C++ Core Guidelines

5 selected rules

C.47, R.13, ES.24, Per.3 & Con.2

C.47: Define and initialize member variables in the order of member declaration

C.47 Define and initialize member variables in the order of member declaration

```
class Foo {
    int m1;
    int m2;
public:
    Foo(int x) :m2{x}, m1{++x} { }   // BAD: misleading initializer order
    // ...
};
```

```
Foo x(1);
```

What is the value of x.m1 and x.m2?

C.47 Define and initialize member variables in the order of member declaration

```
class Foo {
    int m1;
    int m2;
public:
    Foo(int x) :m2{x}, m1{++x} { }   // BAD: misleading initializer order
    // ...
};
```

```
Foo x(1);
```

What is the value of x.m1 and x.m2?

x.m1 == x.m2 == 2, because m1 is declared first and therefore initialized first

R.13: Perform at most one explicit resource allocation in a single expression statement

R.13: Perform at most one explicit resource allocation in a single expression statement

```
// BAD: potential leak
fun(shared_ptr<Widget>(new Widget(a, b)),
        shared_ptr<Widget>(new Widget(c, d)));
```

If you perform two explicit resource allocations in one statement, you could leak resources because **the order of evaluation** of many subexpressions, including function arguments, **is unspecified**.

R.13: Perform at most one explicit resource allocation in a single expression statement

```
// BAD: potential leak
fun(shared_ptr<Widget>(new Widget(a, b)),
       shared_ptr<Widget>(new Widget(c, d)));
```

When **`new`** is used to allocate memory for a C++ class object, the object's constructor is called after the memory is allocated.

R.13: Perform at most one explicit resource allocation in a single expression statement

```
// BAD: potential leak
fun(shared_ptr<Widget>(new Widget(a, b)),
      shared_ptr<Widget>(new Widget(c, d)));
```

The compiler can interleave execution of the two subexpressions.

Memory allocation could be done first for both objects, followed by attempts to call the two Widget constructors

R.13: Perform at most one explicit resource allocation in a single expression statement

```
// BAD: potential leak
fun(shared_ptr<Widget>(new Widget(a, b)),
        shared_ptr<Widget>(new Widget(c, d)));
```

If one of the constructor calls throws an exception, then the other object’s memory will **never be released**!

R.13: Perform at most one explicit resource allocation in a single expression statement

```
fun(make_shared<Widget>(a, b), make_shared<Widget>(c, d));
```

The best solution is to avoid explicit allocation entirely use **factory functions** that return owning objects

`make_shared` constructs an object of type T and wraps it in a std::shared_ptr

ES.24: Use a unique_ptr<T> to hold pointers

ES.24: Use a unique_ptr<T> to hold pointers

- Easy way to avoid leaks - pointer is destroyed when leaving the scope
- Increases ownership safety - for the same reason as above
- increases readability - no lines of pointer destruction needed

ES.24: Use a unique_ptr<T> to hold pointers

```
void fun(){
     auto p1 = make_unique<MyObject>(new MyObject());
     MyObject* p2 = new MyObject();

     if(booleanValue)              //if booleanValue==true p1 gets destroyed, but p2
          return;                  //creates a leak
     //or throwing an Exception

     delete p2;
     //...
}
```

Per.3:
Don’t optimize something that’s not performance critical

Per.3: Don’t optimize something that’s not performance critical

Reason: Optimizing a non-performance-critical part of a program has no effect on system performance.

If your program spends 4% of its processing time doing computation A and 40% of its time doing computation B, a 50% improvement on A is only as impactful as a 5% improvement on B.

Con.2: By default, make member functions const

Con.2: By default, make member functions const

Mark member function as const unless it changes the object’s observable state

- Design intent is clearer
- Errors caught at compile time
- Better readability

Con.2: By default, make member functions const

Compiler error:

```
class MyObject {
     int value;
public:
     int getValue() {return value;}     //compiler thinks this function alters the
}                                       //object's state (const is missing)

void fun(const MyObject& o) {
     int v = o.getValue();              //compiler error: o can’t be changed; compiler
}                                       //thinks getValue() will change o
```