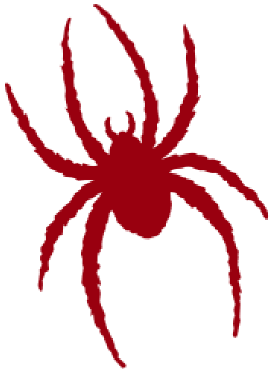
Copy Semantics and Move Semantics in C++

CMSC 240 Software Systems Development

Today – Copy/Move Semantics

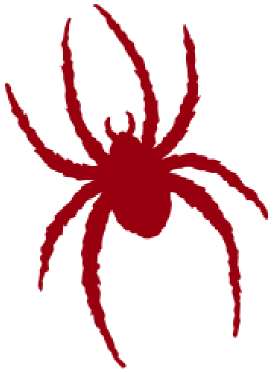
- Copy Semantics
- In-class exercise
- Move Semantics
- In-class exercise



RICHMOND

Today – Copy/Move Semantics

- Copy Semantics
- In-class exercise
- Move Semantics
- In-class exercise



Copy Semantics

Copy Semantics means **"the meaning of copy"**

- The rules for making copies of objects

What we want:

- After `x` is copied into `y` they are **equivalent** and **independent**
  - Equivalence: `x == y`
  - Independence: Modification to `x` does not cause modification to `y`

Object Passed by Value

```
int add_one_to(int value)  // Implicitly makes a copy
{
    value++;          // Increment the copied value.
    return value;
}

int main()
{
    int original = 1;
    int result = add_one_to(original);
    cout << "Original: " << original << "  Result: " << result << endl;
    return 0;
}
```

```
$ ./main
Original: 1  Result: 2
```

When you pass by value, a copy of the actual parameter is made (though you didn't explicitly ask for one)!

Object Passed by Value

- For **plain old data** (POD) types, it is a similar situation
  - Think of POD as a container of members
    - (which may have varying types)
  - The parameter receives a **member-wise copy**

```
struct Point
{
    int x;
    int y;
};

Point transpose(Point p)  // Again an implicit copy
{
    int temp = p.x;
    p.x = p.y;
    p.y = temp;
    return p;
}
```

Member-wise copying

- For built-in (int, float, char, etc.) and plain old data types, copying is done member wise.
  - It's just a **bit-by-bit copy into another location**
  - All good
- But for fully featured classes this can be problematic.

```cpp
class SimpleString
{
public:
    SimpleString(int max)
    : max_size{max}, length{0}
    {
        characters = new char[max_size];
        characters[0] = '\0';
    }

    ~SimpleString() { delete[] characters; }

    bool append(const char* str)
    {
        int str_length = strlen(str);
        if (length + str_length >= max_size)
            return false; // Not enough space to append the string

        strcpy(characters + length, str); // Append at the end of current string
        length += str_length;
        return true;
    }

    void print() const { cout << characters << endl; }
private:
    int max_size, length;
    char* characters;
};
```

What happens if we make a member-wise copy of this SimpleString object?

A Problem

- This can be **bad**
  - Any operation performed on the characters member of one object changes the other

```
int main()
{
    SimpleString myStringOne(20);
    myStringOne.append("Hello");

    SimpleString myStringTwo = myStringOne;  // Make a copy of String One
    myStringTwo.append(", World!");

    myStringOne.print();

    return 0;
}
```

```
$ ./SimpleString
Hello, World!
```

A Problem

- This can be **bad**
  - Any operation performed on the characters member of one object changes the other

```
SimpleString myStringOne(20);
myStringOne.append("Hello");
```

```
SimpleString myStringTwo = myStringOne;
myStringTwo.append(", World!");
```

```
private:
    int max_size, length;
    char* characters;
};
```

```
private:
    int max_size, length;
    char* characters;
};
```

A Problem

- This can be **dangerous!**
  - When one of the objects is destructed, characters is deleted. If the remaining SimpleString tries to write to its buffer, there is undefined behavior.
  - Worse, when the remaining SimpleString is destructed, characters is deleted again, causing a double free error.

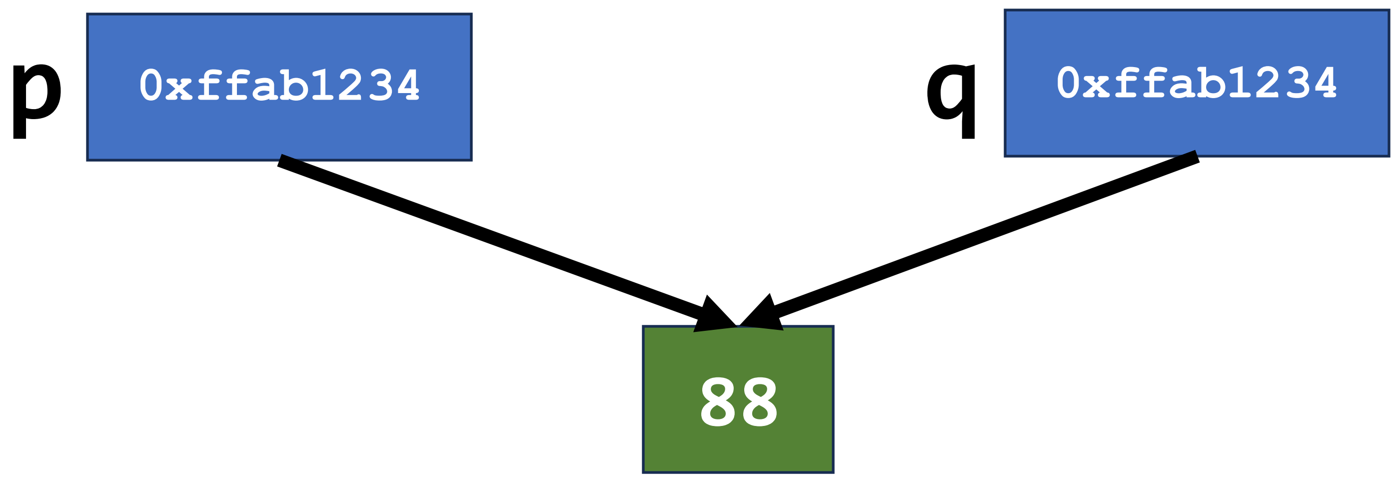
```
$ g++ SimpleString.cpp -o SimpleString
$ ./SimpleString
Hello, World!
free(): double free detected in tcache 2
Aborted (core dumped)
```

Copy Semantics are intended to avoid such situations

Copy Terminology

- **Shallow Copy**
  - Copies only a pointer so that the two pointers now refer to the same object.

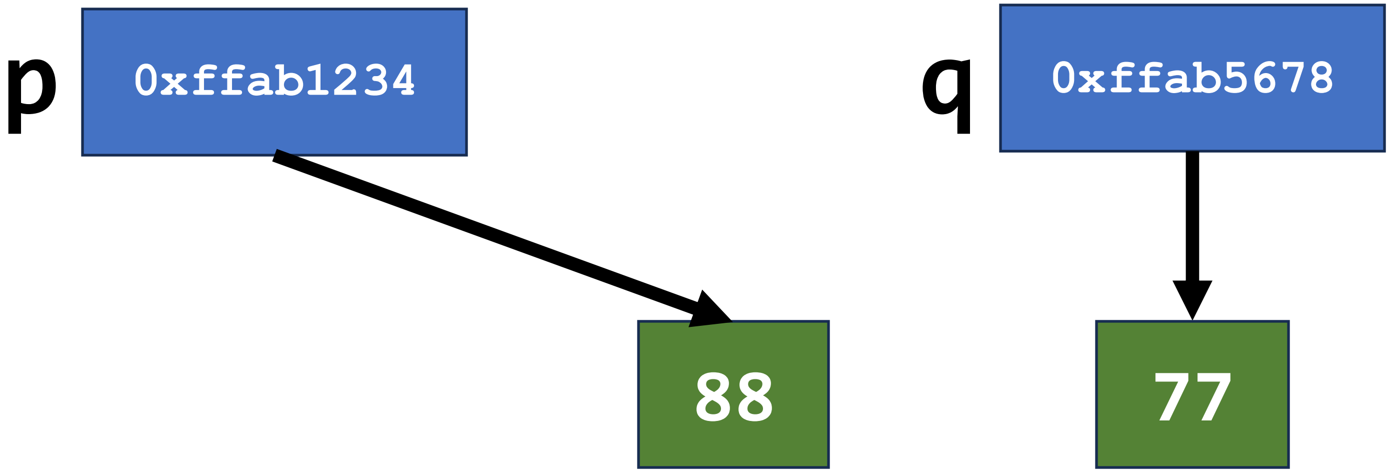
```
int* p = new int{77};
int* q = p;            // copy the pointer
*p = 88;               // change the value of the int pointed to by p AND q
```



Copy Terminology

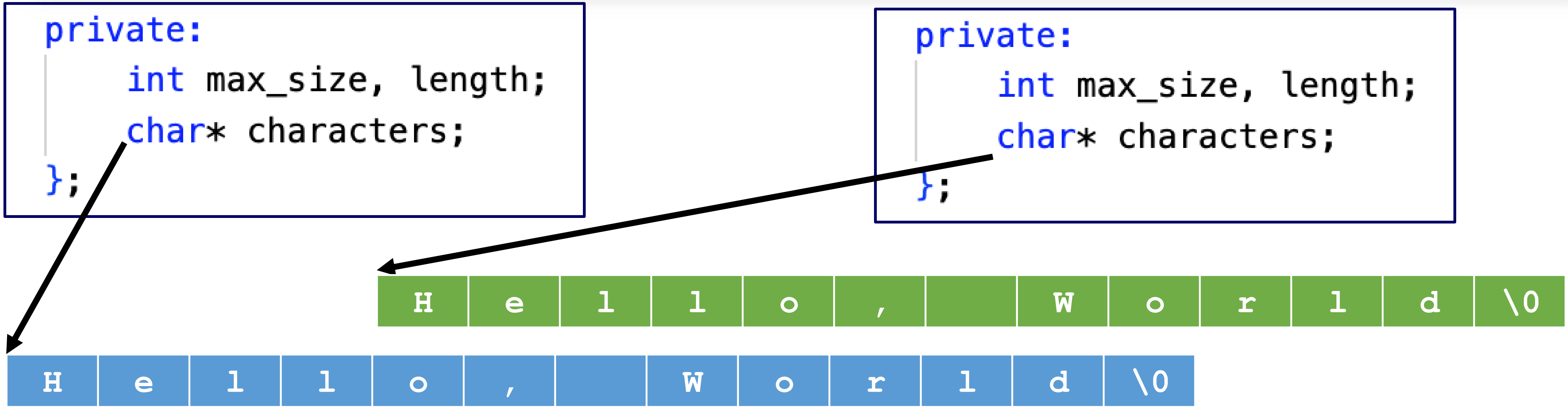
- **Deep Copy**
  - Copies what a pointer points to so that the two pointers now refer to distinct objects.

```
int* p = new int{77};
int* q = new int{*p};  // allocate a new int, then copy the value pointed to by p
*p = 88;               // change the value of the int pointed to by p ONLY
```



Method 1: Copy Constructor

```
// Copy constructor
SimpleString(const SimpleString& other)
    : max_size{other.max_size}, length{other.length}
{
    characters = new char[max_size];          // Get a new char array
    strcpy(characters, other.characters);    // Copy the other's characters
}
```



Code Demo

We Still Have a Problem

```
SimpleString stringOne{50};
stringOne.append("We apologize for the ");

SimpleString stringTwo{50};
stringTwo.append("last message.");

stringOne = stringTwo;
```

- We have not defined a copy assignment operator.

Method 2: Copy Assignment

```
SimpleString& operator=(const SimpleString& other)
{
    if (this == &other) // Self-assignment check
    {
        return *this;
    }

    // Clean up current object's resources
    delete[] characters;

    // Copy over data from other
    max_size = other.max_size;
    length = other.length;
    characters = new char[max_size];
    strcpy(characters, other.characters);

    return *this;
}
```

Default Copy

- Often the compiler will generate default copies for construction and assignment
  - Copy construction or copy assignment on each member of the class

- Be extremely careful with this!
  - Default is likely be wrong
  - Code your own copy constructor and copy assignment operators!

Turn Off Copying

- Some objects should not be copied

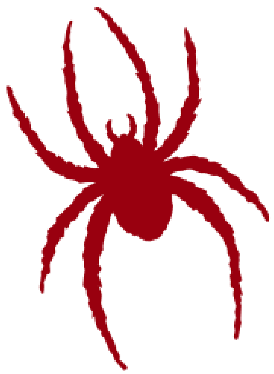
```
class Highlander
{
    Highlander(const Highlander& other) = delete;
    Highlander& operator=(const Highlander& other) = delete;

    // ...
};
```

- Any attempt to copy results in a compiler error

```
Highlander one{};
Highlander two{one};  // There can be only one.
```

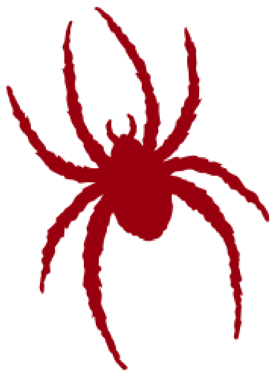
Today – Copy/Move Semantics

- Copy Semantics
- In-class exercise
- Move Semantics
- In-class exercise



Today – Copy/Move Semantics

- Copy Semantics
- In-class exercise
- Move Semantics
- In-class exercise



RICHMOND

Move Semantics

- Copying can be **time consuming** and **memory intensive**, especially if large amounts of data are involved
- It can be more efficient just to **transfer ownership** of resources from one object to another
- Copying and destroying the original works but can be inefficient

Move Semantics

Move semantics are the rules for moving objects

- **Requirements:** After object **y** is moved into object **x**
  - **x** is **equivalent to** the former value of **y**
  - **y** is in a special state called the **moved-from state**
    - Can only do two things with objects in this state: reassign or destruct

Value Categories

- Every expression in C++ has a **type** and **value** category
  - Value category describes what kinds of operations are valid for the expression

- Value categories:
  - **lvalue**: any value that has a name
  - **rvalue**: anything that is not an lvalue

Value Categories

- rvalue, lvalue arose from which side of = operator each originally appeared
  - Ex: **`int x = 50`** (**`x`** is lvalue, **`50`** is rvalue)
  - Not totally accurate: can have an lvalue on right side of =
    - E.g., in copy assignment

`lvalue` (`int i`) **`rvalue`** (`10`)

```
int i = 10;

10 = i;  // Error: Expression must be a modifiable lvalue

int a = i;
```

`lvalue` (`int a`) **`lvalue`** (`i`)

lvalue and rvalue References

- So far, all references we've used have been **lvalue** references
  - Denoted with a single **&**
  - For example,

```
void swap(double& v1, double& v2)
{
    double temp;
    temp = v2;
    v2 = v1;
    v1 = temp;
}
```

- However, function parameters can be **rvalue** references using **&&**

lvalue and rvalue References

```
void referenceType(int& ref)
{
    cout << "lvalue reference " << ref << endl;
}

void referenceType(int&& ref)
{
    cout << "rvalue reference " << ref << endl;
}

int main()
{
    int x = 1;
    referenceType(x);
    referenceType(2);
    referenceType(x + 2);
}
```

```
lvalue reference 1
rvalue reference 2
rvalue reference 3
```

`std::move`

- You can cast an **lvalue** reference to an **rvalue** reference using `std::move` and adding the `#include <utility>` header

- Note you never actually move anything, you are only casting

std::move

```
#include <iostream>
#include <utility>
using namespace std;

void referenceType(int& ref)
{
    cout << "lvalue reference " << ref << endl;
}

void referenceType(int&& ref)
{
    cout << "rvalue reference " << ref << endl;
}

int main()
{
    int x = 1;
    referenceType(move(x));
    referenceType(2);
    referenceType(x + 2);
}
```

```
rvalue reference 1
rvalue reference 2
rvalue reference 3
```

Move Constructors

- Like a copy constructor, but takes an **rvalue** reference

```
// Move Constructor
SimpleString(SimpleString&& other) noexcept
    : max_size{other.max_size}, length{other.length}, characters{other.characters}
{
    other.characters = nullptr; // Leave source in valid state
    other.length = 0;
    other.max_size = 0;
}
```

- **other** is an **rvalue** reference so you can "cannibalize" it
- Move constructor is designed to not throw an exception

Move Constructors

```
int main()
{
    SimpleString stringOne{50};
    stringOne.append("We apologize for the");

    cout << "stringOne: ";
    stringOne.print();

    SimpleString stringTwo{move(stringOne)};

    cout << "stringTwo: ";
    stringTwo.print();

    // Print stringOne again
    cout << "stringOne: ";
    stringOne.print();

    return 0;
}
```

```
stringOne: We apologize for the
stringTwo: We apologize for the
stringOne:
```

Move Assignment

- Like a copy assignment, but takes an **rvalue** reference

```
// Move Assignment Operator
SimpleString& operator=(SimpleString&& other) noexcept
{

    // ...
```

- And as with the move constructor, we designate it **`noexcept`**

```cpp
// Move Assignment Operator
SimpleString& operator=(SimpleString&& other) noexcept
{
    if (this == &other) // Self-assignment check
    {
        return *this;
    }

    // Clean up current resources
    delete[] characters;

    // Transfer ownership of resources
    max_size = other.max_size;
    length = other.length;
    characters = other.characters;

    // Leave source in valid state
    other.characters = nullptr;
    other.length = 0;
    other.max_size = 0;

    return *this;
}
```

```cpp
int main()
{
    SimpleString stringOne{50};
    stringOne.append("We apologize for the");

    SimpleString stringTwo{50};
    stringTwo.append("Last message");

    cout << "stringOne: ";
    stringOne.print();

    cout << "stringTwo: ";
    stringTwo.print();

    // Move stringOne to stringTwo
    stringTwo = move(stringOne);

    cout << "stringTwo: ";
    stringTwo.print();

    return 0;
}
```

```
stringOne: We apologize for the
stringTwo: Last message
stringTwo: We apologize for the
```

Code Demo

Compiler-Generated Methods

- Five methods govern move and copy behavior:
  1. The destructor
  2. The copy constructor
  3. The move constructor
  4. The copy assignment operator
  5. The move assignment operator

- Compiler can generate default implementations in some cases
- **Bottom line: you should define all five**

Today – Copy/Move Semantics

- Copy Semantics
- In-class exercise
- Move Semantics
- In-class exercise