Special Member Functions
Game Plan

• construction vs. assignment
• details and delete
• rule of three/zero
• copy elision
recap
There are 40 (+4) operators you can overload!

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>+</th>
<th>−</th>
<th>*</th>
<th>/</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+=</td>
<td>−=</td>
<td>*=</td>
<td>/=</td>
<td>%=</td>
</tr>
<tr>
<td>Bitwise</td>
<td>&amp;</td>
<td></td>
<td></td>
<td>~</td>
<td>!</td>
</tr>
<tr>
<td>Relational</td>
<td>==</td>
<td>!=</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&lt;=</td>
</tr>
<tr>
<td>Stream</td>
<td>&lt;&lt;=</td>
<td>&gt;&gt;=</td>
<td>&lt;&lt;=</td>
<td>&gt;&gt;=</td>
<td></td>
</tr>
<tr>
<td>Logical</td>
<td>&amp;&amp;</td>
<td></td>
<td></td>
<td></td>
<td>^</td>
</tr>
<tr>
<td>Increment</td>
<td>++</td>
<td>--</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>−&gt;</td>
<td>−&gt;</td>
<td>new</td>
<td>new[]</td>
<td>delete</td>
</tr>
<tr>
<td>Misc</td>
<td>( )</td>
<td>[ ]</td>
<td>,</td>
<td>=</td>
<td>co_await</td>
</tr>
</tbody>
</table>
How does C++ know how to apply operators to user-defined classes?

```cpp
vector<string> v{“Hello”, “World”};
cout << v[0];
v[1] += “!”;
```
Remember to follow the semantics of the operator.

```cpp
vector<string>& vector<string>::operator+=(const int& element) {
    push_back(element);
    return *this;
}

vector<string>& vector<string>::operator+=(const vector<int>& other) {
    for (int val : other) push_back(val);
    return *this;
}
```

Just like for primitives, `+=` should return reference to lhs.
Implement binary symmetric operators as non-member functions.

```cpp
vector<string> operator+(const vector<string>& vec, const string& element) {
    vector<string> copy = vec;
    copy += element;
    return copy;
}

vector<string> operator+(const vector<string>& lhs, const vector<string>& rhs) {
    vector<string> copy = lhs;
    copy += rhs;
    return copy;
}
```

The + operator treats both sides equally (doesn’t change either).
Always think about const vs. non-const for member functions.

```cpp
string& vector<string>::operator[](size_t index) {
    return _elems[index];
}

const string& vector<string>::operator[](size_t index) const {
    return _elems[index];
}
```
The client could call the subscript for both a const and non-const vector.

```cpp
vector<string> v1{“Green”, “Black”, “00-long”};
const vector<string> v2{“16.9”, “fluid”, “ounces”};

v1[1] = 0;  // calls non-const version, v1[1] is reference
int a = v2[1];  // calls const version, this works

```
Principle of Least Astonishment (POLA)

“If a necessary feature has a high astonishment factor, it may be necessary to redesign the feature”.
review: constructor and destructor
Example

Constructor and destructor: StringVector, Fraction, and LoggedStringVector.
Constructor and destructor for StringVector.

```cpp
// tip: use initializer list when possible.
StringVector::StringVector() :
    _logicalSize(0), _allocatedSize(kInitialSize) {
    _elems = new ValueType[allocatedSize];
}

StringVector::~StringVector() {
    delete [] _elems;
}
```
Special member functions are (usually) automatically generated by the compiler.

• Default construction: object created with no parameters.

• Copy construction: object is created as a copy of an existing object.

• Copy assignment: existing object replaced as a copy of another existing object.

• Destruction: object destroyed when it is out of scope.
Which special member function is called with each line?

```cpp
StringVector function(StringVector vec0) {
    StringVector vec1;
    StringVector vec2{"Ito", "En", "Green");
    StringVector vec3();
    StringVector vec4(vec2);
    StringVector vec5{};
    StringVector vec6{vec3 + vec4};
    StringVector vec7 = vec4;
    vec7 = vec2;
    return vec7;
}
```
Which special member function is called with each line?

```cpp
StringVector function(StringVector vec0) {
    StringVector vec1;
    StringVector vec2{"Ito", "En", "Green");
    StringVector vec3();
    StringVector vec4(vec2);
    StringVector vec5{};
    StringVector vec6{vec3 + vec4};
    StringVector vec7 = vec4;
    vec7 = vec2;
    return vec7;
}
```

Copy constructor
Which special member function is called with each line?

```cpp
StringVector function(StringVector vec0) {
    StringVector vec1;
    StringVector vec2{"Ito", "En", "Green"};
    StringVector vec3();
    StringVector vec4(vec2);
    StringVector vec5{};
    StringVector vec6{vec3 + vec4};
    StringVector vec7 = vec4;
    vec7 = vec2;
    return vec7;
}
```

Copy constructor: vec0 is constructed as a copy of the caller’s parameter.
Which special member function is called with each line?

```cpp
StringVector function(StringVector vec0) {
    StringVector vec1;
    StringVector vec2{"Ito", "En", "Green"};
    StringVector vec3();
    StringVector vec4(vec2);
    StringVector vec5{};
    StringVector vec6{vec3 + vec4};
    StringVector vec7 = vec4;
    vec7 = vec2;
    return vec7;
}
```

Default constructor: constructed with no parameters.
Which special member function is called with each line?

```cpp
StringVector function(StringVector vec0) {
    StringVector vec1;
    StringVector vec2{"Ito", "En", "Green"};
    StringVector vec3();
    StringVector vec4(vec2);
    StringVector vec5{};
    StringVector vec6{vec3 + vec4};
    StringVector vec7 = vec4;
    vec7 = vec2;
    return vec7;
}
```

Normal constructor: calls a user-defined function taking 3 string parameters.
Which special member function is called with each line?

```cpp
StringVector function(StringVector vec0) {
    StringVector vec1;
    StringVector vec2{"Ito", "En", "Green"};
    StringVector vec3();
    StringVector vec4(vec2);
    StringVector vec5{};
    StringVector vec6{vec3 + vec4};
    StringVector vec7 = vec4;
    vec7 = vec2;
    return vec7;
}
```

This declares a function! C++’s most vexing parse.
Which special member function is called with each line?

```cpp
StringVector function(StringVector vec0) {
    StringVector vec1;
    StringVector vec2{"Ito", "En", "Green"];
    StringVector vec3();
    StringVector vec4(vec2);    // Copy constructor: vec4 constructed as a copy of vec2.
    StringVector vec5{};
    StringVector vec6{vec3 + vec4};
    StringVector vec7 = vec4;
    vec7 = vec2;
    return vec7;
}
```
Which special member function is called with each line?

```cpp
StringVector function(StringVector vec0) {
    StringVector vec1;
    StringVector vec2{"Ito", "En", "Green"};
    StringVector vec3();
    StringVector vec4(vec2);
    StringVector vec5{};
    StringVector vec6{vec3 + vec4};
    StringVector vec7 = vec4;
    vec7 = vec2;
    return vec7;
}
```

Default constructor: vec5 constructed with zero parameters.
Which special member function is called with each line?

StringVector function(StringVector vec0) {
  StringVector vec1;
  StringVector vec2{"Ito", “En”, “Green”};
  StringVector vec3();
  StringVector vec4(vec2);
  StringVector vec5{};
  StringVector vec6{vec3 + vec4};
  StringVector vec7 = vec4;
  vec7 = vec2;
  return vec7;
}
Which special member function is called with each line?

```cpp
StringVector function(StringVector vec0) {
    StringVector vec1;
    StringVector vec2{"Ito", "En", "Green"};
    StringVector vec3();
    StringVector vec4(vec2);
    StringVector vec5{};
    StringVector vec6{vec3 + vec4};
    StringVector vec7 = vec4;
    vec7 = vec2;
    return vec7;
}
```

Copy constructor: vec7 constructed as a copy of vec4.
Which special member function is called with each line?

```cpp
StringVector function(StringVector vec0) {
    StringVector vec1;
    StringVector vec2{"Ito", "En", "Green"};
    StringVector vec3();
    StringVector vec4(vec2);
    StringVector vec5{};
    StringVector vec6{vec3 + vec4};
    StringVector vec7 = vec4;
    vec7 = vec2;
    return vec7;
}
```

Copy assignment: existing object vec7 is overwritten to be a copy of vec2.
Which special member function is called with each line?

```cpp
StringVector function(StringVector vec0) {
    StringVector vec1;
    StringVector vec2{"Ito", "En", "Green"};
    StringVector vec3();
    StringVector vec4(vec2);
    StringVector vec5{};
    StringVector vec6{vec3 + vec4};
    StringVector vec7 = vec4;
    vec7 = vec2;
    return vec7;
}
```

Copy constructor: creates a copy of local variable to be returned.
Which special member function is called with each line?

```cpp
StringVector function(StringVector vec0) {
    StringVector vec1;
    StringVector vec2{"Ito", "En", "Green"};
    StringVector vec3();
    StringVector vec4(vec2);
    StringVector vec5{};
    StringVector vec6{vec3 + vec4};
    StringVector vec7 = vec4;
    vec7 = vec2;
    return vec7;
}
```

Destructor called on all the vectors made except the return value.
problems from last time
I lied...this code doesn’t actually work.

```cpp
vector<int> operator+(const vector<int>& vec, int elem) {
    vector<int> copy = vec;
    copy += elem;
    return copy;
}
```
Copy is not as simple as copying each member.

```cpp
vec
int size 
int *elems

int size = 4
int *elems = 0

vector<int> vec = {1, 2, 3, 4};
vector<int> copy = vec;
vec += 1;
return copy;
```

```cpp
vector<int> operator+(const vector<int>& vec, int elem) {
    vector<int> copy = vec;
    copy += element;
    return copy;
}
```
Copy is not as simple as copying each member.

```cpp
vector<int> operator+(const vector<int>& vec, int elem) {
    vector<int> copy = vec;
    copy += elem;
    return copy;
}
```

```cpp
int size
int *elems
vec
1 2 3 4
```
Copy is not as simple as copying each member.

```cpp
int size
int *elems

vec

vector<int> operator+(const vector<int>& vec, int elem) {
    vector<int> copy = vec;
    copy += element;
    return copy;
}

int size
int *elems

copy

int size
int *elems

vector<int> copy

1 2 3 4
```
Copy is not as simple as copying each member.

```cpp
int size
int *elems

vector<int> vec;
int size = 4;
int *elems = NULL;

vector<int> copy;
int size = 5;
int *elems = NULL;

vector<int> operator+(const vector<int>& vec, int elem) {
    vector<int> copy = vec;
    copy += element;
    return copy;
}
```
At the return statement, a copy of the local variable is made.

```cpp
vector<int> operator+(const vector<int>& vec, int elem) {
    vector<int> copy = vec;
    copy += element;
    return copy;
}
```
When the copied vector goes out of scope, their destructor tries to free the array.

```cpp
int size
int *elems

vec

vector<int> operator+(const vector<int>& vec, int elem) {
    vector<int> copy = vec;
    copy += element;
    return copy;
} // copy goes out of scope
// note that vec is a reference
// that vector still exists

int size
int *elems

copy

copy goes out of scope
unnamed return value

int size
int *elems

int size
int *elems

1 2 3 4 5

When the copied vector goes out of scope, their destructor tries to free the array.

```cpp
vector<int> operator+(const vector<int>& vec, int elem) {
    vector<int> copy = vec;
    copy += element;
    return copy;
} // copy goes out of scope
// note that vec is a reference
// that vector still exists
```

Freed by copy’s destructor.
The original vector has been destroyed.

vector<int> operator+(const vector<int>& vec, int elem) {
    vector<int> copy = vec;
    copy += element;
    return copy;
} // copy goes out of scope
// note that vec is a reference
// that vector still exists

Freed by copy’s destructor.
The problem is this copy operation.

```cpp
vector<int> operator+(const vector<int>& vec, int elem) {
    vector<int> copy = vec;
    copy += element;
    return copy;
}
```
The copy operations must perform the following tasks.

Copy Constructor

• Use initializer list to copy members where assignment does the correct thing.
  • int, other objects, etc.

• Deep copy all members where assignment does not work.
  • pointers to heap memory

Copy Assignment

• Clean up any resources in the existing object about to be overwritten.

• Copy members using initializer list when assignment works.

• Deep copy members where assignment does not work.
Example

Copy constructor and copy assignment: StringVector and Fraction.
Copy constructor copies each member, creating deep copy when necessary.

```
StringVector::StringVector(const StringVector& other) :
    _logicalSize(other._logicalSize),
    _allocatedSize(other._allocatedSize) {

    _elems = new ValueType[_allocatedSize];
    std::copy(other.begin(), other.end(), begin());
}
```
The copy assignment needs to clean up this’s resources, then perform copy.

// can’t use initializer list – not a constructor!
StringVector& StringVector::operator=(const StringVector& rhs) {
    delete [] _elems;
    _logicalSize = rhs._logicalSize;
    _allocatedSize = rhs._allocatedSize;
    _elems = new ValueType[_allocatedSize];
    std::copy(other.begin(), other.end(), begin());
    return *this;
}
Careful about the edge case: self-assignment.

```cpp
// can’t use initializer list – not a constructor!
StringVector& StringVector::operator=(const StringVector& rhs) {
    if (this != &rhs) {
        delete [] _elems;
        _logicalSize = rhs._logicalSize;
        _allocatedSize = rhs._allocatedSize;
        _elems = new ValueType[_allocatedSize];
        std::copy(other.begin(), other.end(), begin());
    }
    return *this;
}
```
deleted operations
You can prevent copies from being made by explicitly deleting these operations.

class LoggedVector {
public:
    LoggedVector(int num, int denom);
~LoggedVector();
    // other methods
    LoggedVector(const LoggedVector& rhs) = delete;
    LoggedVector& operator=(const LoggedVector& rhs) = delete;
private:
    // other stuff
};
rule of three
When do you need to write your own special member functions?

When the default one generated by the compiler does not work.

Most common reason: ownership issues
A member is a handle on a resource outside of the class.
(eg: pointers, mutexes, filestreams.)
Rule of Three

If you explicitly define (or delete) a copy constructor, copy assignment, or destructor, you should define (or delete) all three.

What’s the rationale?
Rule of Three

If you explicitly define (or delete) a copy constructor, copy assignment, or destructor, you should define (or delete) all three.

The fact that you defined one of these means one of your members has ownership issues that need to be resolved.
Rule of Zero

If the default operations work, then don’t define your own custom ones.
more problems with copying
int main() {
    StringVector words;
    words = findAllWords(“words.txt”);
    // print words
}

StringVector findAllWords(const string& filename) {
    StringVector words;
    // read from filename using an ifstream
    return words;
}
How many vectors are created?

STACK

main()

HEAP
How many vectors are created?

main()

STACK

words

string *elems

HEAP
How many vectors are created?

STACK

main()

words

string *elems

findAllWords()

HEAP
How many vectors are created?

STACK

main()

words

string *elems

_HEAP

findAllWords()

words

string *elems
How many vectors are created?

STACK

main()

words

string *elems

findAllWords()

words

string *elems

HEAP

“ltot” “En” “Green” …10M elems… “Tea”
How many vectors are created?

STACK

main()
- words
  - string *elems

findAllWords()
- words
  - string *elems

HEAP
- unnamed return value
  - string *elems
- “Ito”
- “En”
- “Green”
- “Tea”
- “...10M elems…”

Copy constructor
How many vectors are created?

STACK

main()

words

string *elems

HEAP

unnamed return value

string *elems

“Ito” “En” “Green” ...10M elems... “Tea”
How many vectors are created?

STACK

main()

words

string *elems

unnamed return value

string *elems

HEAP

“Ito” “En” “Green” ...10M elems... “Tea”

“Ito” “En” “Green” ...10M elems... “Tea”

Copy assignment
How many vectors are created?

STACK

main()

words

string *elems

HEAP

“Ito”  “En”  “Green”  …10M elems…  “Tea”
That is a lot of copies.

```c
main()

string *elems

words

findAllWords()

string *elems

unnamed return value

Copy constructor

Copy assignment

STACK

main()

HEAP

words

"Ito"  "En"  "Green"  ...10M elems...  "Tea"

"Ito"  "En"  "Green"  ...10M elems...  "Tea"

unnamed return value

"Ito"  "En"  "Green"  ...10M elems...  "Tea"

Stack

Heap

Copy constructor

Copy assignment

24 November 2019
copy elision and return value optimization (RVO)
In practice: copy elision.

main()
In practice: copy elision.

```
main()
```

STACK

- `words`
- `string *elems`

HEAP

[Diagram showing the allocation of stack and heap memory for the `main()` function, with a pointer to `words` and `elems` on the stack.]
In practice: copy elision.

```
main()
```

```
findAllWords()
```

STACK

```
words
```

```
string *elems
```

HEAP
Compiler: “I know this vector is going to be returned, so I’ll just create it in main’s space.”
The words are still added as normal.

```
main()

words

string *elems

findAllWords()

words/return value

string *elems

STACK

HEAP

"Ito"  "En"  "Green"  ...10M elems...  "Tea"

findAllWords()

words

string *elems

STACK

HEAP

"Ito"  "En"  "Green"  ...10M elems...  "Tea"

main()
We get to skip the copy constructor call for the return.

```
main()

STACK

  words

  string *elems

  unnamed return value

  string *elems

HEAP

  “Ito”  “En”  “Green”  …10M elems…  “Tea”
```
We get to skip the copy constructor call for the return.

STACK

main()

words

string *elems

unnamed return value

string *elems

HEAP

“Ito”  “En”  “Green”  ...10M elems...  “Tea”

Copy assignment

“Ito”  “En”  “Green”  ...10M elems...  “Tea”
We get to skip the copy constructor call for the return.

```
main()

STACK

words

string *elems

HEAP

“Ito”
“En”
“Green”
...10M elems...
“Tea”
```
We get to skip the copy constructor call for the return.

STACK

main()

words

string *elems

findAllWords()

words/return value

string *elems

HEAP

“Ito”  “En”  “Green”  …10M elems…  “Tea”

Copy assignment

“Ito”  “En”  “Green”  …10M elems…  “Tea”

Copy elision!
Here’s a more clever idea?

STACK

main()

HEAP
Here’s a more clever idea?

```
main()

STACK

words

string *elems

HEAP
```
Here’s a more clever idea?

```c
main()

string *elems

findAllWords()

STACK

words

HEAP
```
Here’s a more clever idea?

STACK

main()

words

string *elems

findAllWords()

words/return value

string *elems

HEAP
Here’s a more clever idea?

STACK

main()

words

string *elems

words/return value

string *elems

 HEAP

“Ito” “En” “Green” ...10M elems… “Tea”

findAllWords()
Here's a more clever idea?

```
main()

STACK  HEAP

words

string *elems

unnamed return value

string *elems

“Ito” “En” “Green” ...10M elems... “Tea”
```
Let’s get rid of that empty array.

STACK

main()

words

string * elems

unnamed return value

string * elems

HEAP

“Ito”
“En”
“Green”
...10M elems...
“Tea”
Steal the array of the unnamed return value.
Evict the unnamed return value’s claim over the array.

```c
main()

STACK

words

string *elems

unnamed return value

string *elems

HEAP

“Ito”  “En”  “Green”  …10M

elems...  “Tea”
```

24 November 2019
The return value is temporary, so it will be gone on the very next line.

```
main()
    string *elems
```

STACK

*words*

HEAP

```
“Ito”  “En”  “Green”  ...10M elems...  “Tea”
```
Zero unnecessary copies!!!

main()

STACK

string *elems

words

HEAP

“Ito” “En” “Green” ...10M elems... “Tea”
Questions to ponder when procrastinating in Week 7.

• Are there instances where we can only copy, but NOT move?
• Are there instances where we can move, but NOT copy?
• Are there instances where we can either copy or move, and we PREFER move?
• How can a compiler tell whether we are ALLOWED to move?
• How does move impact compiler optimizations like RVO?
Next time

Move Semantics