The Rule of Seven

(plus or minus two)

or,

The Canonical C++ Class

Hello world!

```
class Cat {
    int legs;
};

Cat c;
```

Slightly more complicated

```
class Cat {
    int legs;
    std::vector<int> toes;
};

Cat c;
```

The Rule of Zero

If your class isn't doing anything too fancy, you don't need to define any special member functions.

The Rule of Zero

If your class isn't doing anything too fancy, you ~~don't need to~~ **shouldn't** define any special member functions.

These are the special member functions

- Copy constructor
- Move constructor
- Copy assignment operator
- Move assignment operator
- Destructor

NOT special member functions

- Default constructor
- Other constructors
- Other assignment operators
- Member swap()
- Non-member swap()

NOT special member functions

- Default constructor
- Other constructors
- Other assignment operators
- Member swap()
- Non-member swap()

We'll talk about these too, but not quite yet.

The Rule of Zero: success!

```
class Cat {
    int legs;
    std::vector<int> toes;
};

Cat c;
```

Let’s do something fancier

```
class Cat {
    int legs;
    std::shared_ptr<Owner> owner;

    Cat(std::shared_ptr<Owner> o) :
      legs(4), owner(std::move(o)) {}
};

auto optr = std::make_shared<Owner>();
Cat c(optr);
```

Rule of Zero: success?

```
class Cat {
    int legs;
    std::shared_ptr<Owner> owner;

    Cat(std::shared_ptr<Owner> o) :
      legs(4), owner(std::move(o)) {}
};

auto optr = std::make_shared<Owner>();
Cat c(optr);
```

Rule of Zero: success!

As long as you want the default behavior.

Copy-constructing a `Cat` will cause the two `Cats` to share one `Owner`, because that's what it means to copy-construct a `shared_ptr`.

The Rule of Zero

... we have arrived at the Rule of Zero (which is actually a particular instance of the *Single Responsibility Principle*):

> *Classes that have custom destructors, copy/move constructors or copy/move assignment operators should deal exclusively with ownership. Other classes should not have custom destructors, copy/move constructors or copy/move assignment operators.*

R. Martinho Fernandes, 15 August 2012
http://flamingdangerzone.com/cxx11/2012/08/15/rule-of-zero.html

Okay, fine. Let's get real.

```
class Dog {
    size_t len;
    void *data;

    Dog(size_t L) :
        len(L), data(malloc(len)) {}
};
```

C++03 style: The Rule of Three

```
class Dog {
    size_t len;
    void *data;
public:
    Dog(size_t L) : len(L), data(malloc(len)) {}
    Dog(const Dog& rhs) : len(rhs.len), data(malloc(len)) {
        memcpy(data, rhs.data, len);
    }
    Dog& operator=(const Dog& rhs) {
        len = rhs.len;
        data = realloc(data, len);
        memmove(data, rhs.data, len);
        return *this;
    }
    ~Dog() { free(data); }
};
```

C++11 style: The Rule of Five

```
class Dog {
    size_t len;
    void *data;
public:
    Dog(size_t L) : len(L), data(malloc(len)) {}
    Dog(const Dog& rhs) : len(rhs.len), data(malloc(len)) {
        memcpy(data, rhs.data, len);
    }
    Dog(Dog&& rhs) : len(rhs.len), data(rhs.data) {
        rhs.data = nullptr; rhs.len = 0;
    }
    Dog& operator=(const Dog& rhs) {
        len = rhs.len;
        data = realloc(data, len);
        memmove(data, rhs.data, len);
        return *this;
    }
    Dog& operator=(Dog&& rhs) {
        if (&rhs != this) {
            free(data); data = rhs.data; rhs.data = nullptr;
            len = rhs.len; rhs.len = 0;
        }
        return *this;
    }
    ~Dog() { free(data); }
};
```

C++11 style: The Rule of Five

```
class Dog {
    size_t len;
    void *data;
public:
    Dog(size_t L) : len(L), data(malloc(len)) {}
    Dog(const Dog& rhs) : len(rhs.len), data(malloc(len)) {
        memcpy(data, rhs.data, len);
    }
    Dog(Dog&& rhs) : len(rhs.len), data(rhs.data) {
        rhs.data = nullptr; rhs.len = 0;
    }
    Dog& operator=(const Dog& rhs) {
        len = rhs.len;
        data = realloc(data, len);
        memmove(data, rhs.data, len);
        return *this;
    }
    Dog& operator=(Dog&& rhs) {
        if (&rhs != this) {
            free(data); data = rhs.data; rhs.data = nullptr;
            len = rhs.len; rhs.len = 0;
        }
        return *this;
    }
    ~Dog() { free(data); }
};
```

This is so much code!

“Unified” assignment operator idiom

```
class Dog {
    size_t len;
    void *data;
public:
    Dog(size_t L) : len(L), data(malloc(len)) {}
    Dog(const Dog& rhs) : len(rhs.len), data(malloc(len)) {
        memcpy(data, rhs.data, len);
    }
    Dog(Dog&& rhs) : len(rhs.len), data(rhs.data) {
        rhs.data = nullptr;
        rhs.len = 0;
    }
    Dog& operator=(Dog rhs) {
        std::swap(data, rhs.data);
        std::swap(len, rhs.len);
        return *this;
    }
    ~Dog() { free(data); }
};
```

```
fido = rex;  // copies rex into rhs,
             // swaps, destroys

fido = std::move(rex);  // moves rex into rhs,
                        // swaps, destroys

fido = (std::move?) fido;  // works fine
```

“Unified” assignment operator idiom

```
class Dog {
    size_t len;
    void *data;
public:
    Dog(size_t L) : len(L), data(malloc(len)) {}
    Dog(const Dog& rhs) : len(rhs.len), data(malloc(len)) {
        memcpy(data, rhs.data, len);
    }
    Dog(Dog&& rhs) : len(rhs.len), data(rhs.data) {
        rhs.data = nullptr;
        rhs.len = 0;
    }
    Dog& operator=(Dog rhs) {
        std::swap(data, rhs.data);
        std::swap(len, rhs.len);
        return *this;
    }
    ~Dog() { free(data); }
};
```

Semantic correctness, but at a cost:

- Self-copy / self-move are expensive.
- Copy can’t exploit realloc().
- One extra temporary object is destroyed.

Speaking of self-assignment...

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=59603

```
/* Shuffle the elements of an array.  */
template<typename T> void shuffle(T *array, int length)
{
    using std::swap;  // enable ADL

    if (length == 0) return;

    for (int i = 1; i < length; ++i) {
        int j = randint(0, i);  // a number in the range [0,i] inclusive
        swap(array[i], array[j]);
    }
}
```

Speaking of self-assignment...

```
template<typename T> void shuffle(T *array, int length)
{
    using std::swap;  // enable ADL

    if (length == 0) return;

    for (int i = 1; i < length; ++i) {
        int j = randint(0, i);  // a number in the range [0,i] inclusive
        swap(array[i], array[j]);
    }
}

std::vector<std::vector<int>> a = { {1,2,3}, {4,5,6} };
shuffle(a.data(), a.size());
```

Speaking of self-assignment...

```
template<typename T> void shuffle(T *array, int length)
{
    using std::swap;  // enable ADL

    if (length == 0) return;

    for (int i = 1; i < length; ++i) {
        int j = randint(0, i);  // a number in the range [0,i] inclusive
        swap(array[i], array[j]);
    }
}                             If i == 1 and j == 1, we swap a[1] with itself.

std::vector<std::vector<int>> a = { {1,2,3}, {4,5,6} };
shuffle(a.data(), a.size());
```

`vector::operator=(vector&&)`

17.6.4.9 Function arguments **[res.on.arguments]**

Each of the following applies to **all** arguments to functions **defined in the C++ standard library**, unless explicitly stated otherwise.

> — If a function argument binds to an rvalue reference parameter, the implementation may assume that this parameter is a **unique reference** to this argument. [*Note:* If a program casts an lvalue to an xvalue while passing that lvalue to a library function (e.g. by calling the function with the argument move(x)), the program is effectively asking that function to treat that lvalue as a temporary. **The implementation is free to optimize away aliasing checks** which might be needed if the argument was an lvalue. —*end note*]

http://stackoverflow.com/questions/13129031/on-implementing-stdswap-in-terms-of-move-assignment-and-move-constructor

In other words,
std::vector does not properly handle self-move-assignment!

Nor do std::string, std::shared_ptr, etc.

Consider a Cat class with a std::string member.

In order for our Cat class to properly handle self-move-assignment, we must provide our own definition for the move assignment operator.

The default one will ***not*** do what we want.

“Oh, but I don’t use self-move.”

Really? What about std::swap?

```
template<class T> void swap(T& a, T& b)
{
    T temp = std::move(a);
    a = std::move(b);
    b = std::move(temp);
}
```

```
Calls a.operator=(std::move(b))

If T is a standard library type, then
T::operator=(T&&) is a function
defined in the standard library,
taking an rvalue reference parameter.

If &a == &b, this invokes undefined
behavior.
```

“Oh, but I provide an ADL swap().”

```
class Cat {
    int legs;
    std::vector<int> toes;
};

void swap(Cat& a, Cat& b) {
    if (&a == &b) return;
    std::swap(a,b);
}

Cat c;
swap(c,c);  // ha, it’s foolproof!
c = std::move(c);  // “Oh, I would never write that.”
```

...until, one day...

```
class BagOfCats {
    Cat one;
    Cat two;
};

// BagOfCats' implementor omits to write an ADL swap()

BagOfCats bag;
swap(bag,bag);  // ADL finds std::swap<BagOfCats>
                // and everything blows up
```

The Rule of At Least One

If your class has any members of std:: types, you'll need a custom move-assignment operator.

The Rule of At Least ~~One~~ Two

If your class has any members of std:: types, you'll need a custom move-assignment operator.

- So you'll need a move constructor (because you won't get one by default anymore).

The Rule of At Least ~~One~~ ~~Two~~ 4

If your class has any members of std:: types, you'll need a custom move-assignment operator.

So you'll need a move constructor (because you won't get one by default anymore).

So you'll need a copy constructor and copy assignment operator (because now you won't get them either).

Picking up again with C++11...

```
class Dog {
    size_t len;
    void *data;
public:
    Dog(size_t L) : len(L), data(malloc(len)) {}
    Dog(const Dog& rhs) : len(rhs.len), data(malloc(len)) {
        memcpy(data, rhs.data, len);
    }
    Dog(Dog&& rhs) : len(rhs.len), data(rhs.data) {
        rhs.data = nullptr; rhs.len = 0;
    }
    Dog& operator=(const Dog& rhs) {
        len = rhs.len;
        data = realloc(data, len);
        memmove(data, rhs.data, len);
        return *this;
    }
    Dog& operator=(Dog&& rhs) {
        if (&rhs != this) {
            free(data); data = rhs.data; rhs.data = nullptr;
            len = rhs.len; rhs.len = 0;
        }
        return *this;
    }
    ~Dog() { free(data); }
};
```

The copy-and-swap idiom

```
class Dog {
    size_t len;
    void *data;
public:
    Dog() : len(0), data(nullptr) {}                              // default constructor is required
    Dog(size_t L) : len(L), data(malloc(len)) {}
    Dog(const Dog& rhs) : len(rhs.len), data(malloc(len)) {       // someone needs to know how to copy a Dog
        memcpy(data, rhs.data, len);
    }
    Dog(Dog&& rhs) : Dog() { this->swap(rhs); }                      // move-construction is efficient
    Dog& operator=(const Dog& rhs) { return this->swap(Dog(rhs)); }  // perhaps inefficient, but semantically
correct
    Dog& operator=(Dog&& rhs) { return this->swap(rhs); }            // move-assignment is efficient
    ~Dog() { free(data); }

    Dog& swap(Dog& rhs) {                                         // swap mustn't throw exceptions
        if (this != &rhs) {
            using std::swap;
            swap(len, rhs.len);
            swap(data, rhs.data);
        }
        return *this;
    }
};
```

Use copy-and-swap

It's not efficient for copy-assignment,
but it's **guaranteed correct**,
it's easy to remember,
and you can always optimize
the copy-assignment operator
later
if it's really a bottleneck.

The copy-and-swap idiom

```
class Dog {
public:
    Dog() {  ... }
    Dog(const Dog&) { ... }
    ~Dog() { ... }

    Dog(Dog&& rhs) : Dog() { swap(rhs); }
    Dog& operator=(const Dog& rhs) { return swap(Dog(rhs)); }
    Dog& operator=(Dog&& rhs) { return swap(rhs); }

    Dog& swap(Dog& rhs) {
        if (this != &rhs) {
            using std::swap;  // enable ADL
            swap(every member);
        }
        return *this;
    }
};
```