Copy Semantics and Move Semantics in C++

CMSC 240

Many examples borrowed/modified from

C++ Crash Course by Josh Lospinoso

No Starch Press

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
  - I.e., x==y (equivalence)
  - Modification to x does not cause modification to y (independence)

Object Passed by Value

```
#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;
}
```

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, 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, y;
};

Point make_transpose(Point p) {
  int tmp = p.x;
  p.x = p.y;
  p.y = tmp;
  return p;
}
```

Again an implicit copy

Bottom line

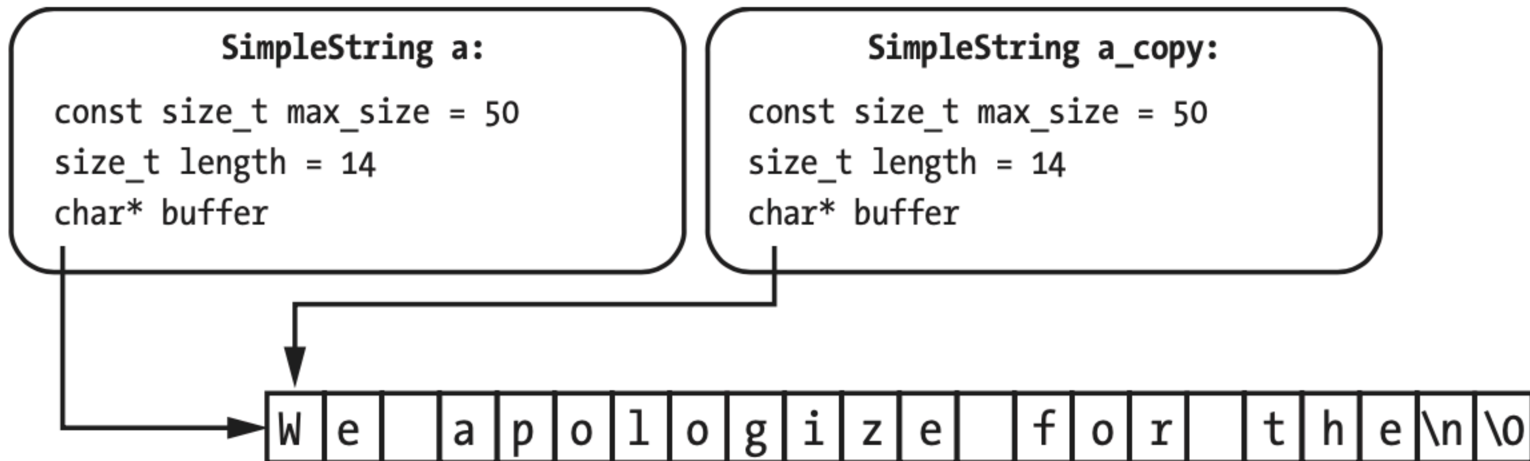
- For fundamental and plain old data types, copying is done member wise
  - It’s just a bit by bit copy into another location
  - All good
- But for fully featured classes, this can be a problematic

```cpp
#include <cstdio>
#include <cstring>
#include <stdexcept>

struct SimpleString {
  SimpleString(size_t max_size)
      : max_size{ max_size }
      , length{} {
    if(max_size == 0) {
      throw std::runtime_error{ "Max size must be at least 1." };
    }
    buffer = new char[max_size];
    buffer[0] = 0;
  }
  ~SimpleString() {
    delete[] buffer;
  }
  void print(const char* tag) const {
    printf("%s: %s", tag, buffer);
  }
  bool append_line(const char* x) {
    const auto x_len = strlen(x);
    if(x_len + length + 2 > max_size)
      return false;
    strncpy(buffer + length, x, max_size - length);
    length += x_len;
    buffer[length++] = '\n';
    buffer[length] = 0;
    return true;
  }

  private:
  size_t max_size;
  char* buffer;
  size_t length;
};
```

What happens if we perform a member wise copy of a SimpleString object?

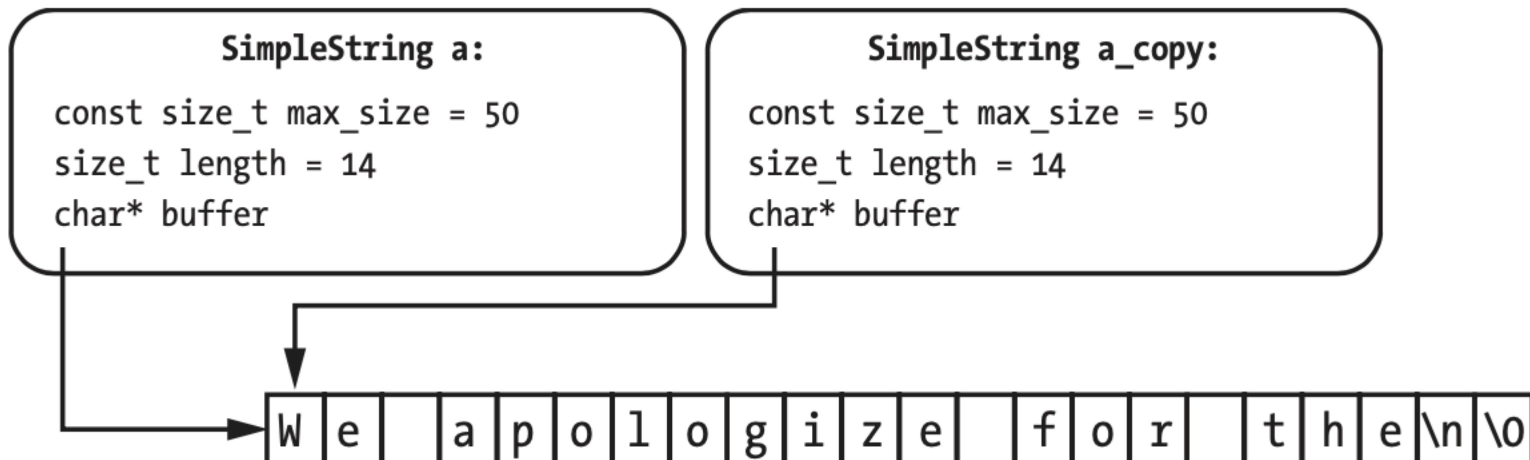
A Problem

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



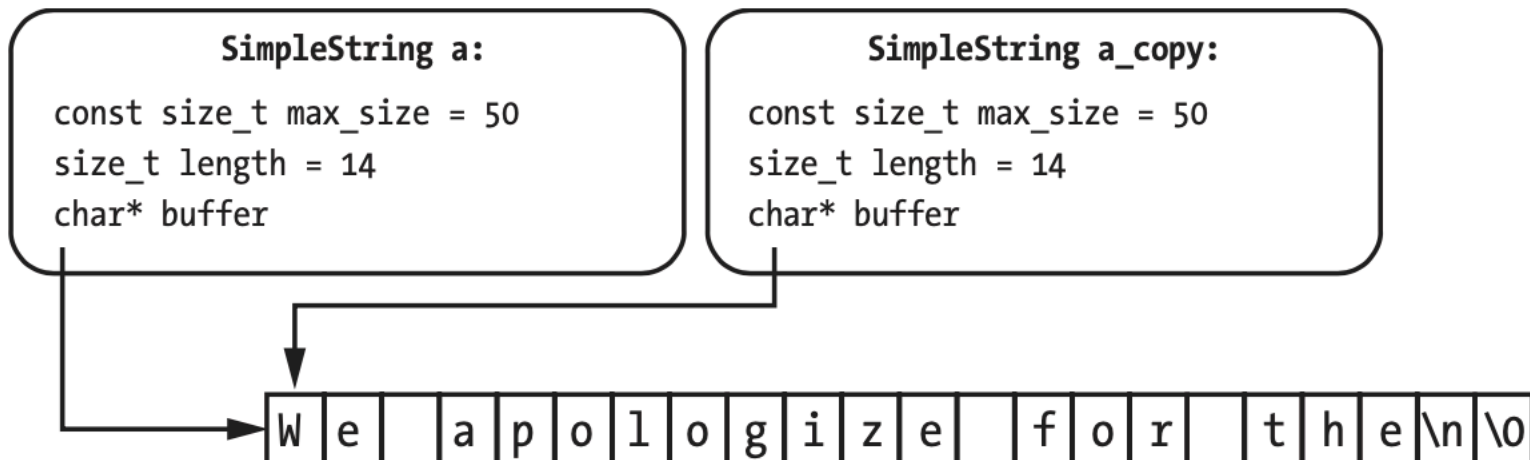
A Problem

- 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!



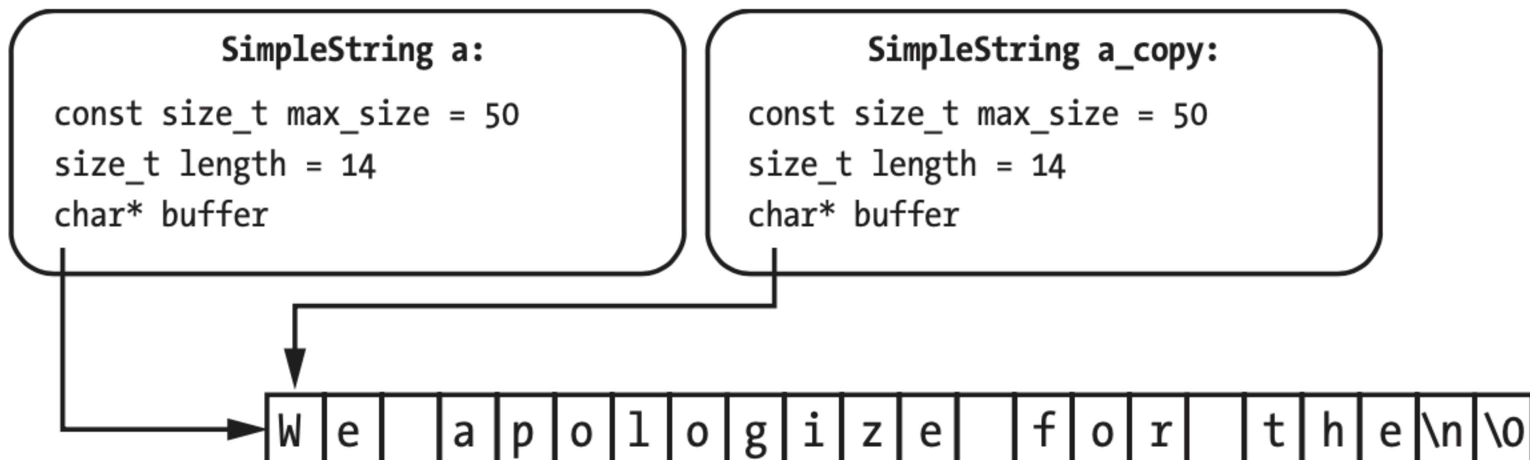
A Problem

- 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)



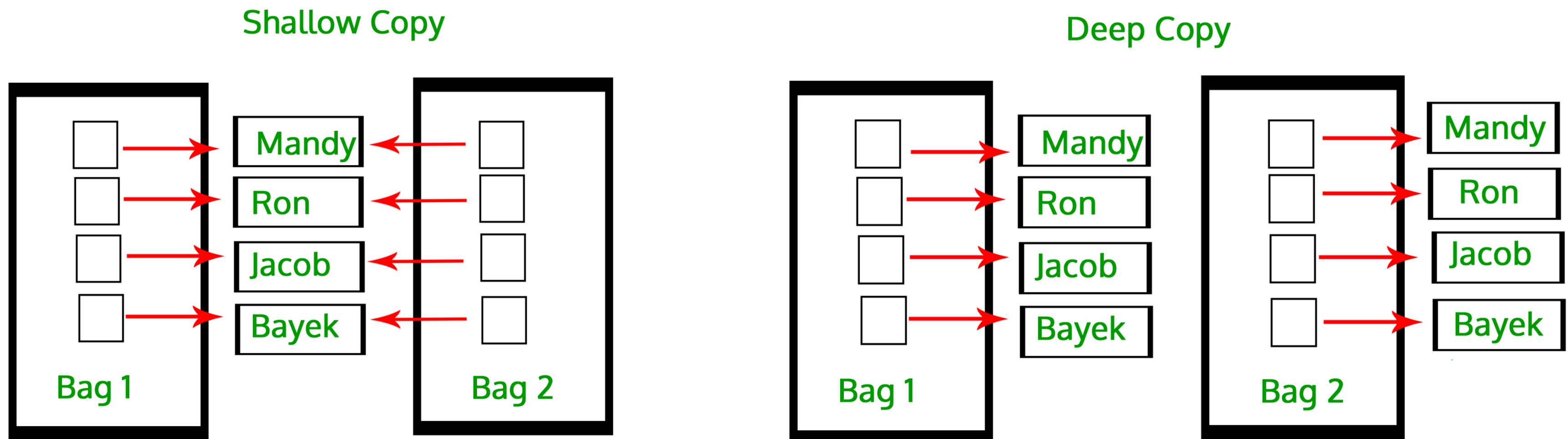
A Problem

- This can be very dangerous!
  - When the remaining object is destructed, buffer will be freed again, a *double free*
    - See: https://sensepost.com/blog/2017/linux-heap-exploitation-intro-series-riding-free-on-the-heap-double-free-attacks/



Copy Semantics are intended to avoid such situations

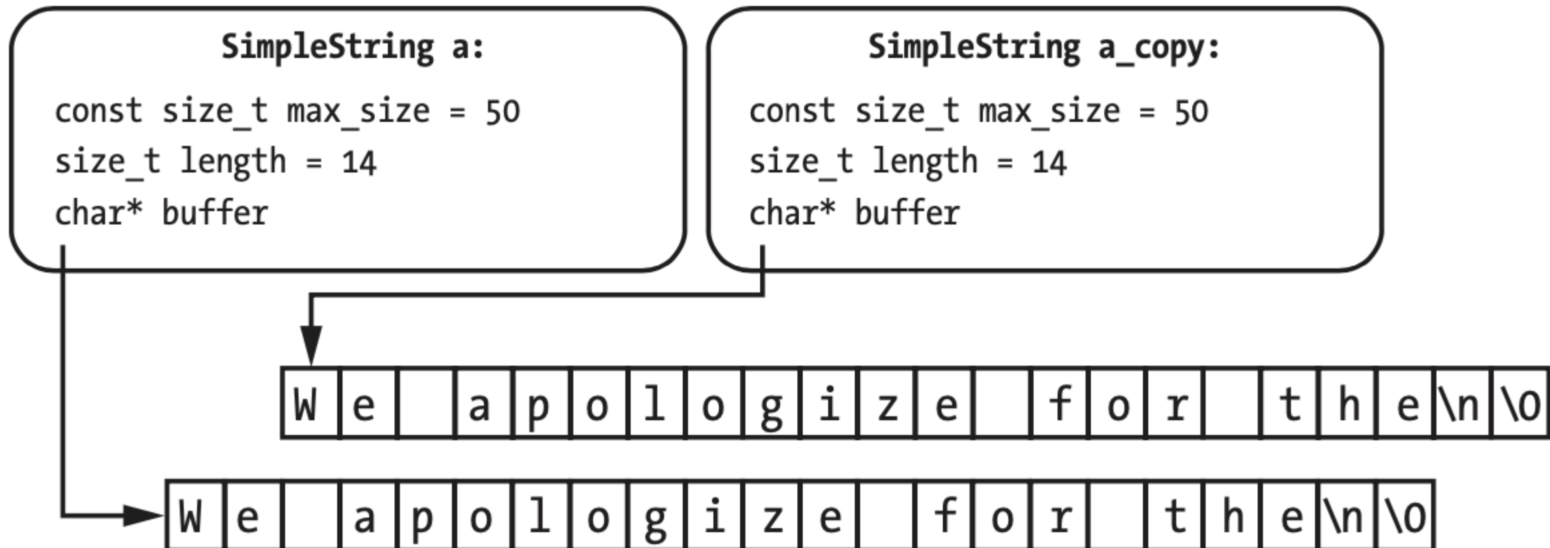
Shallow Copy vs Deep Copy



We want deep copies

Method 1: Copy Constructor

```
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);
}
```



Copy Constructor

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

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

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

Copy Constructor

- Why is `other` passed by reference and not by value?

```
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);
}
```

Copy Constructor

- Why is `other` passed by reference and not by value?
  - Because if it was passed by value, then when it was passed the copy constructor would automatically be called. But calling the copy constructor would require another copy, which would call the copy constructor…

```
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);
}
```

Method 2: Copy Assignment

```
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

```
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;
}
```

Why does copy assignment return a reference to `SimpleString`?

Copy Assignment

```
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;
}
```

Why does copy assignment return a reference to `SimpleString`? Strictly speaking, one doesn't have to. But not doing so precludes assignment chaining (a = b = c;) [which associates from right]

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

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

Repress Generation

- Some objects should not be copied
  - E.g., the object manages a file
  - E.g., objects represents a mutual exclusion lock

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

  - Any attempt to copy results in compiler error

```
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 objects in this state: reassign or destruct

Move Semantics

- This raises a fairly reasonable question: Why would anyone ever want to move an object into another object (without changing any of the data) then get rid of the first?
  - Why not just keep the original object and work with that?

An Aside: Returning Values from Functions

- We often talk about how parameters are passed to functions, but rarely talk about how they are returned!
- So, how are they passed to functions?
  - Pass by value
  - Pass by pointer
  - Pass by reference

An Aside: Returning Values from Functions

- So, how are they returned from functions?
  - The same three ways
- Data just travels in the other direction
- BUT with a big caveat: Local variables go out of scope and are destroyed when the function ends. We need to consider the effect of this!

Thanks: https://www.learncpp.com/cpp-tutorial/returning-values-by-value-reference-and-address/

An Aside: Returning Values from Functions

- Return by value

```
1  int doubleValue(int x)
2  {
3      int value{ x * 2 };
4      return value; // A copy of value will be returned
   here
5  } // value goes out of scope here
```

- Since a copy is passed to caller, no issue with variable going out of scope
- BUT requires a copy that can be expensive for large objects/structs

An Aside: Returning Values from Functions

- Return by pointer

```
1  int* doubleValue(int x)
2  {
3      int value{ x * 2 };
4      return &value; // return value by address here
5  } // value destroyed here
```

- Pointer is returned to caller, BUT goes out of scope at function end!
- And accessing this memory via a pointer gives undefined behavior

An Aside: Returning Values from Functions

- Return by pointer

```
1  int* allocateArray(int size)
2  {
3      return new int[size];
4  }
```

- An easy fix: return the address of memory that has been dynamically allocated! It does not go out of scope at function end!
- But there is another problem. What?

An Aside: Returning Values from Functions

```
1   int* allocateArray(int size)
2   {
3       return new int[size];
4   }
5
6   int main()
7   {
8       int *array{ allocateArray(25) };
9
10      // do stuff with array
11
12      delete[] array;
13      return 0;
14  }
```

- But there is another problem. What?
  - The caller has to deallocate the memory!

An Aside: Returning Values from Functions

- But there is another problem. What?
  - The caller has to deallocate the memory!
- In general, allocating and deallocating memory in different functions can be problematic
- Manually allocating and deallocating memory can make it difficult to know who is responsible for deallocation
  - Or whether the resource needs to be deleted at all

An Aside: Returning Values from Functions

- But there is another problem. What?
  - The caller has to deallocate the memory!
- This may not seem like a big deal.
  - But if you have complex large scale software that does a lot of this, keeping track of who deletes what can be a nightmare

An Aside: Returning Values from Functions

- Return by reference

```
1  int& doubleValue(int x)
2  {
3      int value{ x * 2 };
4      return value; // return a reference to value here
   } // value is destroyed here
5
```

- This returns a reference to memory whose lifetime ends when the function ends
  - So this is a reference to garbage
  - Fortunately the compiler will usually catch this

Move Semantics

- So hopefully at this point you see why you might want to transfer the contents one object to another and then kill off the first
  - Of in some cases, know that the first is going to be killed automatically
- Again, you could make a copy. But that can be inefficient
  - Why copy a large array when you can instead transfer ownership of that memory?

Move Semantics

- Move semantics effectively allows *efficiently* returning by value
  - And thus avoiding the issues with pointers and references
- All STL collection classes (and many others) support move semantics
  - So you know that for collection classes (and classes that support move semantics) returning by value will be efficient

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*

Example (Pedagogical)

- Consider:

```
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;
};
```

Example

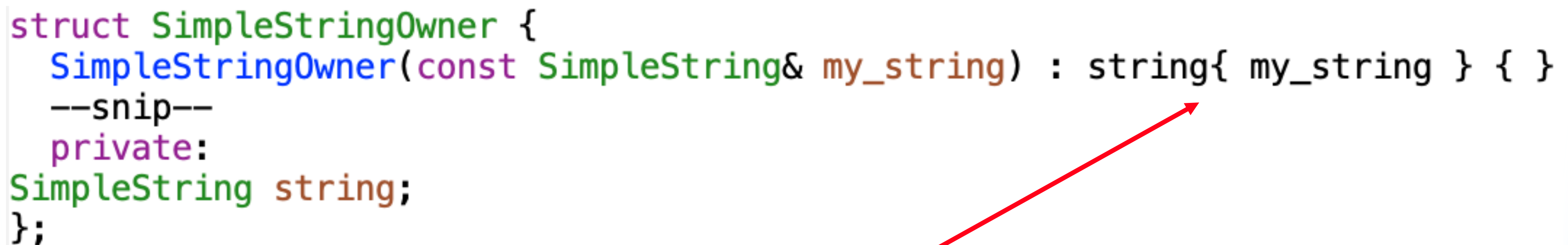
- Suppose you want to move a SimpleString into a SimpleStringOwner as follows:

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

Assumes `SimpleString` has a copy constructor. So…

Example

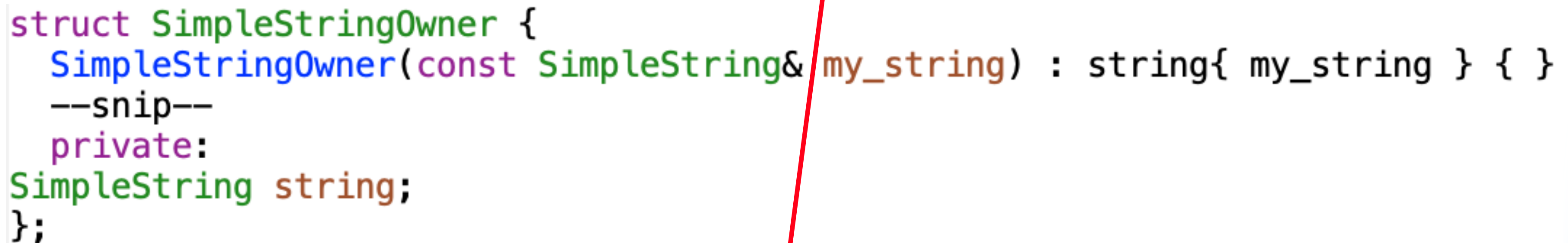
```
--snip--
void own_a_string() {
  SimpleString a{ 50 };
  a.append_line("We apologize for the");
  a.append_line("inconvenience.");
  SimpleStringOwner b{ a };
--snip--
}
```



Assumes `SimpleString` has a copy constructor. So… we use it.

Example

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

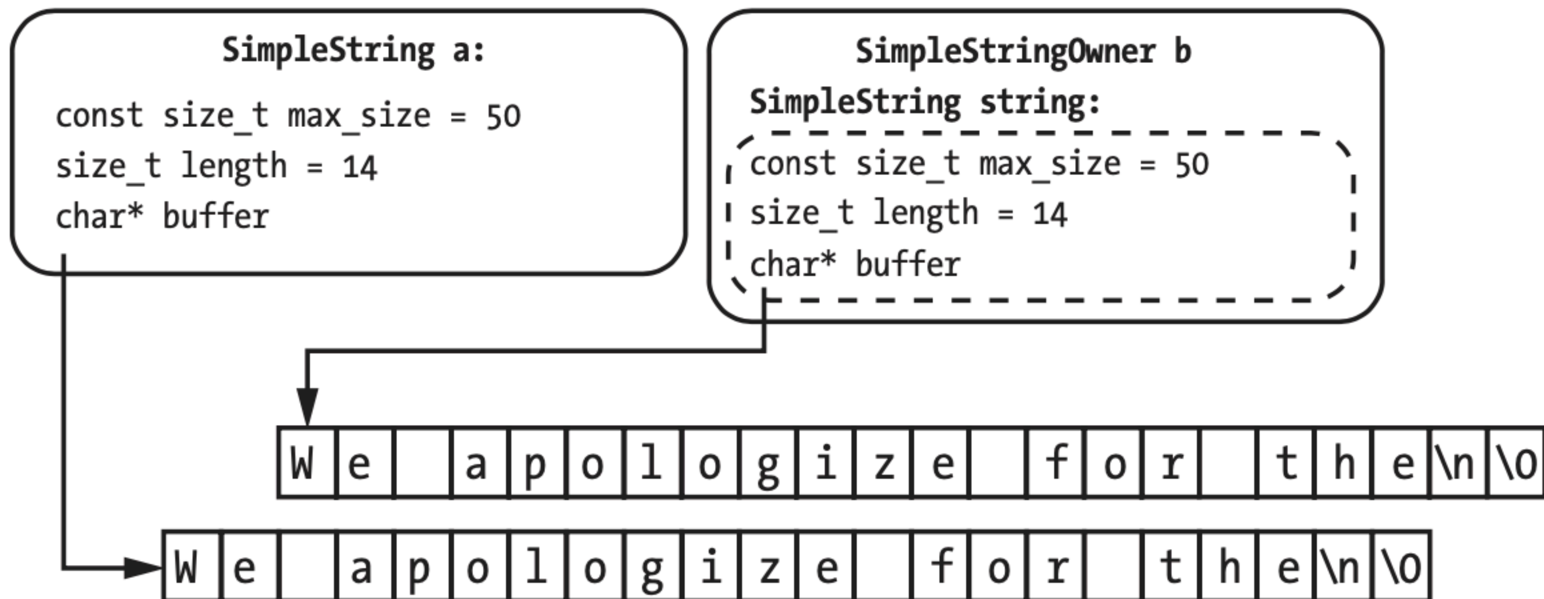


```
struct SimpleStringOwner {
  SimpleStringOwner(const SimpleString& my_string) : string{ my_string } { }
  --snip--
  private:
SimpleString string;
};
```

Hidden waste: Caller never uses the pointed to object again after constructing `string` (in this example, `a` is never used again)

Why Move?

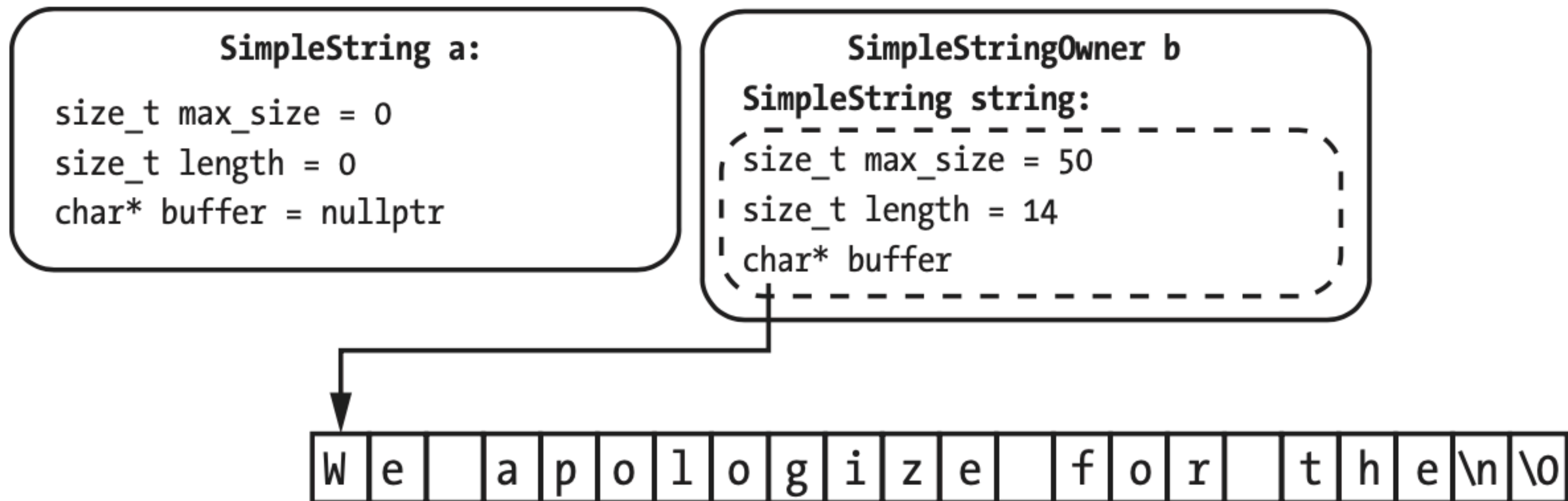
Hidden waste: Caller never uses the pointed to object again after constructing `string`



Better to move the "guts" of SimpleString `a` into the string field of SimpleStringOwner `b`

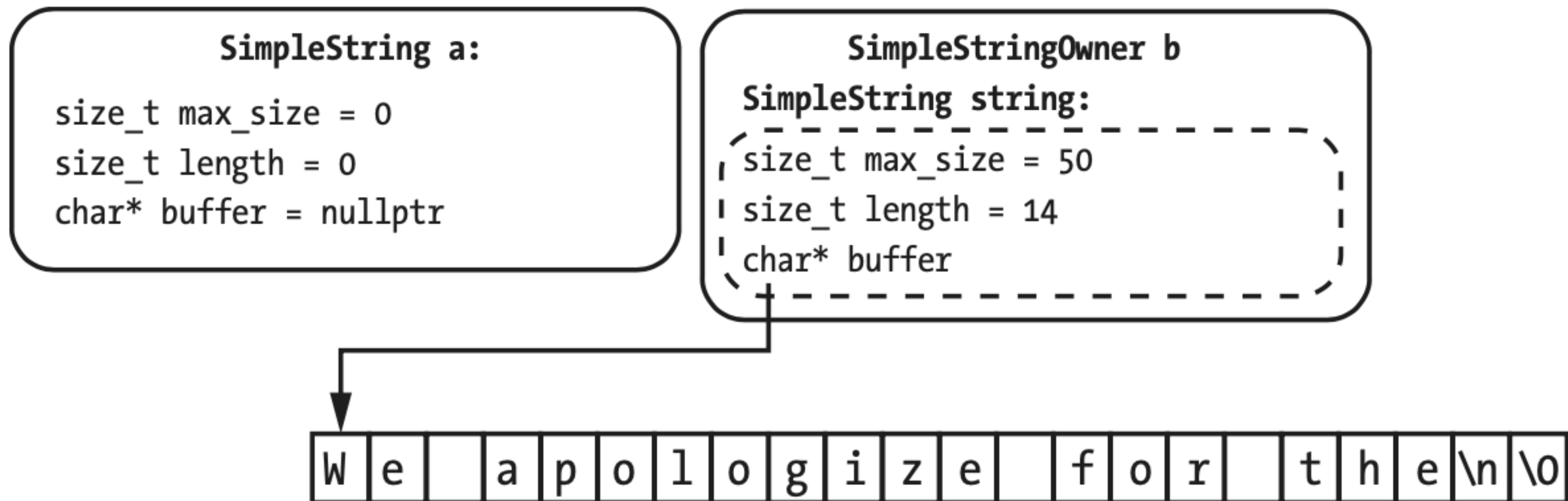
Why Move?

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



Why Move?

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



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:

```
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

lvalue 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 `&&`

lvalue and rvalue References

- Scott Meyer:
- rvalues indicate objects eligible for move operations
- In concept:
  - rvalues correspond to temporary objects returned from functions
  - lvalues correspond to objects you can refer to by name or following a pointer or lvalue reference

lvalue and rvalue References

- More Scott Meyer:
- Conceptually: Can you (in your program) take the address of an expression
  - If yes, it’s probably an lvalue
  - If not, usually an rvalue

lvalue and rvalue References

- 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

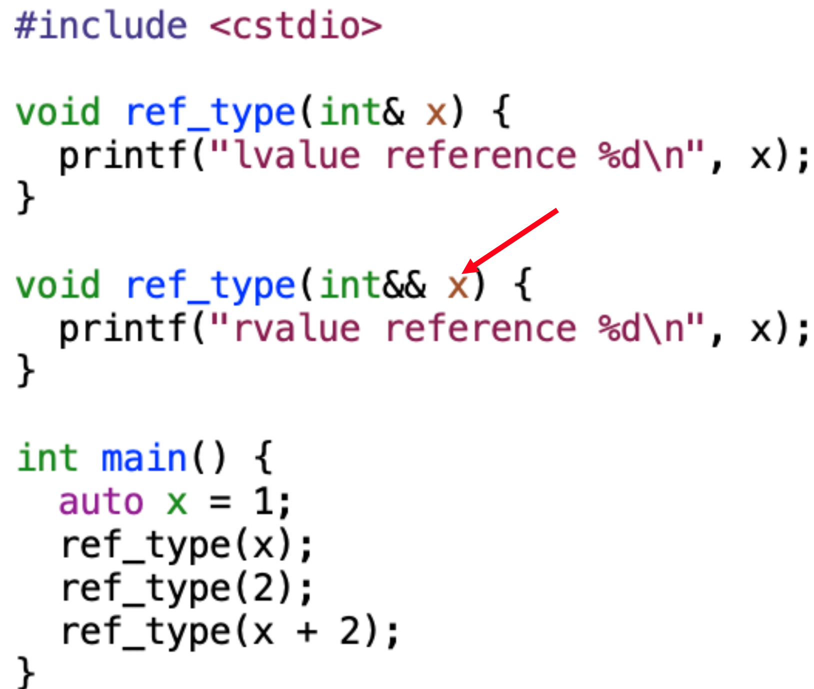
lvalue and rvalue References

```
#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
```

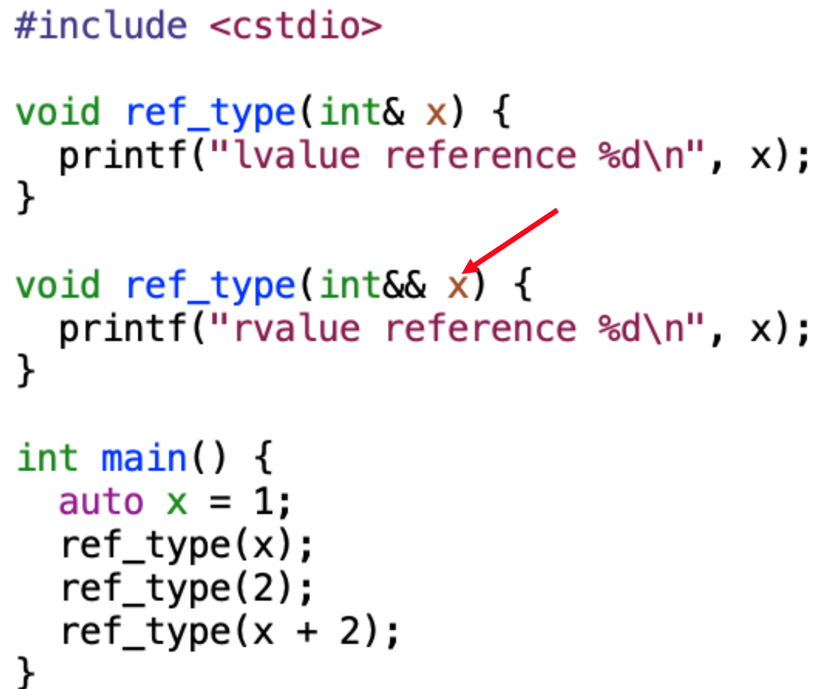
lvalue and rvalue References



```
#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);
}
```

The details of lvalues and rvalues can be tricky and subtle.
Question: what is x? (rvalue or lvalue?)

lvalue and rvalue References



```
#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);
}
```

lvalues and rvalues can be tricky and subtle.

Solution: `x` has a name, so an lvalue (it is a variable of type rvalue reference)

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
  - In fact, there is not a single byte of code associated with std::move! (It simply informs the compiler.)

std::move

```
#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

```
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

```
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 (usually, but not always) much less expensive than copy constructor
  - In some cases it can even be more expensive

Move Construction

- Move constructor is designed to not throw exception so you should always mark it `noexcept`
  - Compiler cannot use exception throwing move constructors and will use copy constructor instead
- Why? If an exception is thrown during the move, then data being processed can be lost
  - Not an issue with copy, as original is unchanged

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`

Move Assignment

```
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`

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

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

Move Assignment

```
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 in 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

Compiler-Generated Methods

- 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!**