http://web.stanford.edu/class/cs106l/

Special Member Functions

Special operations within your classes!

CS106L - Spring 2024

Attendance!

https://bit.ly/3QC60PQ

Special Member Functions

http://web.stanford.edu/class/cs106l/

CONTENTS

Special Member Functions

http://web.stanford.edu/class/cs106l/

CONTENTS

As we may recall…

Classes have three main parts: **the constructor and destructor**, **member variables**, and **functions**.

As we may recall...

Classes have three main parts: **the constructor and destructor**, **member variables**, and **functions**.

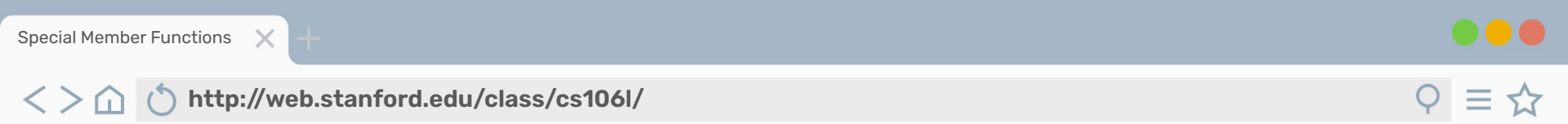
The constructor is called every time a new instance is created, and the destructor is called when it goes out of scope.

As we may recall...

Classes have three main parts: **the constructor and destructor**, **member variables**, and **functions**.

The constructor is called every time a new instance is created, and the destructor is called when it goes out of scope.

These are **special member functions** – every class has them by default!



There are six special member functions!

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

There are six special member functions!

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

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

There are six special member functions!

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
}
```

There are six special member functions!

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 operator
    ~Widget();
    Widget (Widget&& rhs);
    Widget& operator = (Widget&& rhs);         // move operator
}
```

Takes no parameters and creates a new object

There are six special member functions!

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();
    Widget (Widget&& rhs);
    Widget& operator = (Widget&& rhs);       // move assignment operator
}
```

Creates a new object as a member-wise copy of another

There are six special member functions!

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

Assigns an already existing object to another

There are six special member functions!

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
}
```

Called when an object goes out of scope

There are six special member functions!

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);    // 
    ~Widget();                               // destructor
    Widget (Widget&& rhs);                   // move constructor
    Widget& operator = (Widget&& rhs);       // move assignment operator
}
```

We'll come back to these!

There are six special member functions!

These functions are generated only when they're called (a

before any are explicitly defined by you):

We don't have to write out any of these! They all have default versions that are generated automatically!

```
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

http://web.stanford.edu/class/cs106l/

CONTENTS

Review: Initializer Lists

When we create a constructor, we need to initialize all of our member variables.

Review: Initializer Lists

When we create a constructor, we need to initialize all of our member variables.

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

Review: Initializer Lists

When we create a constructor, we need to initialize all of our member variables.

- However, initializing them to be the default value and then reassigning is inefficient!

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

Review: Initializer Lists

Instead, we can use initializer lists to declare and initialize them with the desired values all at once!

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

Review: Initializer Lists

- It's quicker and more efficient to directly construct member variables with intended values

Review: Initializer Lists

- It's quicker and more efficient to directly construct member variables with intended values
- What if the variable is a non-assignable type?

Review: Initializer Lists

- It's quicker and more efficient to directly construct member variables with intended values
- What if the variable is a non-assignable type?
- Can be used for any constructor, even non-default ones with parameters!

Why override special member functions?

Sometimes, the default special member functions aren't sufficient!

Why override special member functions?

Sometimes, the default special member functions aren't sufficient!

- By default, the copy constructor will create copies of each member variable.

Why override special member functions?

Sometimes, the default special member functions aren't sufficient!

- By default, the copy constructor will create copies of each member variable.

```
_newVar = _var;
```

Why override special member functions?

Sometimes, the default special member functions aren't sufficient!

- By default, the copy constructor will create copies of each member variable.
- This is member-wise copying!

```
_newVar = _var;
```

Why override special member functions?

Sometimes, the default special member functions aren't sufficient!

- By default, the copy constructor will create copies of each member variable.
- This is member-wise copying!
- But is this always good enough?

```
_newVar = _var;
```

What about pointers?

If your variable is a pointer, a member-wise copy will point to the same allocated data, not a fresh copy!

What about pointers?

If your variable is a pointer, a member-wise copy will point to the same allocated data, not a fresh copy!

- Look at our vector:

What about pointers?

If your variable is a pointer, a member-wise copy will point to the same allocated data, not a fresh copy!

- Look at our vector:

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

What about pointers?

If your variable is a pointer, a member-wise copy will point to the same allocated data, not a fresh copy!

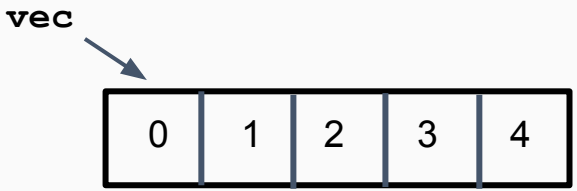
- Look at our vector:

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

These pointers will point at the same underlying array!

What about pointers?

If your variable is a pointer, a member-wise copy will point to the same allocated data, not a fresh copy!



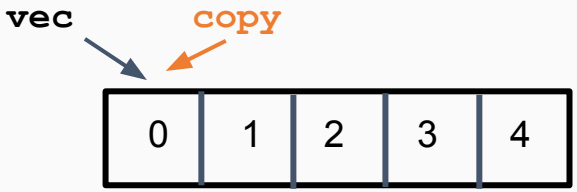
- Look at our vector:

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

These pointers will point at the same underlying array!

What about pointers?

If your variable is a pointer, a member-wise copy will point to the same allocated data, not a fresh copy!



- Look at our vector:

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

These pointers will point at the same underlying array!

Copying isn’t always simple!

Many times, you will want to create a copy that does more than just copies the member variables.

Copying isn't always simple!

Many times, you will want to create a copy that does more than just copies the member variables.

- Deep copy: an object that is a complete, **independent** copy of the original

Copying isn't always simple!

Many times, you will want to create a copy that does more than just copies the member variables.

- Deep copy: an object that is a complete, **independent** copy of the original

In these cases, you'd want to override the default special member functions with your own implementation!

Copying isn't always simple!

Many times, you will want to create a copy that does more than just copies the member variables.

- Deep copy: an object that is a complete, **independent** copy of the original

In these cases, you'd want to override the default special member functions with your own implementation!

Declare them in the header and write their implementation in the .cpp, like any function!

CONTENTS

What would you do to prevent copies?

Let's say you have a class that handles all of your passwords:

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

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

We can delete special member functions!

Setting a special member function to **delete** removes its functionality!

```
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 ...
}
```

We can delete special member functions!

Setting a special member function to **delete** removes its functionality!

```
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 ...
}
```

Now copying isn't a possible operation!

Uses

We can selectively allow functionality of special member functions!

Uses

We can selectively allow functionality of special member functions!

- This has lots of uses – what if we only want one copy of an instance to be allowed?
- This is how classes like `std::unique_ptr` work!

Uses

We can selectively allow functionality of special member functions!

- This has lots of uses – what if we only want one copy of an instance to be allowed?
- This is how classes like `std::unique_ptr` work!

The class satisfies the requirements of *MoveConstructible* and *MoveAssignable*, but of neither *CopyConstructible* nor *CopyAssignable*.

= default?

We can also keep the default copy constructor if we declare other constructors!

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

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

= default?

We can also keep the default copy constructor if we declare other constructors!

```
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 ...
}
```

Declaring any user-defined constructor will make the default disappear without this!

The Rule of 0

If the default SMFs work, **don't define your own**!

The Rule of 0

If the default SMFs work, **don’t define your own**!

We should only define new ones when the default ones generated by the compiler won't work.

The Rule of 0

If the default SMFs work, **don’t define your own**!

We should only define new ones when the default ones generated by the compiler won't work.

- This usually happens when we work with dynamically allocated memory, like pointers to things on the heap!

The Rule of 3

If you have to define a **destructor**, **copy constructor**, or **copy assignment operator**, **you should define all three**!

The Rule of 3

If you have to define a **destructor**, **copy constructor**, or **copy assignment operator**, **you should define all three**!

- Needing one signifies you're handling certain resources manually.

The Rule of 3

If you have to define a **destructor**, **copy constructor**, or **copy assignment operator**, **you should define all three**!

- Needing one signifies you're handling certain resources manually.
- We then should handle the creation, assignment, use, and destruction of those resources ourselves!

Recap

The four special member functions discussed 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.

Pop quiz!

What type of operation or function is each of these lines?

```
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;
}
```

Pop quiz!

What type of operation or function is each of these lines?

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;
}
```

Pop quiz!

What type of operation or function is each of these lines?

Not an SMF – a constructor with parameters

```
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;
}
```

Pop quiz!

What type of operation or function is each of these lines?

Not an SMF – uses an 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;
}
```

Pop quiz!

What type of operation or function is each of these lines?

Function declaration! Tricky!

```
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;
}
```

Pop quiz!

What type of operation or function is each of these lines?

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;
}
```

Pop quiz!

What type of operation or function is each of these lines?

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;
}
```

Pop quiz!

What type of operation or function is each of these lines?

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;
}
```

Pop quiz!

What type of operation or function is each of these lines?

Also 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;
}
```

Pop quiz!

What type of operation or function is each of these lines?

Copy assignment operator!

```
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;
}
```

Pop quiz!

What type of operation or function is each of these lines?

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;
}
```

Pop quiz!

What type of operation or function is each of these lines?

Also 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;
}
```

Special Member Functions

http://web.stanford.edu/class/cs106l/

CONTENTS

Is copying enough?

We've learned about the default constructor, destructor, and the copy constructor and assignment operator.

Is copying enough?

We've learned about the default constructor, destructor, and the copy constructor and assignment operator.

- We can create an object, get rid of it, and copy its values to another object!

Is copying enough?

We've learned about the default constructor, destructor, and the copy constructor and assignment operator.

- We can create an object, get rid of it, and copy its values to another object!
- Is this ever insufficient?

This can be wasteful!

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.

This can be wasteful!

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;
}
```

This can be wasteful!

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 copy every value in the `values` map one by one! Very slowly!

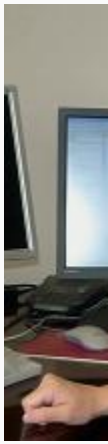
This can be wasteful!

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 copy every value in the `values` map one by one! Very slowly!



This can be wasteful!

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 copy every value in the `values` map one by one! Very slowly!

This can be wasteful!

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 copy every value in the `values` map one by one! Very slowly!

This can be wasteful!

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 copy every value in the `values` map one by one! Very slowly!

There are six special member functions!

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
}
```

There are six special member functions!

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);    // 
    ~Widget();                               // destructor
    Widget (Widget&& rhs);                   // move constructor
    Widget& operator = (Widget&& rhs);       // move assignment operator
}
```

Let's talk about it now!

Move operations

- Move constructors and move assignment operators will perform "memberwise moves"

Move operations

- Move constructors and move assignment operators will perform "memberwise moves"
- Defining a move assignment operator **prevents generation** of a move copy constructor, and vice versa

Move operations

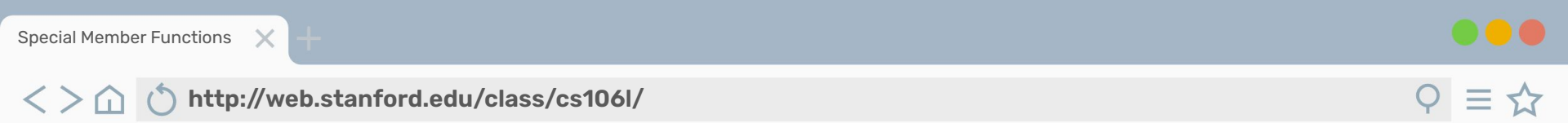
- Move constructors and move assignment operators will perform "memberwise moves"
- Defining a move assignment operator **prevents generation** of a move copy constructor, and vice versa

Unlike copy operations!

Move operations

- Move constructors and move assignment operators will perform "memberwise moves."
- Defining a move assignment operator **prevents generation** of a move copy constructor, and vice versa.
    - If the move assignment operator needs to be re-implemented, there'd likely be a problem with the move constructor!

Unlike copy operations!



Caveats

Move constructors and operators are only generated if:

Caveats

Move constructors and operators are only generated if:

- No copy operations are declared

Caveats

Move constructors and operators are only generated if:

- No copy operations are declared
- No move operations are declared

Caveats

Move constructors and operators are only generated if:

- No copy operations are declared
- No move operations are declared
- No destructor is declared

Caveats

Move constructors and operators are only generated if:

- No copy operations are declared
- No move operations are declared
- No destructor is declared

Declaring any of these will get rid of the default C++ generated operations.

Caveats

If we want to explicitly support move operations, we can set the operators to default:

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

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

Caveats

If we want to explicitly support move operations, we can set the operators to default:

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

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

Caveats

If we want to explicitly support move operations, we can set the operators to default:

???

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

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

http://web.stanford.edu/class/cs106l/

Thanks!

Next up: Move semantics!