MTH 4300/4299: Programming and Computer Science II

Lecture 14: Rule of Five (Part II — Move Semantics)

Jaime Abbariao

▒▒ Recap: What We Have So Far

From Lecture 13, we implemented three of the Rule of Five:

#	Function	What it does
1	Destructor	Frees memory when object dies
2	Copy constructor	Deep copies into a new object
3	Copy assignment	Frees old memory, then deep copies into an existing object

Today: #4 and #5 — the move operations.

These let us steal resources instead of copying them.

▒▒ The Problem: Unnecessary Copies

```
DynamicArray create_big_array() {
  DynamicArray arr(1000000);
  for (int i = 0; i < 1000000; i++) {
    arr.push_back(i);
  }
  return arr;  // returns a COPY of arr
}

int main() {
  DynamicArray result = create_big_array();
  return 0;
}
```

Without move semantics, `result` gets a deep copy of `arr`:

- Allocate 1,000,000 ints for `result`
- Copy all 1,000,000 elements one by one
- Destroy the original `arr`

We copied 1 million integers just to throw the original away on the next line.

▒▒ What if We Could Just... Steal?

Instead of copying, what if `result` just took `arr`'s pointer?

```
BEFORE (copy):
arr.data ──→ [0, 1, 2, ..., 999999]    ← original
result.data ──→ [0, 1, 2, ..., 999999]  ← brand new copy (expensive!)
Then arr is destroyed.

AFTER (move — what we want):
arr.data ──→ nullptr                     ← emptied out
result.data ──→ [0, 1, 2, ..., 999999]  ← stole arr's pointer (free!)
Then arr is destroyed (nullptr — safe).
```

The returned `arr` is temporary — it's about to die anyway. Why not just take its stuff?

This is the core idea of move semantics.

▒▒ Lvalues vs Rvalues

To understand move semantics, we need to distinguish two kinds of expressions:

Expression	Type	Why?
`x` (a variable)	lvalue	Has a name, persists
`42`	rvalue	Temporary literal
`x + 3`	rvalue	Temporary result
`create_array()`	rvalue	Temporary return value
`DynamicArray(5)`	rvalue	Temporary unnamed object
`arr[i]`	lvalue	Refers to a persistent location

Simple test: can you take its address with `&`?

- `&x` — yes, it's an lvalue
- `&(x + 3)` — no (won't compile), it's an rvalue

Lvalue = "left value" — can appear on the left side of `=`

Rvalue = "right value" — temporary, about to disappear

▒▒▒ Rvalue References (&&)

You already know `&` means reference. `&&` means rvalue reference — a reference that binds to temporaries.

This is NOT logical AND. When `&&` appears in a type, it means rvalue reference.

Where rvalue references actually matter — as function parameters:

```
// This catches lvalues (named variables)
void process(DynamicArray& arr);

// This catches rvalues (temporaries)
void process(DynamicArray&& arr);
```

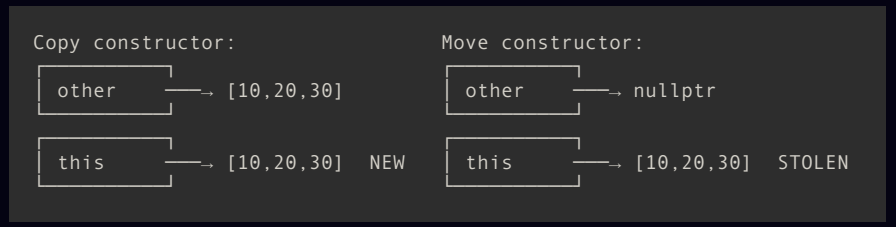
When you call `process(create_array())`, the compiler picks the `&&` version because the argument is a temporary.

This is exactly how the move constructor works — it's a constructor whose parameter is `&&`.

▒▒ Move Constructor: The Idea

Copy constructor (from L13): allocate new memory, copy everything.

Move constructor (new): steal the pointer, null out the source.



Copy: O(n) — allocates memory, copies every element.

Move: O(1) — just swaps a few pointers and integers.

▒▒ Move Constructor: DynamicArray Implementation

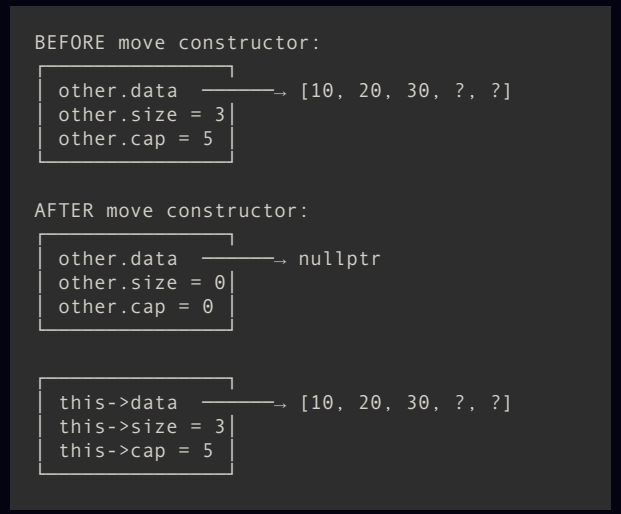
```
// Move constructor — steal resources from a temporary
DynamicArray(DynamicArray&& other) {
  // Step 1: steal the source's resources
  data = other.data;
  size = other.size;
  capacity = other.capacity;

  // Step 2: null out the source so its destructor is safe
  other.data = nullptr;
  other.size = 0;
  other.capacity = 0;
}
```

Key points:

- Parameter is `DynamicArray&&` — only binds to temporaries (rvalues)
- No `new`, no loop — we just take the pointer
- We must set `other.data = nullptr` — otherwise `other`'s destructor would `delete[]` our memory

▒▒ Move Constructor: Visual



The source is left in a valid but empty state. Its destructor will call `delete[] nullptr`, which is safe (it does nothing).

▒▒ Move Assignment Operator

Same relationship as copy constructor vs copy assignment from Lecture 13:

- Move constructor: target is a new object (no old memory to free)
- Move assignment: target already exists (must free its old memory first)

The three extra steps (same as copy assignment):

1. Self-assignment check — `if (this == &other) return *this;`
2. Free old memory — `delete[] data;`
3. Return `*this` — for chained assignment

But instead of deep copying, we steal.

▒▒ Move Assignment: DynamicArray Implementation

```
// Move assignment operator — steal resources from a temporary
DynamicArray& operator=(DynamicArray&& other) {
  // Step 1: self-assignment check
  if (this == &other) {
    return *this;
  }

  // Step 2: free our old memory
  delete[] data;

  // Step 3: steal the source's resources
  data = other.data;
  size = other.size;
  capacity = other.capacity;

  // Step 4: null out the source
  other.data = nullptr;
  other.size = 0;
  other.capacity = 0;

  // Step 5: return *this (for chaining)
  return *this;
}
```

Self-move-assignment (`a = std::move(a)`) is rare in practice, but we check anyway for safety.

▒▒▒ `std::move` — Forcing a Move

The compiler automatically moves from temporaries (rvalues).

But what if you want to move from a named variable (lvalue)?

```
DynamicArray a(5);
a.push_back(10);
a.push_back(20);

// I'm done with a — let b steal its resources
DynamicArray b = std::move(a);  // move constructor (not copy!)
```

`std::move(a)` casts `a` to an rvalue reference. It does not move anything by itself — it just says "treat `a` as a temporary."

After `std::move`: `a` is in a valid but empty state. Do not read from it — only assign to it or let it be destroyed.

▒▒▒ When Would I Use `std::move`?

Use `std::move` when you have a large object and you know you won't need it anymore:

```cpp
// You built a big vector and want to store it in a class
std::vector<int> build_data() {
  std::vector<int> v;
  // ... fill v with 1 million elements ...
  return v;
}

int main() {
  std::vector<int> data = build_data();

  // I'm done with data — pass it to the container
  DataStore store;
  store.set_data(std::move(data));
  // data is now empty — don't use it!
  return 0;
}
```

Rule of thumb: use `std::move` when transferring ownership. If you still need the object, don't move from it.

When is Each Function Called?

Code	Which function?	Why?
`DynamicArray b = a;`	Copy constructor	`a` is an lvalue
`DynamicArray c(a);`	Copy constructor	`a` is an lvalue
`d = a;`	Copy assignment	`d` exists, `a` is an lvalue
`DynamicArray e = std::move(a);`	Move constructor	`std::move(a)` is an rvalue
`d = std::move(a);`	Move assignment	`d` exists, rvalue
`DynamicArray f = create();`	Move constructor*	Return value is an rvalue
`d = create();`	Move assignment	`d` exists, rvalue

*The compiler may optimize `f = create()` to construct directly in place (RVO — Return Value Optimization). In that case, neither copy nor move constructor is called. Use `std::move` for guaranteed moves.

```
class DynamicArray {
private:
  int* data;
  int size;
  int capacity;
public:
  // Constructor
  DynamicArray(int cap) : size(0), capacity(cap) {
    data = new int[capacity];
  }

  // 1. Destructor
  ~DynamicArray() {
    delete[] data;
  }
```

```
// 2. Copy constructor
DynamicArray(const DynamicArray& other)
  : size(other.size), capacity(other.capacity) {
  data = new int[capacity];
  for (int i = 0; i < size; i++) data[i] = other.data[i];
}

// 3. Copy assignment
DynamicArray& operator=(const DynamicArray& other) {
  if (this == &other) return *this;
  delete[] data;
  size = other.size;
  capacity = other.capacity;
  data = new int[capacity];
  for (int i = 0; i < size; i++) data[i] = other.data[i];
  return *this;
}
```

```
  // 4. Move constructor
  DynamicArray(DynamicArray&& other)
    : data(other.data), size(other.size),
      capacity(other.capacity) {
    other.data = nullptr;
    other.size = 0;
    other.capacity = 0;
  }

  // 5. Move assignment
  DynamicArray& operator=(DynamicArray&& other) {
    if (this == &other) return *this;
    delete[] data;
    data = other.data;
    size = other.size;
    capacity = other.capacity;
    other.data = nullptr;
    other.size = 0;
    other.capacity = 0;
    return *this;
  }
};
```

▒▒ Summary

- Copy = duplicate resources (safe but slow for big data)
- Move = transfer ownership (fast — O(1) pointer swap)
- After a move, the source is valid but empty (safe to destroy)

Operation	Allocates?	Copies data?	Speed
Copy constructor	Yes	Yes	O(n)
Copy assignment	Yes	Yes	O(n)
Move constructor	No	No	O(1)
Move assignment	No	No	O(1)

Rule of Five: if you define any one of destructor, copy constructor, copy assignment, move constructor, or move assignment — define all five.

▓▓ Lab: Add Move Semantics to StringHolder

In Lecture 13's lab, you added copy constructor and copy assignment to `StringHolder`. Now add the move constructor and move assignment operator.

```
class StringHolder {
private:
  char* data;
  int len;
public:
  StringHolder(const char* str) {
    len = std::strlen(str);
    data = new char[len + 1];
    std::strcpy(data, str);
  }
  ~StringHolder() { delete[] data; }

  // Copy constructor (from L13)
  // Copy assignment (from L13)

  // TODO: Move constructor
  // TODO: Move assignment operator

  int length() const { return len; }
  void print() const {
    if (data) { std::cout << data << std::endl; }
    else { std::cout << "(empty)" << std::endl; }
  }
};
```

▓▓ Lab: Test Code

```
StringHolder create_greeting(const char* name) {
  StringHolder result(name);
  return result;  // may trigger move constructor
}

int main() {
  // Test move constructor via std::move
  StringHolder a("Hello");
  StringHolder b = std::move(a);
  std::cout << "a: "; a.print();   // a: (empty)
  std::cout << "b: "; b.print();   // b: Hello

  // Test move assignment
  StringHolder c("World");
  c = std::move(b);
  std::cout << "b: "; b.print();   // b: (empty)
  std::cout << "c: "; c.print();   // c: Hello

  // Test move from function return
  StringHolder d = create_greeting("Baruch");
  std::cout << "d: "; d.print();   // d: Baruch

  return 0;
}
```

Compile: `g++ -std=c++17 -fno-elide-constructors -o out stringholder.cpp`

(The `-fno-elide-constructors` flag disables RVO so you can see the move constructor being called.)

▒▒ Lab: Expected Output

```
a: (empty)
b: Hello
b: (empty)
c: Hello
d: Baruch
```

Hints:

- Move constructor: same as copy constructor, but steal instead of allocate+copy. Set source's `data` to `nullptr` and `len` to `0`.
- Move assignment: same as copy assignment, but steal instead of allocate+copy. Don't forget: self-assignment check, free old memory, return `*this`.
- Use `std::cstring` for `std::strlen` and `std::strcpy` (same as the constructor).