Copy Semantics and Move Semantics in C++

CMSC 240
All examples borrowed/modified from
C++ Crash Course by Josh Lospinoso
No Starch Press
Copy Semantics

- Copy semantics mean “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
  - I.e., $x == y$ (equivalence)
  - Modification to $x$ does not cause modification to $y$ (independence)
```c
#include <cstdio>

int add_one_to(int x) {
    x++;
    return x;
}

int main() {
    auto original = 1;
    auto result = add_one_to(original);
    printf("Original: %d; Result: %d", original, result);
    return 0;
}
```
Object Passed by Value

- For plain old data (POD) types, similar situation
  - The parameter receives a *member-wise* copy

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

Point make_transpose(Point p) {
    int tmp = p.x;
    p.x = p.y;
    p.y = tmp;
    return p;
}
```
For fundamental 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 a problematic
What happens if we perform a member wise copy of a SimpleString object?
A Problem

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

```c
SimpleString a:
const size_t max_size = 50
size_t length = 14
char* buffer

SimpleString a_copy:
const size_t max_size = 50
size_t length = 14
char* buffer

We apologize for the null
```
A Problem

• This can be **dangerous!**
  - When one of the objects is destructed, buffer is deleted. If the remaining SimpleString tries to write its buffer, undefined behavior!

```
SimpleString a:
const size_t max_size = 50
size_t length = 14
char* buffer

SimpleString a_copy:
const size_t max_size = 50
size_t length = 14
char* buffer

We apologize for the
```
A Problem

- This can be very dangerous!
  - When the remaining object is destructed, buffer will be freed again, a *double free*
    - Which in some circumstances can cause serious security vulnerabilities (it messes with data structures that hold free store info)

```c
SimpleString a:
const size_t max_size = 50
size_t length = 14
char* buffer

SimpleString a_copy:
const size_t max_size = 50
size_t length = 14
char* buffer
```

We apologize for the

0
A Problem

• This can be very dangerous!
  ♦ When the remaining object is destructed, buffer will be freed again, a *double free*

```c
SimpleString a:
const size_t max_size = 50
size_t length = 14
char* buffer
```

```c
SimpleString a_copy:
const size_t max_size = 50
size_t length = 14
char* buffer
```

```c
We apologize for the
```

Copy Semantics are intended to avoid such situations
Shallow Copy vs Deep Copy

We want deep copies
Method 1: Copy Constructor

```cpp
SimpleString(const SimpleString& other) :
  max_size{ other.max_size }
, buffer{ new char[other.max_size] }
, length{ other.length }
{ std::strncpy(buffer, other.buffer, max_size); }
```

**SimpleString a:**
- `const size_t max_size = 50`
- `size_t length = 14`
- `char* buffer`

**SimpleString a_copy:**
- `const size_t max_size = 50`
- `size_t length = 14`
- `char* buffer`

- We apologize for the
- We apologize for the

Copy Constructor

- Copy constructor is automatically invoked when passing a SimpleString object into a method by value.

```java
void foo(SimpleString x) {
    x.append_line("This change is lost.");
}

int main() {
    SimpleString a{ 20 };  // Invokes copy constructor
    foo(a);
    a.print("Still empty");
}
```
Method 2: Copy Assignment

```cpp
void dont_do_this() {
    SimpleString a{ 50 };  
    a.append_line("We apologize for the");
    SimpleString b{ 50 };  
    b.append_line("Last message");
    b = a;
}
```

- The problems
  - Behavior is undefined because we have not defined a copy assignment operator
  - More complicated than copy construction because b might already have a value
    - So you have to clean up b’s resources before copying a
Copy Assignment

- Default behavior: Copy members from the source object to the destination object. **Dangerous!**
  - b’s buffer gets rewritten without freeing the original, which was dynamically allocated
  - Now a and b own the same buffer
    - Issue with change to one changing the other
    - Double free (once again)
Copy Assignment

- Default behavior: Copy members from the source object to the destination object. Dangerous!
- So you must implement a copy assignment operator that rectifies these issues (i.e., clean handoff)
Copy Assignment

```cpp
SimpleString& operator=(const SimpleString& other) {
    if(this == &other)
        return *this;
    const auto new_buffer = new char[other.max_size];
    delete[] buffer;
    buffer = new_buffer;
    length = other.length;
    max_size = other.max_size;
    std::strncpy(buffer, other.buffer, max_size);
    return *this;
}
```
Default Copy

- Often compiler will generate default copies for construction and assignment
  - Invoke copy construction or copy assignment on each member of the class
- Be extremely careful with this!
  - Default is likely to be wrong
  - Code your own copy constructor and copy assignment operators!
Default Copy

- To explicitly invoke default copy, use `default` keyword

```cpp
struct Replicant {
    Replicant(const Replicant&) = default;
    Replicant& operator=(const Replicant&) = default;
    --snip--
};
```
Repress Generation

- Some objects should not be copied
  - Manages a file
  - Represents a mutual exclusion lock

```c
struct Highlander {
    Highlander(const Highlander&) = delete;
    Highlander& operator=(const Highlander&) = delete;
    //snip--
};
```

- Any attempt to copy results in compiler error
  ```c
  int main() {
      Highlander a;
      Highlander b{ a }; // Bang! There can be only one.
  }
  ```
Move Semantics

• Copying can be time consuming, especially if large amount of data involved
• It can be more efficient to just *transfer ownership* of resources from one object to another
• Making a copy and destroying the original works, but is often inefficient
Move Semantics

- Move semantics is move’s corollary to copy semantics

- 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 object in this state: reassign or destruct
Move Semantics

- Move semantics is move’s corollary to copy semantics.
- Note moving is not just renaming: you’re dealing with separate objects with separate storage and potentially different lifetimes.
- As with copying, you must specify *move constructor* and *move assignment operator*.
Why Move?

- Consider:

```cpp
struct SimpleStringOwner {
    SimpleStringOwner(const char* x)
    : string{ 10 } {
        if(!string.append_line(x)) {
            throw std::runtime_error{ "Not enough memory!" };}
        }
        string.print("Constructed");
    }
~SimpleStringOwner() {
    string.print("About to destroy");
}

private:
    SimpleString string;
};
```
Why Move?

• Suppose you want to move a SimpleString into a SimpleStringOwner as follows:

```cpp
void own_a_string() {
    SimpleString a{ 50 };  
    a.append_line("We apologize for the");
    a.append_line("inconvenience.");
    SimpleStringOwner b{ a };
```

Assumes we have a copy constructor. So…
Why Move?

```cpp
--snip--
void own_a_string() {
    SimpleString a{ 50 ];
    a.append_line("We apologize for the");
    a.append_line("inconvenience.");
    SimpleStringOwner b{ a ];

--snip--
}

struct SimpleStringOwner {
    SimpleStringOwner(const SimpleString& my_string) : string{ my_string } { }
    --snip--
    private:
    SimpleString string;
};
```
Why Move?

Hidden waste: Caller never uses the pointed to object again after constructing string
Why Move?

Hidden waste: Caller never uses the pointed to object again after constructing `string`

```c
SimpleString a:
const size_t max_size = 50
size_t length = 14
char* buffer
```

```c
SimpleStringOwner b
SimpleString string:
const size_t max_size = 50
size_t length = 14
char* buffer
```

We apologize for the

Better to move the “guts” of SimpleString a into the string field of SimpleStringOwner b
Why Move?

What you want: SimpleStringOwner b steals the guts of SimpleString a and then sets a into a destructible state

Better to move the "guts" of SimpleString a into the string field of SimpleStringOwner b
Why Move?

What you want: `SimpleStringOwner b` steals the guts of `SimpleString a` and then sets `a` into a destructible state.

```
We apologize for the \n\0
```
Why Move?

What you want: SimpleStringOwner b steals the guts of SimpleString a and then sets a into a destructible state.

SimpleString a:
- size_t max_size = 0
- size_t length = 0
- char* buffer = nullptr

SimpleStringOwner b
- SimpleString string:
  - size_t max_size = 50
  - size_t length = 14
  - char* buffer

After the move, the SimpleString of b is equivalent to the former state of a, and a is destructible.
A Caveat

• Moving can be dangerous: If you accidentally use a moved-from object, you’ve got a problem
  - No guarantee that class invariants are satisfied in a moved-from object

• However, compiler has built-in safeguards: lvalues and rvalues
Value Categories

• Every expression has a *type* and *value category*
  ▶ Value category describes what kind of operations are valid for the expression

• Value categories in C++ can be complicated
  ▶ We’ll just take a relatively simplistic view:
    ▪ *lvalue*: any value that has a name
    ▪ *rvalue*: anything that isn’t an lvalue
Value Categories

• Example:

```java
SimpleString a{ 50 };
SimpleStringOwner b{ a };  // a is an lvalue
SimpleStringOwner c{ SimpleString{ 50 } };  // SimpleString{ 50 } is an rvalue
```

• 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
Ivalue and rvalue References

• Up to this point, all references we’ve used have been lvalue references
  • Denoted with single &
• You can take a parameter by rvalue reference using &&
Compiler is very good at determining whether an object is an lvalue or an rvalue

- You can use function overloading and compiler will call the correct function based on what arguments are provided on function invocation.
#include <cstdio>

void ref_type(int& x) {
    printf("lvalue reference %d\n", x);
}

void ref_type(int&& x) {
    printf("rvalue reference %d\n", x);
}

int main() {
    auto x = 1;
    ref_type(x);
    ref_type(2);
    ref_type(x + 2);
}

Output:

lvalue reference 1
rvalue reference 2
rvalue reference 3
std::move

- Cast an lvalue reference to an rvalue reference using `std::move` in the `<utility>` header
- Note you never actually move anything. You’re only casting
  - Probably should have been called `std::rvalue`
```cpp
#include <cstdio>
#include <utility>

void ref_type(int &x) {
    printf("lvalue reference %d\n", x);
}

void ref_type(int &&x) {
    printf("rvalue reference %d\n", x);
}

int main() {
    auto x = 1;
    ref_type(std::move(x));
    ref_type(2);
    ref_type(x + 2);
}
```

Output:
```
rvalue reference 1
rvalue reference 2
rvalue reference 3
```
std::move

• Warning: Be careful when using std::move
  ◆ You’ve removed the built-in safeguards that prevent you from interaction with a moved-from object
    ▪ Remember: only can reassign it or destroy it

• Rules:
  ◆ If you have lvalue, moving is suppressed
  ◆ If you have rvalue, moving enabled
Move Construction

- Like copy construction, but takes an rvalue reference instead of lvalue ref

```cpp
SimpleString(SimpleString&& other) noexcept
: max_size{ other.max_size },
  buffer(other.buffer),
  length(other.length) {
    other.length = 0;
    other.buffer = nullptr;
    other.max_size = 0;
}
```

- Other is an rvalue reference so you can “cannibalize” it
Move Construction

```cpp
SimpleString(SimpleString&& other) noexcept
    : max_size{ other.max_size },
      buffer(other.buffer),
      length(other.length) {
    other.length = 0;
    other.buffer = nullptr;
    other.max_size = 0;
}
```

- Copy all fields of `other` into this, zero out all fields of `other`
  - This is important: puts `other` in a moved-from state
    - What happens if not done, and `other` is destructed?
Move Construction

- Executing move constructor is much less expensive than copy constructor
- Move constructor is designed to not throw exception so you mark it noexcept
  - Preference should be to use noexcept move constructors
  - Compiler cannot use exception throwing move constructors and will use copy constructor instead
Move Assignment

• Analogous to copy assignment via `operator=`

• Move assignment operator takes rvalue reference instead of const lvalue reference
  • And as with move constructor, designate it `noexcept`
SimpleString& operator=(SimpleString&& other) noexcept {
    if(this == &other) {
        return *this;
    }
    delete[] buffer;
    buffer = other.buffer;
    length = other.length;
    max_size = other.max_size;
    other.buffer = nullptr;
    other.length = 0;
    other.max_size = 0;
    return *this;
}
Move Assignment

- We can use this now for the `SimpleString` constructor of `SimpleStringOwner`

```cpp
struct SimpleStringOwner {
    SimpleStringOwner(const char* x)
        : string{ 10 } {
            if(!string.append_line(x)) {
                throw std::runtime_error{ "Not enough memory!" };}
            string.print(" Constructed ");
        }
    string.print(" Constructed ");

    SimpleStringOwner(SimpleString&& x) : string{ std::move(x) } { }

    ~SimpleStringOwner() {
        string.print(" About to destroy ");
    }

    private:
    SimpleString string;
};
```
Move Assignment

- \( x \) is an lvalue, so you must use `std::move` to cast it to rvalue.
  - Might seem strange, since \( x \) is an rvalue `reference` when passed (note rvalue/lvalue and lvalue `reference` and rvalue `reference` are not same things)
  - But consider what happens if moved from \( x \) then tried to use it in the constructor

```cpp
SimpleStringOwner(SimpleString&& x) : string{ std::move(x)} {} 
```
Move Assignment

```cpp
int main() {
    SimpleString a{ 50 };  
a.append_line("We apologise for the");
    SimpleString b{ 50 };  
b.append_line("Last message");
a.print("a");
b.print("b");  
b = std::move(a);  
// a is "moved-from"
b.print("b");
}
```

Note need to cast `a` to rvalue in order to use move assignment

Output:

```
a: We apologise for the
b: Last message
b: We apologise for the
```
Compiler-Generated Methods

- Five methods govern move and copy behavior:
  - The destructor
  - The copy constructor
  - The move constructor
  - The copy assignment operator
  - The move assignment operator

- Compiler can generate default implementations on some cases
Compiler-Generated Methods

- Compiler can generate default implementations on some cases
  - But it varies among implementations and is complicated

- Rule-of-five: There are five methods to implement. Implement them all to avoid headaches down the road
• If you define nothing, compiler generates defaults for all five
  - This is the so called *rule-of-zero*
• If you define *any* of destructor/copy constructor/ or copy assignment operator, you get all three
  - Generally dangerous
• If you define *only* move semantics, compiler will *only* generate destructor
Compiler–Generated Methods

• Bottom line: define all five!