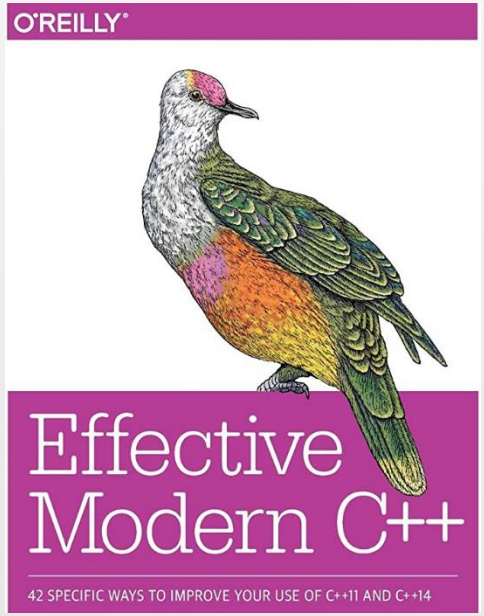
ITEM 17:

Understand special member function generation

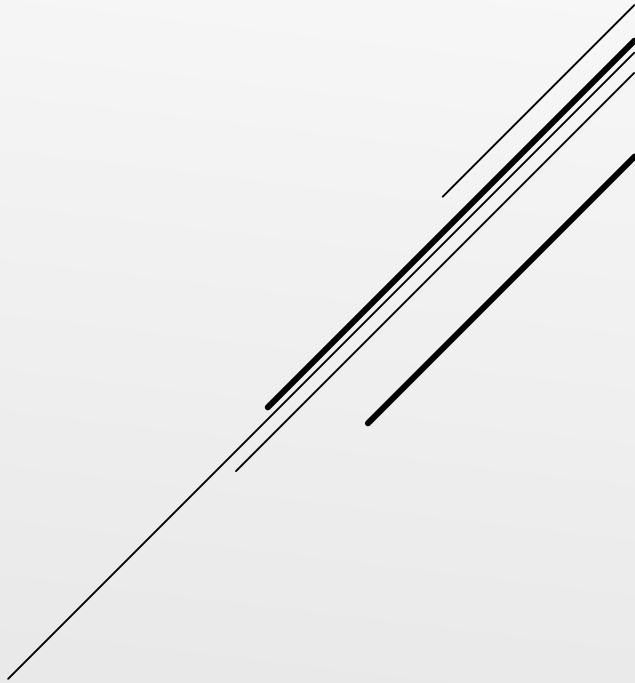
Scott Meyers - Effective Modern C++



Presented By
@7eRoM

- **Default Constructor**
  - If a class does not have any constructor defined, the compiler generates a default constructor
  - It initializes the object's member variables to their default values.
  - The default constructor is invoked when an object is created without any arguments.

- **Destructor**
  - If the programmer does not define a destructor, the compiler generates a default destructor.

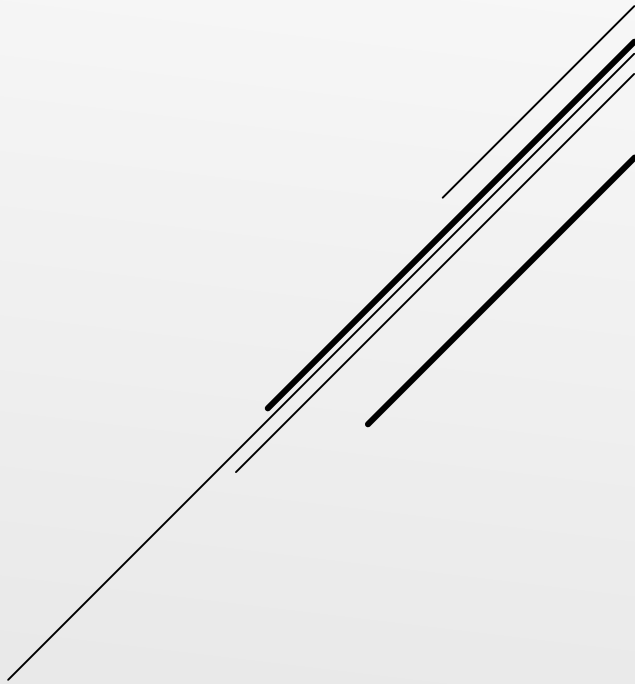


```cpp
class MyClass {
private:
    int* data; // Pointer to dynamically allocated memory

public:
    // Constructor
    MyClass() {
        std::cout << "Default constructor called." << std::endl;
        data = new int(0);
    }

    // Parameterized constructor
    MyClass(int value) {
        std::cout << "Parameterized constructor called." << std::endl;
        data = new int(value);
    }

    // Destructor
    ~MyClass() {
        std::cout << "Destructor called." << std::endl;
        delete data;
    }

    void printData() const {
        std::cout << "Data: " << *data << std::endl;
    }
};
```

```cpp
int main() {
    MyClass obj1(42); // Parameterized constructor called

    return 0;
}
```

- **Copy Constructor**
  - The copy constructor creates a new object by making a copy of an existing object.
  - It is called when an object is **initialized** with another object of the same class.

- **Copy Assignment Operator**
  - The copy assignment operator is used to assign the values of one object to another object of the same class.
  - It is invoked when the **assignment operator (=)** is used between two objects.

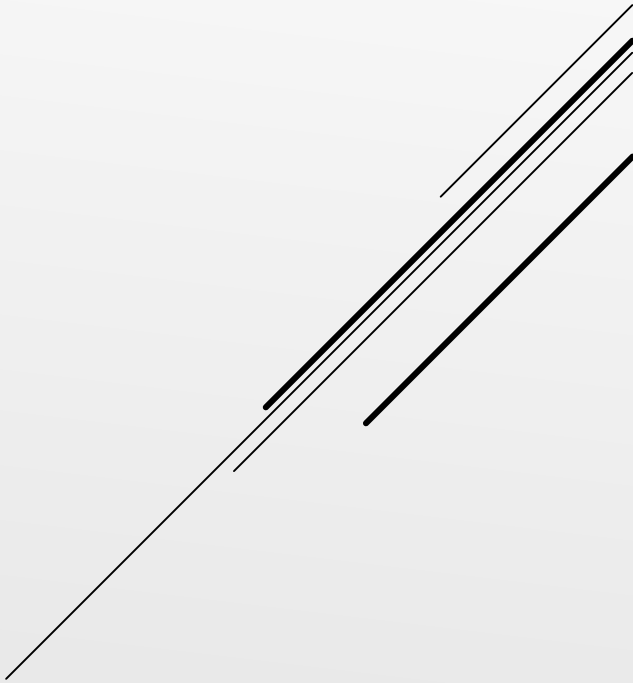


```cpp
// Copy constructor
MyClass(const MyClass& other) {
    std::cout << "Copy constructor called." << std::endl;
    data = new int(*other.data);
}

// Copy assignment operator
MyClass& operator=(const MyClass& other) {
    std::cout << "Copy assignment operator called." << std::endl;
    if (this != &other) {
        delete data;
        data = new int(*other.data);
    }
    return *this;
}
```

```cpp
MyClass obj2 = obj1; // Copy constructor called
obj2.printData();

MyClass obj3;
obj3 = obj1; // Copy assignment operator called
obj3.printData();
```

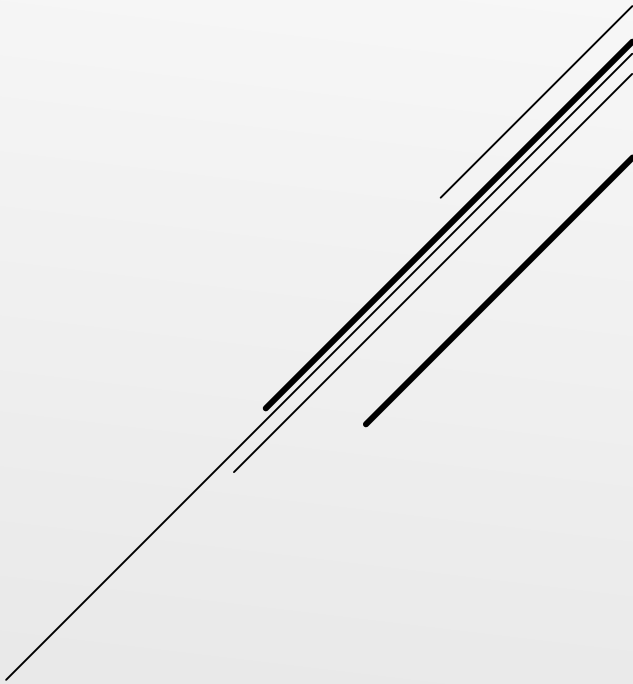
- **Move Constructor**
  - Introduced in C++11, the move constructor is used to efficiently transfer the resources (such as dynamically allocated memory) from one object to another.
  - It is invoked when an object is **initialized** with an rvalue (such as a temporary object).

- **Move Assignment Operator**
  - Introduced in C++11, the move assignment operator is used to efficiently transfer the resources from one object to another object of the same class.
  - It is invoked when the move **assignment operator (=)** is used between two objects.



```cpp
// Move constructor
MyClass(MyClass&& other) noexcept {
    std::cout << "Move constructor called." << std::endl;
    data = other.data;
    other.data = nullptr;
}

// Move assignment operator
MyClass& operator=(MyClass&& other) noexcept {
    std::cout << "Move assignment operator called." << std::endl;
    if (this != &other) {
        delete data;
        data = other.data;
        other.data = nullptr;
    }
    return *this;
}
```

```cpp
MyClass obj4 = std::move(obj1); // Move constructor called
obj4.printData();

MyClass obj5;
obj5 = std::move(obj2); // Move assignment operator called
obj5.printData();
```



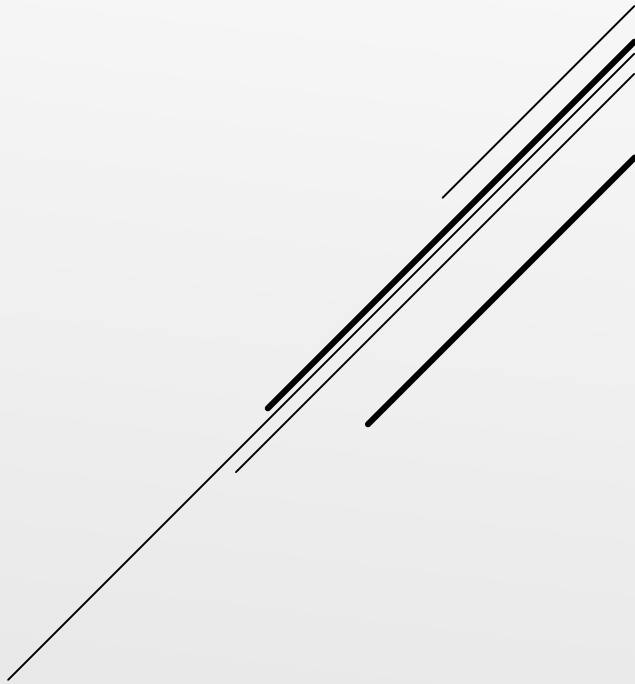
- **member-wise** copy refers to the process of copying the values of each member variable from one object to another object of the same class.
- This copy is typically performed using the assignment operator (=) or the copy constructor of the member variable's type.

```cpp
class MyClass {
private:
    int value;

public:
    MyClass(int value) : value(value) {}

    void printValue() const {
        std::cout << "Value: " << value << std::endl;
    }
};
```

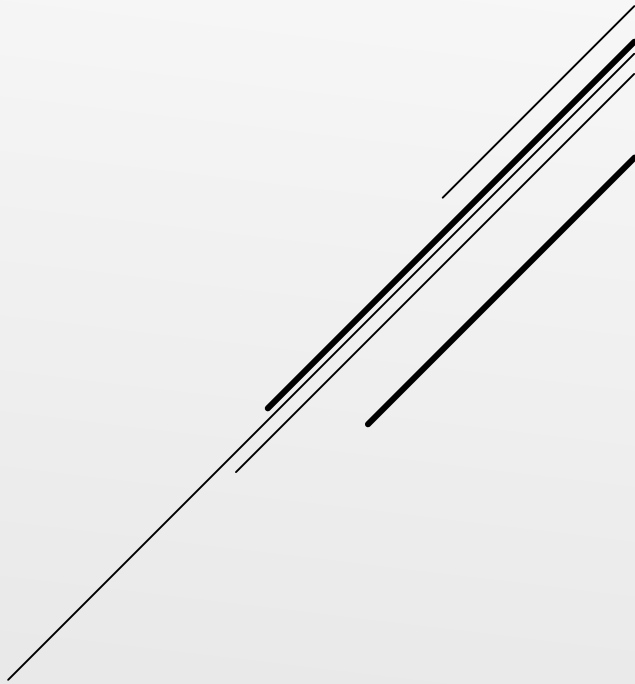
```cpp
int main() {
    MyClass obj1(42);
    obj1.printValue(); // Output: Value: 42

    MyClass obj2 = obj1; // Member-wise copy using copy constructor
    obj2.printValue(); // Output: Value: 42

    MyClass obj3(100);
    obj3 = obj1; // Member-wise copy using copy assignment operator
    obj3.printValue(); // Output: Value: 42

    return 0;
}
```

- C++98 has four special member functions:
  - the default constructor
  - the destructor
  - the copy constructor
  - the copy assignment operator
- As of C++11, the special member functions club has two more inductees:
  - the move constructor
  - the move assignment operator

- **memberwise moves** on the non-static data members of the class.
- "Memberwise moves" are, in reality, more like memberwise move **requests**, because types that aren't move-enabled (e.g., most C++98 legacy classes)
- a memberwise move consists of move operations on data members and base classes that support move operations, but a copy operation for those that don't.

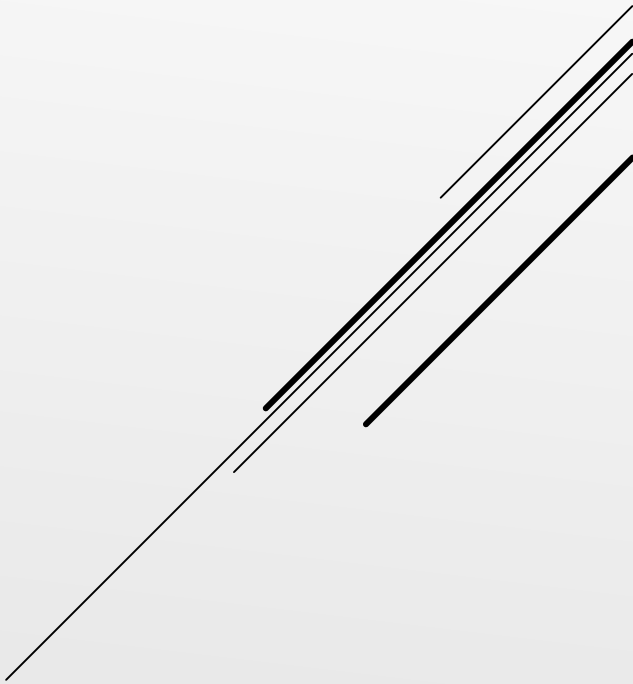


- As is the case with the copy operations, the move operations aren't generated if you declare them yourself.
- The two copy operations are independent: declaring one doesn't prevent compilers from generating the other.
  - if you declare a copy constructor, but no copy assignment operator, then write code that requires copy assignment, compilers will generate the copy assignment operator for you.
  - if you declare a copy assignment operator, but no copy constructor, yet your code requires copy construction, compilers will generate the copy constructor for you.
- The two move operations are not independent: If you declare either, that prevents compilers from generating the other.
  - The rationale is that if you declare, say, a move constructor for your class, you're indicating that there's something about how move construction should be implemented that's different from the default memberwise move that compilers would generate.
  - And if there's something wrong with memberwise move construction, there'd probably be something wrong with memberwise move assignment, too.



- move operations won't be generated for any class that explicitly declares a copy operation.
  - declaring a copy operation indicates that the normal approach to copying an object (memberwise copy) isn't appropriate for the class, and compilers figure that if memberwise copy isn't appropriate for the copy operations, memberwise move probably isn't appropriate for the move operations.

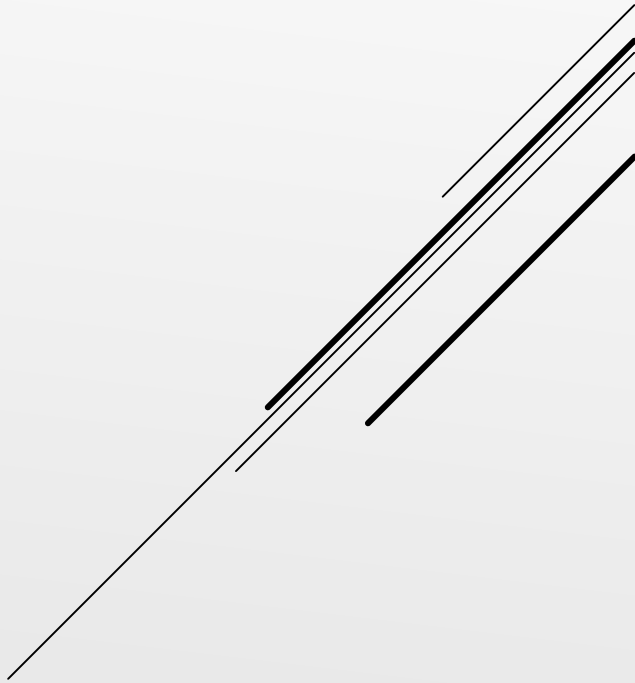
- Declaring a move operation in a class causes compilers to disable the copy operations (by deleting them).
  - if memberwise move isn't the proper way to move an object, there's no reason to expect that memberwise copy is the proper way to copy it.



- **Rule of Three:** if you declare any of a copy constructor, copy assignment operator, or destructor, you should declare all three.
  - whatever resource management was being done in one copy operation probably needed to be done in the other copy operation
  - the class destructor would also be participating in management of the resource (usually releasing it)

- A consequence of the Rule of Three is that the presence of a user-declared destructor indicates that simple memberwise copy is unlikely to be appropriate for the copying operations in the class.

- if a class declares a destructor, the copy operations probably shouldn't be automatically generated, because they wouldn't do the right thing.

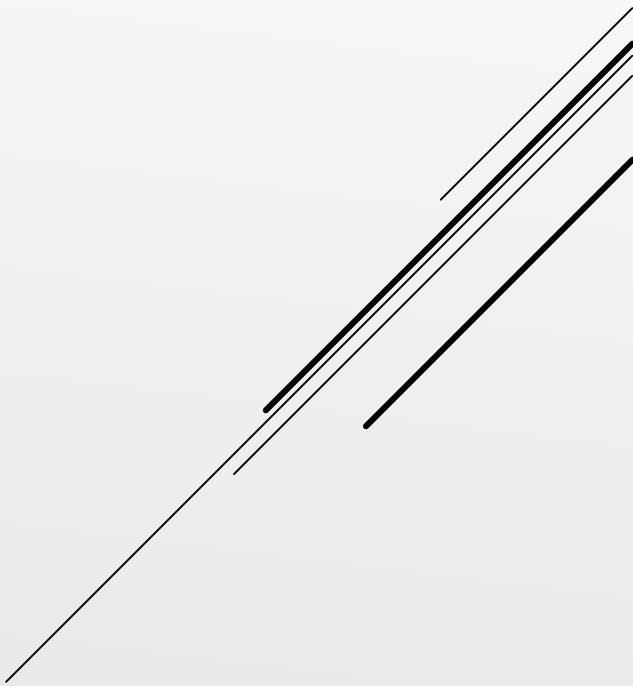


- move operations are generated for classes (when needed) only if these three things are true:
  - No copy operations are declared in the class.
  - No move operations are declared in the class.
  - No destructor is declared in the class.
- if memberwise copying of the class's non-static data members is what you want), your job is easy, because C++11's "= default" lets you say that explicitly

```
class Widget {
public:
  …
  ~Widget();                                  // user-declared dtor

  …                                           // default copy ctor
  Widget(const Widget&) = default;            // behavior is OK

  Widget&                                     // default copy assign
    operator=(const Widget&) = default;       // behavior is OK
  …
};
```

- This approach is often useful in polymorphic base classes, i.e., classes defining interfaces through which derived class objects are manipulated.

```
class Base {
public:
  virtual ~Base() = default;                    // make dtor virtual

  Base(Base&&) = default;                       // support moving
  Base& operator=(Base&&) = default;

  Base(const Base&) = default;                  // support copying
  Base& operator=(const Base&) = default;

  …

};
```



- Explicitly declaring and using "= default" for copy and move operations, even when compilers can generate them correctly, provides clearer intentions and helps avoid subtle bugs. As an example:

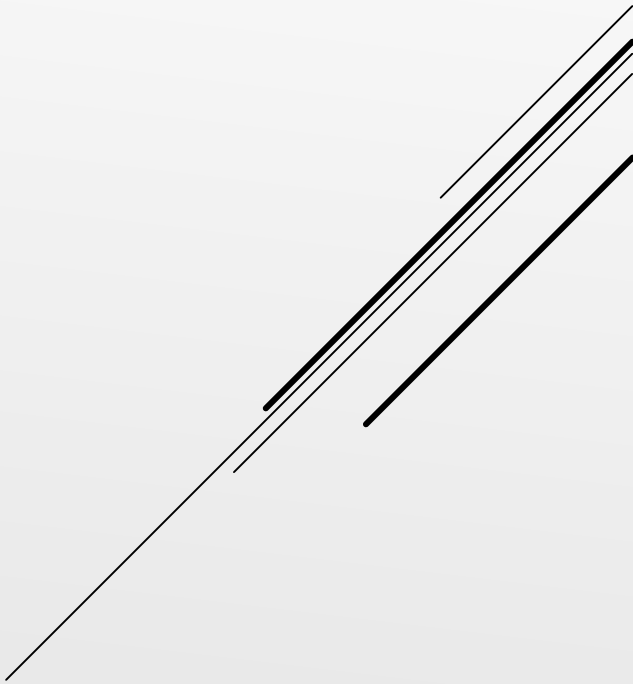
```
class StringTable {
public:
  StringTable() {}
  …                  // functions for insertion, erasure, lookup,
                     // etc., but no copy/move/dtor functionality

private:
  std::map<int, std::string> values;
};
```

- sometime later, it's decided that logging the default construction and the destruction:

```
class StringTable {
public:
  StringTable()
  { makeLogEntry("Creating StringTable object"); }     // added

  ~StringTable()                                       // also
  { makeLogEntry("Destroying StringTable object"); }   // added

  …                                   // other funcs as before
private:
  std::map<int, std::string> values;    // as before
};
```

- This looks reasonable, but declaring a destructor has a potentially significant side effect:
  - it prevents the move operations from being generated.
  - requests to move cause copies to be made.
  - copying a std::map<int, std::string> is likely to be orders of magnitude slower than moving it.

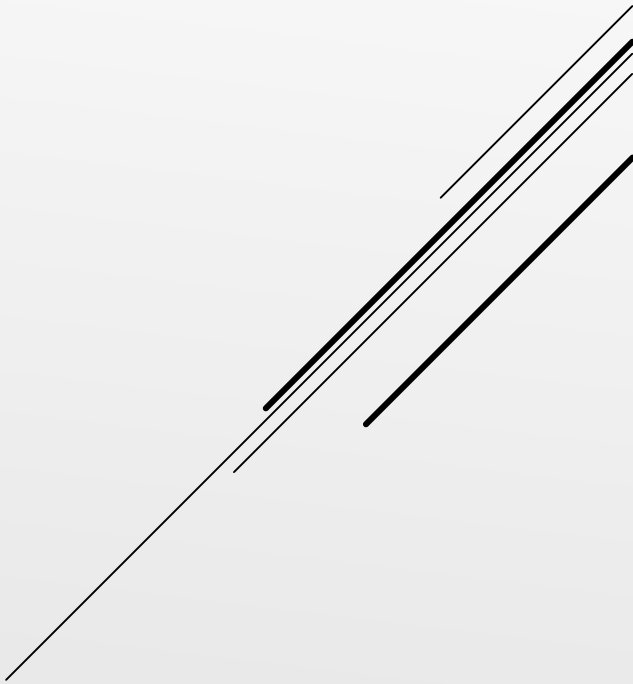
- Explicitly defining copy and move operations using "= default" would have prevented this issue from occurring.



- The C++11 rules governing the special member functions are thus:

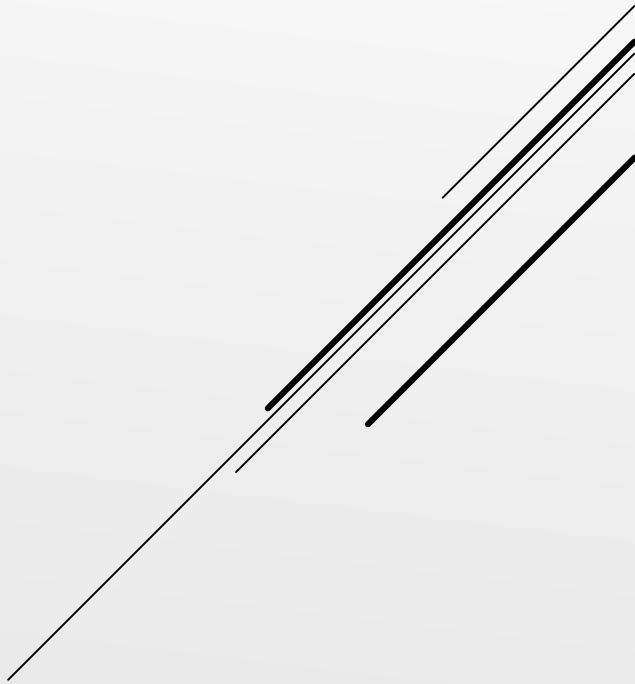
- **Default constructor**
  - Same rules as C++98. Generated only if the class contains no user-declared constructors.

- **Destructor**
  - Essentially same rules as C++98; sole difference is that destructors are noexcept by default.
  - As in C++98, virtual only if a base class destructor is virtual.

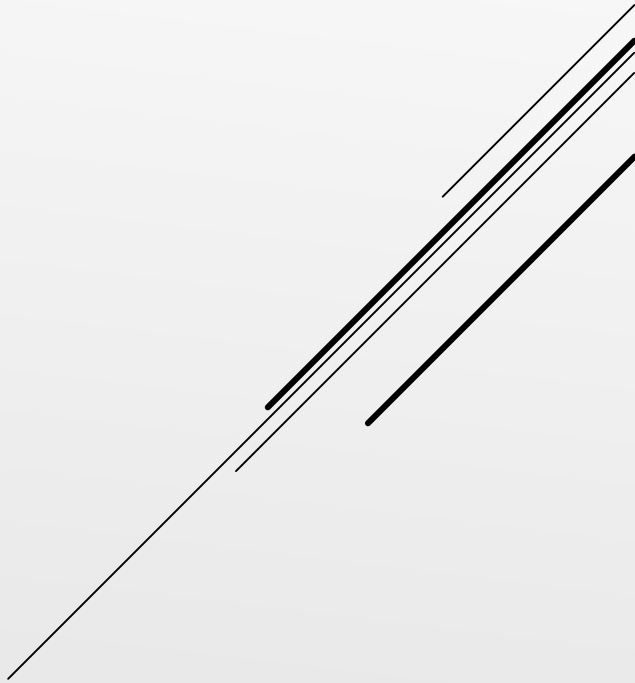


- **Copy constructor**
  - Same runtime behavior as C++98: memberwise copy construction of non-static data members.
  - Generated only if the class lacks a user-declared copy constructor.
  - Deleted if the class declares a move operation.
  - Generation of this function in a class with a user-declared copy assignment operator or destructor is deprecated.

- **Copy assignment operator**
  - Same runtime behavior as C++98: memberwise copy assignment of non-static data members.
  - Generated only if the class lacks a user-declared copy assignment operator.
  - Deleted if the class declares a move operation.
  - Generation of this function in a class with a user-declared copy constructor or destructor is deprecated.



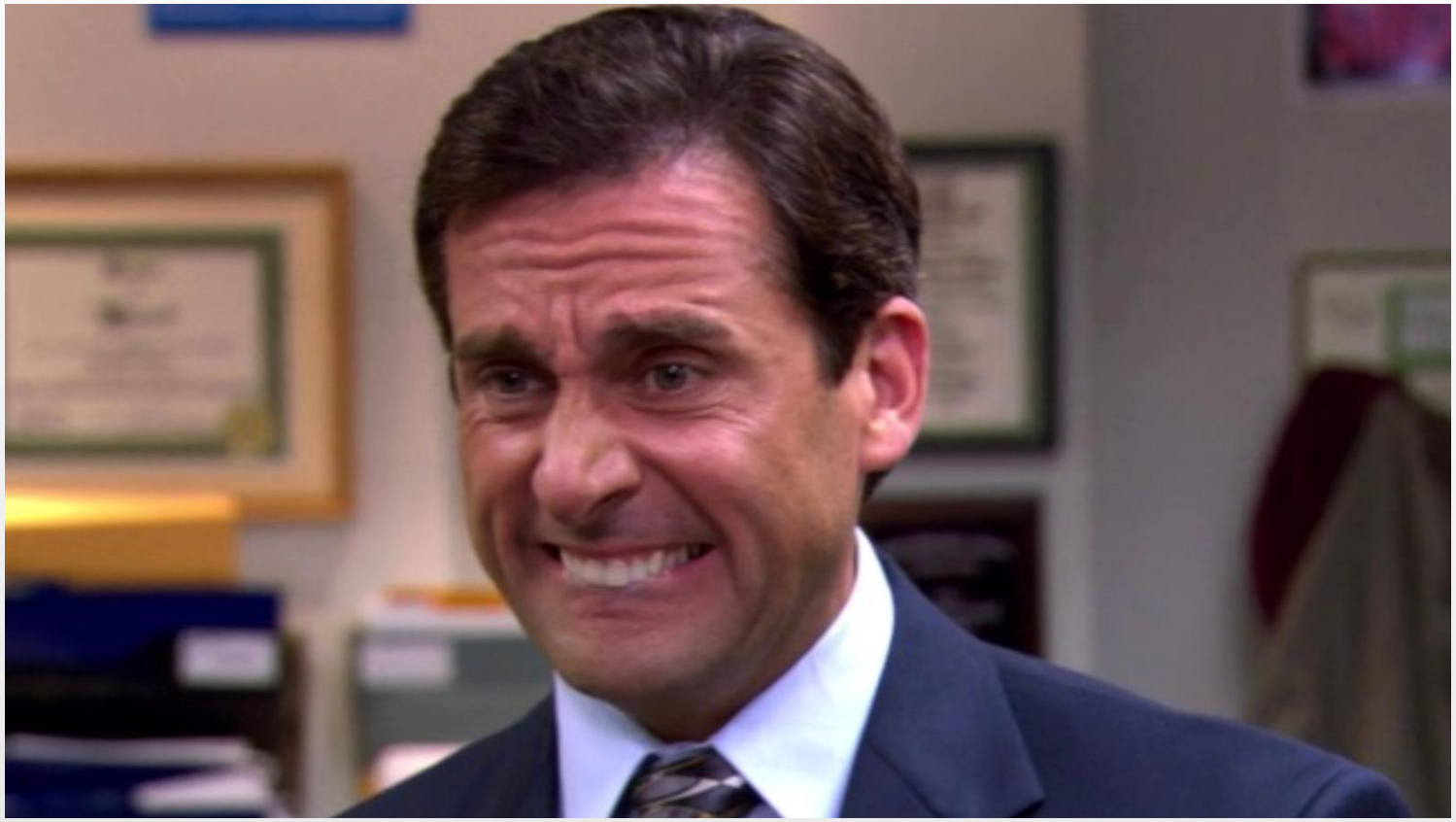
- **Move constructor** and **move assignment operator**
  - Each performs memberwise moving of non-static data members.
  - Generated only if the class contains no user-declared copy operations, move operations, or destructor.



- Note that there's nothing in the rules about the existence of a member function template preventing compilers from generating the special member functions

```
class Widget {
  …
  template<typename T>                    // construct Widget
  Widget(const T& rhs);                   // from anything

  template<typename T>                    // assign Widget
  Widget& operator=(const T& rhs);        // from anything
  …
};
```

- Compilers will still generate copy and move operations for Widget, even though these templates could be instantiated to produce the signatures for the copy constructor and copy assignment operator (That would be the case when T is Widget.)
- It can have significant consequences (Item 26)



I’m cool as a cucumber