Special Member Function

CS 106L, Fall '21

Controlling fundamental behaviors of your classes for efficiency and memory.

CS 106B covers the barebones of C++ classes!

we'll be covering the rest.

template classes • const correctness • operator overloading • move semantics • special member functions • RAII

Key questions we'll answer today

- What are special member functions? When are they called?
- When should we declare a special member function?
- When should we not declare a special member function?
- What are modern standards for SMFs? How do they differ from classic, C++98 standards?

Today's agenda

- The 6 Special Member Functions
- Copy Constructors, Copy Assignment Operators
- Default and Delete
- Rules of 3 and 0
- Move Constructors and Move Assignment Operators
- Moving away from C++98 to C++11: Rule of 5

Special Member Functions (SMFs)

- These functions are generated only when they're called (and before any are explicitly defined by you):
  - Default Constructor
  - Copy Constructor
  - Copy Assignment Operator
  - Destructor
  - Move Constructor
  - Move Assignment Operator

Special Member Functions (SMFs)

- These functions are generated only when they're called (and before any are explicitly defined by you):

```
class Widget {
  public:
    Widget();                                   // default constructor
    Widget (const Widget& w);                   // copy constructor
    Widget& operator = (const Widget& w);       // copy assignment operator
    ~Widget();                                  // destructor
    Widget (Widget&& rhs);                      // move constructor
    Widget& operator = (Widget&& rhs);          // move assignment operator
}
```

Special Member Functions (SMFs)

- These functions are generated only when they're called (and before any are explicitly defined by you):

```
class Widget {
  public:
    Widget();                                     // default constructor
    Widget (const Widget& w);
    Widget& operator = (const Widget& w);
    ~Widget();
    Widget (Widget&& rhs);
    Widget& operator = (Widget&& rhs);
}
```

- object is created with no parameters
- constructor also has no parameters
- all SMFs are **`public`** and **`inline`** function, meaning that wherever it's used is replaced with the generated code in the function

Special Member Functions (SMFs)

- These functions are generated only when they're called (and before any are explicitly defined by you):

```
class Widget {
  public:
    Widget();                                  // default constructor
    Widget (const Widget& w);                  // copy constructor
    Widget& operator = (const Widget& w);
    ~Widget();
    Widget (Widget&& rhs);
    Widget& operator = (Widget&& rhs);
}
```

- another type of constructor that creates an instance of a class
- constructs a member-wise copy of an object (deep copy)

Special Member Functions (SMFs)

- These functions are generated only when they're called (and before any are explicitly defined by you):

```
class Widget {
  public:
    Widget();                                 // default constructor
    Widget (const Widget& w);                 // copy constructor
    Widget& operator = (const Widget& w);     // copy assignment operator
    ~Widget();                                // destructor
    Widget (Widget&& rhs);
    Widget& operator = (Widget&& rhs);
}
```

- very similar to copy constructor, except called when trying to set one object equal to another e.g. `w1 = w2;`

Special Member Functions (SMFs)

- These functions are generated only when they're called (and before any are explicitly defined by you):

```
class Widget {
  public:
    Widget();                                  // default constructor
    Widget (const Widget& w);                  // copy constructor
    Widget& operator = (const Widget& w);      // copy assignment operator
    ~Widget();                                 // destructor
    Widget (Widget&& rhs);
    Widget& operator = (Widget&& rhs);
}
```

- called whenever object goes out of scope
- can be used for deallocating member variables and avoiding memory leaks

Special Member Functions (SMFs)

- These functions are generated only when they're called (and before any are explicitly defined by you):

```
class Widget {
  public:
    Widget();                                  // default constructor
    Widget (const Widget& w);                  // copy constructor
    Widget& operator = (const Widget& w);      /                  operator
    ~Widget();                                 // destructor
    Widget (Widget&& rhs);                     // move constructor
    Widget& operator = (Widget&& rhs);         // move assignment operator
}
```

we'll come back to these!

Special Member Functions (SMFs)

- Let's recap about the four SMFs we've learned about so far!
- Default Constructor
  - Object created with no parameters, no member variables instantiated
- Copy Constructor
  - Object created as a copy of existing object (member variable-wise)
- Copy Assignment Operator
  - Existing object replaced as a copy of another existing object.
- Destructor
  - Object destroyed when it is out of scope.

Which SMF is called on each line?

```
using std::vector;
vector<int> func(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2};
  vector<int> vec{};
  vector<int> vec{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- Default construction: object created with no parameters.
- Copy construction: object treated as copy of existing object.
- Copy assignment: existing object replaced as a copy of another existing object.
- Destruction: object destroyed when it's out of scope.

Copy constructor (passing by value)

```
using std::vector;
vector<int> func(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2};
  vector<int> vec{};
  vector<int> vec{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- Default construction: object created with no parameters.
- Copy construction: object treated as copy of existing object.
- Copy assignment: existing object replaced as a copy of another existing object.
- Destruction: object destroyed when it's out of scope.

Default constructor creates empty vector

```
using std::vector;
vector<int> func(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2};
  vector<int> vec{};
  vector<int> vec{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- Default construction: object created with no parameters.
- Copy construction: object treated as copy of existing object.
- Copy assignment: existing object replaced as a copy of another existing object.
- Destruction: object destroyed when it's out of scope.

Not a SMF - calls a constructor with parameters → {0, 0, 0}

```cpp
using std::vector;
vector<int> func(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2};
  vector<int> vec{};
  vector<int> vec{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- Default construction: object created with no parameters.
- Copy construction: object treated as copy of existing object.
- Copy assignment: existing object replaced as a copy of another existing object.
- Destruction: object destroyed when it's out of scope.

Also not a SMF, uses initializer_list

```
using std::vector;
vector<int> func(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2};
  vector<int> vec{};
  vector<int> vec{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- Default construction: object created with no parameters.
- Copy construction: object treated as copy of existing object.
- Copy assignment: existing object replaced as a copy of another existing object.
- Destruction: object destroyed when it's out of scope.

A function declaration! (C++'s most vexing parse)

```
using std::vector;
vector<int> func(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2};
  vector<int> vec{};
  vector<int> vec{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- Default construction: object created with no parameters.
- Copy construction: object treated as copy of existing object.
- Copy assignment: existing object replaced as a copy of another existing object.
- Destruction: object destroyed when it's out of scope.

Copy constructor - vector created as copy of another

```
using std::vector;
vector<int> func(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2};
  vector<int> vec{};
  vector<int> vec{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- Default construction: object created with no parameters.
- Copy construction: object treated as copy of existing object.
- Copy assignment: existing object replaced as a copy of another existing object.
- Destruction: object destroyed when it's out of scope.

Also the default constructor!

```
using std::vector;
vector<int> func(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2};
  vector<int> vec{};
  vector<int> vec{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- Default construction: object created with no parameters.
- Copy construction: object treated as copy of existing object.
- Copy assignment: existing object replaced as a copy of another existing object.
- Destruction: object destroyed when it's out of scope.

Copy constructor

```
using std::vector;
vector<int> func(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2};
  vector<int> vec{};
  vector<int> vec{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- Default construction: object created with no parameters.
- Copy construction: object treated as copy of existing object.
- Copy assignment: existing object replaced as a copy of another existing object.
- Destruction: object destroyed when it's out of scope.

Copy constructor - vec8 is newly constructor

```
using std::vector;
vector<int> func(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2};
  vector<int> vec{};
  vector<int> vec{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- Default construction: object created with no parameters.
- Copy construction: object treated as copy of existing object.
- Copy assignment: existing object replaced as a copy of another existing object.
- Destruction: object destroyed when it's out of scope.

Copy assignment - vec8 is an existing object

```
using std::vector;
vector<int> func(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2};
  vector<int> vec{};
  vector<int> vec{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- Default construction: object created with no parameters.
- Copy construction: object treated as copy of existing object.
- Copy assignment: existing object replaced as a copy of another existing object.
- Destruction: object destroyed when it's out of scope.

Copy constructor: copies vec8 to location outside of func

```
using std::vector;
vector<int> func(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2};
  vector<int> vec{};
  vector<int> vec{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- Default construction: object created with no parameters.
- Copy construction: object treated as copy of existing object.
- Copy assignment: existing object replaced as a copy of another existing object.
- Destruction: object destroyed when it's out of scope.

Destructors on all values (except return value) are called

```
using std::vector;
vector<int> func(vector<int> vec0) {
  vector<int> vec1;
  vector<int> vec2(3);
  vector<int> vec3{3};
  vector<int> vec4();
  vector<int> vec5(vec2};
  vector<int> vec{};
  vector<int> vec{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}
```

- Default construction: object created with no parameters.
- Copy construction: object treated as copy of existing object.
- Copy assignment: existing object replaced as a copy of another existing object.
- Destruction: object destroyed when it's out of scope.

Copy Constructors and Copy Assignment Operators

Let's discuss their default behavior (SMFs) and how to change them!

Let's go in depth on constructors!

How we're used to writing constructors:

```
template <typename T>
vector<T>::vector<T>() {
  _size = 0;
  _capacity = kInitialSize;
  _elems = new T[kInitialSize];
}
```

Let's go in depth on constructors!

How we're used to writing constructors:

```
template <typename T>
vector<T>::vector<T>() {
  _size = 0;
  _capacity = kInitialSize;
  _elems = new T[kInitialSize];
}
```

members are first default constructed (declared to be their default values)

Then each member is reassigned. This seems wasteful!

Let's go in depth on constructors!

The technique below is called an ***initializer list***!

```
template <typename T>
vector<T>::vector<T>() {
  _size = 0;
  _capacity = kInitialSize;
  _elems = new T[kInitialSize];
}

template <typename T>
vector<T>::vector<T>() :
  _size(0), _capacity(kInitialSize),
  _elems(new T[kInitialSize]) { }
```

Directly construct each member with a starting value!

Quick summary of initializer lists

- Prefer to use member initializer lists, which directly constructs each member with a given value.
  - Faster! Why construct, and then immediately reassign?
  - What if members are a non-assignable type (you'll see by the end of lecture how this can be possible!)
- Important clarification: you can use member initializer lists for ANY constructor, even if it has parameters (and thus isn't an SMF)

Why aren't the default SMFs always sufficient?

Why aren't the default SMFs always sufficient?

The default compiler-generated copy constructor and copy assignment operator functions work by ***manually copying each member variable***!

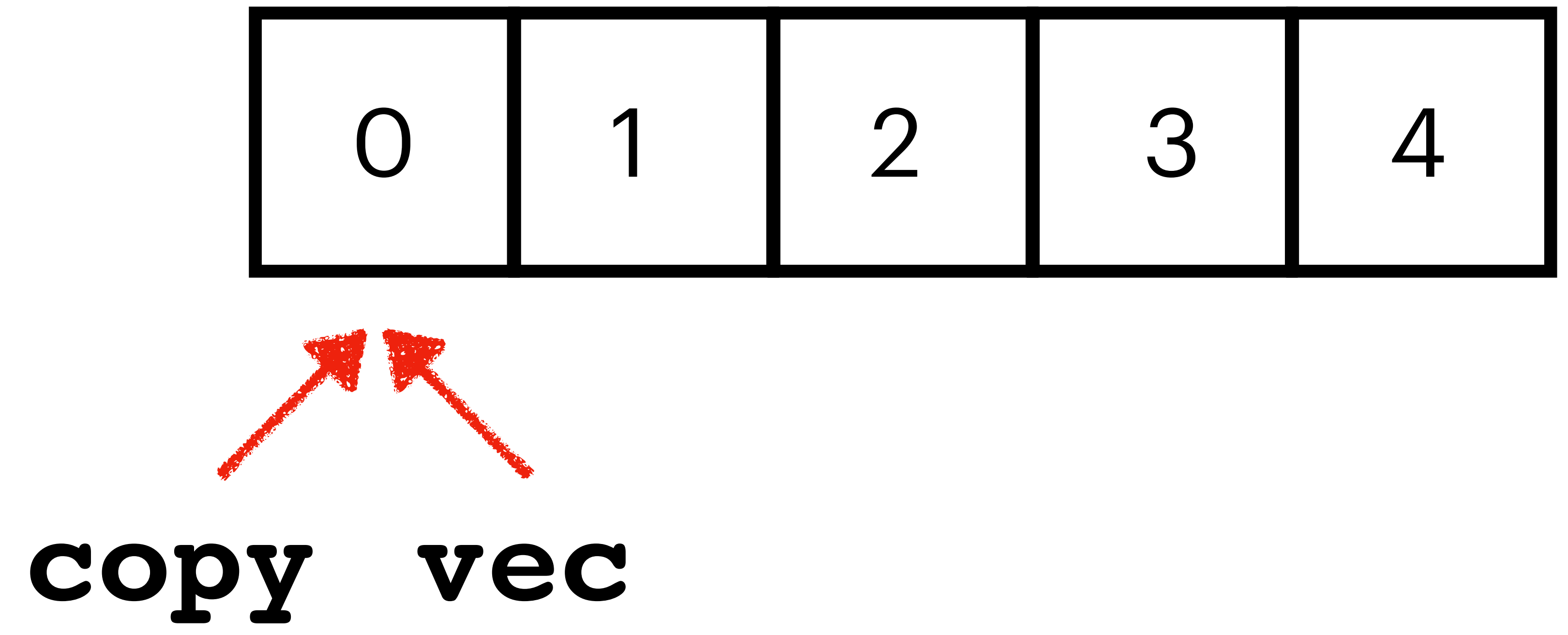
Here's a default copy constructor:

This SMF would automatically be generated for you:

```
template <typename T>
vector<T>::vector<T>(const vector::vector<T>& other) :
  _size(other._size),
  _capacity(other._capacity),
  _elems(other._elems) { }
```

Both copy and vec will point to the same underlying array!

```
vector<int> operator + (const vector<int>& vec, int elem) {
  vector<int> copy = vec;
  copy += element;          // assume that the += operator is defined
  return copy;
}
```



The culprit? This line in the default copy constructor!

- Remember, **`_elems`** is a pointer, so this line makes a copy of a pointer!
- This is very different from copying the underlying array that **`_elems`** points to!

```
template <typename T>
vector<T>::vector<T>(const vector::vector<T>& other) :
  _size(other._size),
  _capacity(other._capacity),
  _elems(other._elems) { }
```

Moral of the story: in many cases, copying is not as simple as copying each member variable!

Moral of the story: in many cases, copying is not as simple as copying each member variable!

This is one example of when you might want to overwrite the default SMFs with your own implementation!

Copy operations: fixing the issues we just saw

- Before we continue, a quick recap of definitions!
- Default Constructor
  - Object created with no parameters, no member variables instantiated
- Copy Constructor
  - Object created as a copy of existing object (member variable-wise)
- Copy Assignment Operator
  - Existing object replaced as a copy of another existing object.
- Destructor
  - Object destroyed when it is out of scope.

How do we fix the default copy constructor?

Any ideas?

```
template <typename T>
vector<T>::vector<T>(const vector::vector<T>& other) :
  _size(other._size),
  _capacity(other._capacity),
  _elems(other._elems) {




}
```

We can create a new array!

Let's create a new array and copy all of the elements over?

```
template <typename T>
vector<T>::vector<T>(const vector::vector<T>& other) :
  _size(other._size),
  _capacity(other._capacity),
  _elems(other._elems) {
  _elems = new T[other._capacity];
  std::copy(other._elems,
            other._elems + other._size, _elems);
}
```

Even better: let's move this to the initializer list!

We can move our reassignment of _elems up!

```
template <typename T>
vector<T>::vector<T>(const vector::vector<T>& other) :
  _size(other._size),
  _capacity(other._capacity),
  _elems(new T[other._capacity]) {
  std::copy(other._elems,
            other._elems + other._size, _elems);

}
```

How do we fix the default copy constructor?

Similarly, we need to fix the default copy assignment operator.

```
template <typename T>
vector<T>& vector<T>::operator = (const vector<T>& other) {
  _size = other._size;
  _capacity = other._capacity;
  _elems = other._elems;
  return *this;






}
```

Attempt 1: Allocate a new array and copy over elements

Do you notice any problems with this approach!

```
template <typename T>
vector<T>& vector<T>::operator = (const vector<T>& other) {
  _size = other._size;
  _capacity = other._capacity;
  _elems = new T[other._capacity];




  std::copy(other._elems,
            other._elems + other._size, _elems);


}
```

A huge issue: memory leaks!

What happened to the old array of elements that **`_elems`** pointed to?

```
template <typename T>
vector<T>& vector<T>::operator = (const vector<T>& other) {
  _size = other._size;
  _capacity = other._capacity;
  _elems = new T[other._capacity];


  std::copy(other._elems,
            other._elems + other._size, _elems);

}
```

We've lost access to the old value of **`_elems`**, and leaked the array that it pointed to!

Attempt 2: Deallocate the old array and make a new one

Do you notice any problems with this approach?

```
template <typename T>
vector<T>& vector<T>::operator = (const vector<T>& other) {
  _size = other._size;
  _capacity = other._capacity;
  delete[] _elems;
  _elems = new T[other._capacity];



  std::copy(other._elems,
            other._elems + other._size, _elems);
}
```

Remember to return a reference to the vector itself!

Remember to check your function signature and return as necessary!

```
template <typename T>
vector<T>& vector<T>::operator = (const vector<T>& other) {
  _size = other._size;
  _capacity = other._capacity;
  delete[] _elems;
  _elems = new T[other._capacity];


  std::copy(other._elems,
            other._elems + other._size, _elems);
  return *this;
}
```

Also, be careful about self-reassignment!

Remember to handle all edge cases, including doing nothing!

```
template <typename T>
vector<T>& vector<T>::operator = (const vector<T>& other) {
  if (&other == this) return *this;
  _size = other._size;
  _capacity = other._capacity;
  delete[] _elems;
  _elems = new T[other._capacity];

  std::copy(other._elems,
            other._elems + other._size, _elems);
  return *this;
}
```

Summary: Steps to follow for an assignment operator

1. Check for self-assignment
2. Make sure to free existing members if applicable.
3. Copy assign each automatically assignable member.
4. Manually copy all other members
5. Return a reference to *this (that was just reassigned).

Copy operations must perform these tasks:

Copy constructor

- Use initializer list to copy members where simple copying does the correct thing.
  - int, other objects, etc.
- Manually copy all members otherwise
  - pointers to heap memory
  - non-copyable things

Copy assignment

- Clean up any resources in the existing object about to be overwritten
- Copy members using direct assignment when assignment works
- Manually copy members where assignment does not work
- You don't have to do these in this order!

Summary: Steps to follow for an assignment operator

1. Check for self-assignment
2. Make sure to free existing members if applicable.
3. Copy assign each automatically assignable member.
4. Manually copy all other members
5. Return a reference to *this (that was just reassigned).

= delete and = default

Manipulating SMFs when you write your own functions

What would you do to prevent copies?

Here's a class we would not like copy assignment/operators to be used on.

```
class PasswordManager {
  public:
    PasswordManager();
    ~PasswordManager();
    // other methods ...
    PasswordManager(const PasswordManager& rhs);
    PasswordManager& operator = (const PasswordManager& rhs);

  private:
    // other important members ...
}
```

What would you do to prevent copies?

Explicitly delete the copy member functions!

```
class PasswordManager {
  public:
    PasswordManager();
    PasswordManager(const PasswordManager& pm);
    ~PasswordManager();
    // other methods ...
    PasswordManager(const PasswordManager& rhs) = delete;
    PasswordManager& operator = (const PasswordManager& rhs) = delete;

  private:
    // other important members ...
}
```

What would you do to prevent copies?

Explicitly delete the copy member functions!

```
class PasswordManager {
  public:
    PasswordManager();
    PasswordManager(const PasswordManager&
    ~PasswordManager();
    // other methods ...
    PasswordManager(const PasswordManager& rhs) = delete;
    PasswordManager& operator = (const PasswordManager& rhs) = delete;

  private:
    // other important members ...
}
```

Adding `= delete;` after a function prototype tells C++ to ***not*** generate the corresponding SMF!

What if you wanted to keep SMFs if you're overwriting them?

Is there a way to keep, say, the default copy constructor if you write another constructor?

```
class PasswordManager {
  public:
    PasswordManager();
    PasswordManager(const PasswordManager& pm);
    ~PasswordManager();
    // other methods ...
    PasswordManager(const PasswordManager& rhs) = delete;
    PasswordManager& operator = (const PasswordManager& rhs) = delete;

  private:
    // other important members ...
}
```

What if you wanted to keep SMFs if you're overwriting them?

Is there a way to keep, say, the default copy constructor if you write another constructor?

```
class PasswordManager {
  public:
    PasswordManager();
    PasswordManager(const PasswordManager& pm) = default;
    ~PasswordManager();
    // other methods ...
    PasswordManager(const PasswordManager& rhs) = delete;
    PasswordManager& operator = (const PasswordManager& rhs) = delete;

  private:
    // other important members ...
}
```

What if you wanted to keep SMFs if you're overwriting them?

Is there a way to keep, say, the default copy constructor if you write another constructor?

```
class PasswordManager {
  public:
    PasswordManager();
    PasswordManager(const PasswordManager& pm) = default;
    ~PasswordManager();
    // other methods ...
    PasswordManager(const PasswordManager&
    PasswordManager& operator = (const Pass

  private:
    // other important members ...
}
```

Adding `= default;` after a function prototype tells C++ to still generate the default SMF, even if you're defining other SMFs!

Rule of 0 and Rule of 3

When should we rewrite SMFs?

Rule of 0

- If the default operations work, then don't define your own!

When should you define your own SMFs?

- When the default ones generated by the compiler won't work
- Most common reason: there's a resource that our class uses that's not stored inside of our class
  - e.g. dynamically allocated memory
    - our class only stores the pointers to arrays, not the arrays in memory itself!

Rule of 3 (C++ 98)

- If you explicitly define a copy constructor, copy assignment operator, or destructor, you should define all three!
- What's the rationale?
  - If you're explicitly writing your own copy operation, you're controlling certain resources manually
  - You should then manage the creation, use, ***and*** releasing of those resources!

Special Member Functions (SMFs)

- Let's recap about the four SMFs we've learned about so far!
- Default Constructor
  - Object created with no parameters, no member variables instantiated
- Copy Constructor
  - Object created as a copy of existing object (member variable-wise)
- Copy Assignment Operator
  - Existing object replaced as a copy of another existing object.
- Destructor
  - Object destroyed when it is out of scope.

Are these 4 enough?

We currently know about the default ctor, copy ctor, copy assignment operator, and destructor. Why do we need anything else?

Are these 4 enough?

We currently know about the default ctor, copy ctor, copy assignment operator, and destructor. Why do we need anything else?

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

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

Are these 4 enough?

Let's say we had to copy our current StringTable into another, whose reference is given to us, and we have no use for our StringTable afterwards.

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

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

Are these 4 enough?

Let's say we had to copy our current StringTable into another, whose reference is given to us, and we have no use for our StringTable afterwards.

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

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

The copy constructor will be extremely slow, as it'll copy over every single value in the `values` map!

Move constructors and move assignment operators

How are **`std::move`** operations implemented in SMFs?

Move Operations (C++11)

- These functions are generated only when they're called (and before any are explicitly defined by you):

```
class Widget {
  public:
    Widget();
    Widget (const Widget& w);
    Widget& operator = (const Widget& w); // copy assignment operator
    ~Widget();                            // destructor
    Widget (Widget&& rhs);                // move constructor
    Widget& operator = (Widget&& rhs);    // move assignment operator
}
```

- Allow for moving objects and std::move operations (rvalue refs)

Move Operations (C++11)

- Move constructors and move assignment operators will perform "memberwise moves"
- Defining a copy constructor does not affect generation of a default copy assignment operator, and vice versa
- Defining a move assignment operator ***prevents*** generation of a move copy constructor, and vice versa
  - Rationale: if the move assignment operator needs to be re-implemented, there'd likely be a problem with the move constructor

Some nuances to move operation SMFs

- Move operations are generated for classes only if these 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
    - Can get around all of these by using default:

Some nuances to move operation SMFs

- Move operations are generated for classes only if these 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
        - Can get around all of these by using default:

```
Widget(Widget&&) = default;
Widget& operator=(Widget&&) = default;          // support moving

Widget(const Widget&) = default;
Widget& operator=(const Widget&) = default;    // support copying
```