointer& operator=(const FooPointer& other) {
  int val = other.pFoo->value;
  delete pFoo;
  pFoo = new Foo(val);
  return *this;
}
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;
  }
  ~FooPointer() {
    delete pFoo;
  }
};
```

```
int main() {
  FooPointer a(5);
  FooPointer b = a; // no error
  return 0;
}
```

See

- CERT, OOP54-CPP. - Gracefully handle self-copy assignment