Move Semantics
Game Plan

- lvalues vs. rvalues
- move constructor and assignment
- swap
- perfect forwarding
aside: emplace_back
recap
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.
Constructor and destructor for StringVector.

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

StringVector::~StringVector() {
    delete [] _elems;
}
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.
Copy constructor copies each member, creating deep copy when necessary.

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

    _elems = new ValueType[_allocatedSize];
    std::copy(other.begin(), other.end(), begin());
}
```
Careful about the edge case: self-assignment.

// 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;
}
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.
problems with copying
Quick quiz: how many times is each special member function called (with and without copy elision)?

```cpp
int main() {
    StringVector names1 = readNames(54321234);
    StringVector names2;
    names2 = readNames(54321234);
    cout << "done!" << endl;
}

StringVector findAllWords(size_t size) {
    StringVector names(size, "Ito");
    return names;
}
```
lvalues and rvalues

Note: this is a simplification of a complicated topic!
Value Categories: l-value vs. r-value

An **l-value** is an expression that has a name (identity).
- can find address using address-of operator (&var)

An **r-value** is an expression that does not have a name (identity).
- temporary values
- cannot find address using address-of operator (&var)

Intuitive though technically not true: l-values can appear on both sides of assignments, but r-values *usually* can only appear on the right side of an assignment.
Intuitively (though not technically true...):

An l-value is an expression that can appear either left or right of an assignment.***

An r-value is an expression that can appear only on the right of an assignment.***

***technically there are these weird things called gl-values, pr-values, x-values, ...
Examples: what the value categories of each expression?

```cpp
int val = 2;
int* ptr = 0x02248837;
vector<int> v1{1, 2, 3};

auto v4 = v1 + v2;
v1 += v4;
size_t size = v.size();
val = static_cast<int>(size);
v1[1] = 4*i;
ptr = &val;
v1[2] = *ptr;
```
Examples: what the value categories of each expression?

```cpp
int val = 2; // lvalue = rvalue
int* ptr = 0x02248837; // lvalue = rvalue
vector<int> v1{1, 2, 3}; // v1 = {1, 2, 3}, lvalue = rvalue

auto v4 = v1 + v2; // lvalue = rvalue
v1 += v4; // lvalue += lvalue
size_t size = v.size(); // lvalue = rvalue
val = static_cast<int>(size); // lvalue = rvalue (due to cast)

v1[1] = 4*i; // lvalue = rvalue
ptr = &val; // lvalue = rvalue
v1[2] = *ptr; // lvalue = lvalue
```
Examples: what the value categories of each expression?

```cpp
int val = 2;
int* ptr = 0x02248837;
vector<int> v1{1, 2, 3};

auto& ptr2 = ptr; // ptr2 is an l-value reference
auto&& v4 = v1 + v2; // v4 is an r-value reference
auto& ptr3 = &val; // ERROR: can’t bind l-val ref to r-value
auto&& val2 = val; // ERROR: can’t bind r-val ref to l-value
const auto& ptr3 = ptr + 5; // OKAY: CAN bind const l-val ref to r-value (WHY?)
```
Fun C++ errors: “Invalid non-const ref of type X& from r-value of type X”

```cpp
void nocos_Lref(vector<int>& v);
void const_Lref(const vector<int>& v);
void nocos_Rref(vector<int>&& v);
// BTW: no one uses const_Rref

vector<int> v1 = v2 + v3;
nocos_Lref(v1);
nocos_Rref(v1);
nocos_Lref(v2 + v3);
const_Rref(v2 + v3);
nocos_Rref(v2 + v3);
```
Fun C++ errors: "Invalid non-const ref of type X& from r-value of type X"

```cpp
void nocos_Lref(vector<int>& v);
void const_Lref(const vector<int>& v);
void nocos_Rref(vector<int>&& v);
// BTW: no one uses const_RRef

vector<int> v1 = v2 + v3; // v1 is l-value
nocos_Lref(v1); // OKAY: l-val ref binds to l-v
nocos_Rref(v1); // ERROR: r-val ref NO bind to r-v
nocos_Lref(v2 + v3); // ERROR: l-val ref NO bind to l-v
const_Rref(v2 + v3); // OKAY: const l-ref binds to r-v
nocos_Rref(v2 + v3); // OKAY: r-val ref binds to r-v
```
move operations

This is pretty conceptually intense. Please stop me at any time if you have questions!
Why r-values are key to move semantics.

An object that is an l-value is NOT disposable.

An object that is an r-value is disposable.
Why r-values are key to move semantics.

An object that is an l-value is NOT disposable, so you can copy from, but definitely cannot move from.

An object that is an r-value is disposable, so you can either copy or move from.

Why?
Why r-values are key to move semantics.

An object that is an l-value is NOT disposable, so you can copy from, but definitely cannot move from.

An object that is an r-value is disposable, so you can either copy or move from.

Key insight: if an object might potentially be reused, you cannot steal its resources.
Welcome the two new special member functions!

- Default constructor
- Copy constructor (create new from existing l-value)
- Copy assignment (overwrite existing from existing l-value)
- Destructor
- Move constructor (create new from existing r-value)
- Move assignment (overwrite existing from existing r-value)
Function signatures of all our special member functions.

```
StringVector();
StringVector(const StringVector& other) noexcept;
StringVector& operator=(const StringVector& rhs) noexcept;
~StringVector();

StringVector(StringVector&& other) noexcept;
StringVector& operator=(StringVector&& rhs) noexcept;
```
Key steps for a move constructor

• Transfer the contents of other to this.
  • Move instead of copy whenever possible!

• Leave other in an undetermined but valid state
  • Highly recommended: set it to the default value of class
Move constructor
(warning: this is not perfect...we’ll come back to this!)

```
StringVector(StringVector&& other) noexcept :
    elems(other.elems), logicalSize(other.logicalSize),
    allocatedSize(other.allocatedSize) {

    other.elems = nullptr;
}
```
Key steps for a move assignment

- Prevent self-assignment.
- Clean up resources in this.
- Transfer the contents of other to this.
  - Perform memberwise move
- Leave other in an undefined but valid state
  - Highly recommended: set it to the default value of class
Simulation of move assignment.

main()

STACK

words

string *elems

unnamed return value

string *elems

HEAP

“Ito”  “En”  “Green”  ...10M elems...  “Tea”
Simulation of move assignment.

```
main()

STACK

words

string *elems

unnamed return value

string *elems

HEAP

“Ito”

“En”

“Green”

...10M elems...

“Tea”
```
Simulation of move assignment.

```
main()

words

string *elems

unnamed return value

string *elems

```

STACK

HEAP

```
“Itō” “En” “Green” ...10M elems... “Tea”

```

7 November 2019
Simulation of move assignment.

```
main()

STACK

words

string *elems

unnamed return value

string *elems

HEAP

“Ito” “En” “Green” ...10M elems... “Tea”
```
Simulation of move assignment.

STACK

main()

words

string *elems

HEAP

“Ito” “En” “Green” …10M elems... “Tea”
Move assignment
(warning: this is not perfect...we’ll come back to this!)

```cpp
StringVector& operator=(StringVector&& rhs) noexcept {
    if (this == &rhs) {
        delete[] elems;
        allocatedSize = rhs.allocatedSize;
        logicalSize = rhs.logicalSize;
        elems = rhs.elems;
        rhs.elems = nullptr;
    }
    return *this;
}
```
This is a small problem in our code.

Did we actually move all the members?
Consider this other example: are we correctly moving the other members?

class Axess {
public:
    Axess()
    // other special member functions
    Axess(Axess&& other) : students(other.students) { }
    Axess& operator=(Axess&& rhs) {
        students = rhs.students;
    }
private:
    vector<Student> students;
}
Consider this other example: are we correctly moving the other members?

class Axess {
public:
    Axess()
    // other special member functions
    Axess(Axess&& other) : students(other.students) {} 
    Axess& operator=(Axess&& rhs) {
        students = rhs.students;
    }
private:
    vector<Student> students;
}
The type of the parameter is an rvalue reference, but they themselves are lvalues.

Axess(Axess&& other) : students(other.students) {
}

Axess& operator=(Axess&& rhs) {
    students = rhs.students;
}

They have names, and you can find their addresses!
The solution: `std::move` unconditionally casts a variable to an r-value.

```cpp
Axess(Axess&& other) : students(std::move(other.students)) {
}

Axess& operator=(Axess&& rhs) {
    students = std::move(rhs.students);
}
```

Note: move by itself does not actually move anything!
Move constructor
(fixed by moving all the members)

```cpp
StringVector(StringVector&& other) noexcept :
  elems(std::move(other.elems)),
  logicalSize(std::move(other.logicalSize)),
  allocatedSize(std::move(other.allocatedSize)) {

  other.elems = nullptr;
}
```
Move assignment
(fixed by moving all the members)

```cpp
StringVector& operator=(StringVector&& rhs) noexcept {
    if (this == &rhs) {
        delete[] elems;
        allocatedSize = std::move(rhs.allocatedSize);
        logicalSize = std::move(rhs.logicalSize);
        elems = std::move(rhs.elems);
        rhs.elems = nullptr;
    }
    return *this;
}
```
Key takeaways about std::move.

• Move constructor/assignment operators must perform member-wise move.

• The parameters that are r-value references are actually l-values.

• std::move unconditionally casts an expression to an r-value.

• std::move by itself does not move anything!

• After std::move-ing an object, do not use that object anymore.
We now call the move constructor/assignment operator, and is very fast.

```cpp
int main() {
    StringVector names1 = readNames(54321234);
    StringVector names2;
    names2 = readNames(54321234);
    cout << "done!" << endl;
}

StringVector findAllWords(size_t size) {
    StringVector names(size, "Ito");
    return names;
}
```
What if you wanted to do this instead:

```cpp
int main() {
    StringVector names1 = readNames(54321234);
    StringVector names2;
    names2 = names1;
    cout << "done!" << endl;
}

StringVector findAllWords(size_t size) {
    StringVector names(size, "Ito");
    return names;
}
```
The fix: use std::move.

```cpp
int main() {
    StringVector names1 = readNames(54321234);
    StringVector names2;
    names2 = std::move(names1);
    cout << "done!" << endl;
}

StringVector findAllWords(size_t size) {
    StringVector names(size, "Ito");
    return names;
}
```
Final followup: what happens if you don’t declare the move operations but pass in an r-value?

```cpp
StringVector();
StringVector(const StringVector& other) noexcept;
StringVector& operator=(const StringVector& rhs) noexcept;
~StringVector();

StringVector(StringVector&& other) noexcept;
StringVector& operator=(StringVector&& rhs) noexcept;

StringVector v1(v2 + v3); // parameter is r-value?
```
swap
Your task: write a generic swap function.

```cpp
int main() {
    vector<string> v1("En", 73837463);
    vector<string> v2("Ito", 10000000);
    swap(v1, v2);

    Patient patient1{"Anna", 2};
    Patient patient2{"Avery", 3};
    swap(patient1, patient2);
}
```
Simulation of move assignment.

STACK

swap()

a

string *elems

b

string *elems

HEAP

“En”  “En”  “En”  …73M elems...  “En”

“Ito”  “Ito”  “Ito”  …10M elems...  “Ito”
Simulation of move assignment.

swap()

STACK

a

string *elems

Move

b

string *elems

HEAP

"En"  "En"  "En"  ...73M elems...  "En"

"Ito"  "Ito"  "Ito"  ...10M elems...  "Ito"
Simulation of move assignment.

**STACK**

a
- string *elems

b
- string *elems

c
- string *elems

**HEAP**

"En"  "En"  "En"  ...73M elems...

"Ito"  "Ito"  "Ito"  ...10M elems...

swap()
Simulation of move assignment.

STACK

```
swap()

a

string *elems

b

string *elems

c

string *elems
```

HEAP

```
“En”  “En”  “En”  ...73M elems...  “En”

“Ito” “Ito” “Ito” ...10M elems... “Ito”
```
Simulation of move assignment.

```plaintext
swap()

STACK

a
 string *elems

b
 Move
 string *elems

c
 string *elems

HEAP

"En"  "En"  "En"  ...73M elems...  "En"

"Ito"  "Ito"  "Ito"  ...10M elems...  "Ito"
```
Simulation of move assignment.

STACK

swap()

a

string *elems

b

string *elems

C

string *elems

HEAP

“En”  “En”  “En”  ...73M elems...  “En”

“Ito”  “Ito”  “Ito”  ...10M elems...  “Ito”
Simulation of move assignment.

STACK

swap()

a
string *elems

b
string *elems

c
Move

HEAP

"En"  "En"  "En"  ...
...73M elems...
"En"

"Ito"  "Ito"  "Ito"  ...
...10M elems...
"Ito"
Simulation of move assignment.

STACK

swap() 

a

string *elems

b

string *elems

C

string *elems

HEAP

“En” “En” “En” ...

...73M elems...

“En”

“lto” “lto” “lto” ...

...10M elems...

“lto”
Simulation of move assignment.

STACK

swap()
a
string *elems

b
string *elems

HEAP

“En” “En” “En” ...73M elems... “En”

“Ito” “Ito” “Ito” ...10M elems... “Ito”
template <typename T>
void swap(T& a, T& b) noexcept {
    T c(std::move(a)); // move constructor
    a = std::move(b); // move assignment
    b = std::move(c); // move assignment
}
Rule of Five

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

The fact that you defined one of these means one of your members has ownership issues that need to be resolved.
When does the compiler automatically generate these functions?

```
<table>
<thead>
<tr>
<th>compiler implicitly declares</th>
<th>default constructor</th>
<th>destructor</th>
<th>copy constructor</th>
<th>copy assignment</th>
<th>move constructor</th>
<th>move assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nothing</td>
<td>defaulted</td>
<td>defaulted</td>
<td>defaulted</td>
<td>defaulted</td>
<td>defaulted</td>
<td>defaulted</td>
</tr>
<tr>
<td>Any constructor</td>
<td>not declared</td>
<td>defaulted</td>
<td>defaulted</td>
<td>defaulted</td>
<td>defaulted</td>
<td>defaulted</td>
</tr>
<tr>
<td>default constructor</td>
<td>user declared</td>
<td>defaulted</td>
<td>defaulted</td>
<td>defaulted</td>
<td>defaulted</td>
<td>defaulted</td>
</tr>
<tr>
<td>destructor</td>
<td>defaulted</td>
<td>user declared</td>
<td>defaulted</td>
<td>defaulted</td>
<td>not declared</td>
<td>not declared</td>
</tr>
<tr>
<td>copy constructor</td>
<td>not declared</td>
<td>defaulted</td>
<td>user declared</td>
<td>defaulted</td>
<td>not declared</td>
<td>not declared</td>
</tr>
<tr>
<td>copy assignment</td>
<td>defaulted</td>
<td>defaulted</td>
<td>defaulted</td>
<td>user declared</td>
<td>not declared</td>
<td>not declared</td>
</tr>
<tr>
<td>move constructor</td>
<td>not declared</td>
<td>defaulted</td>
<td>deleted</td>
<td>deleted</td>
<td>user declared</td>
<td>not declared</td>
</tr>
<tr>
<td>move assignment</td>
<td>defaulted</td>
<td>defaulted</td>
<td>deleted</td>
<td>deleted</td>
<td>not declared</td>
<td>user declared</td>
</tr>
</tbody>
</table>
```
perfect forwarding and emplace_back

Never mind, there’s no way we are getting this far. Ask me or ask on Piazza if you want to learn more!
Like seriously this stuff is sooooo cool.
Next time

Template classes