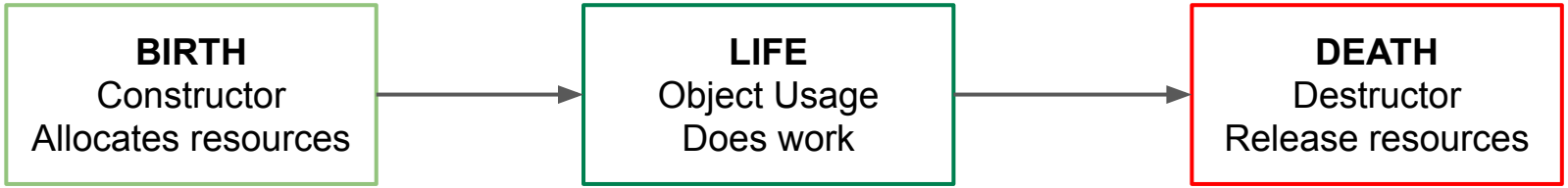
Lec-8: C++ Destructors, Copy, Move, Assignment Constructors

Bernard Nongpoh

What is Destructor?

- A destructor is a special member function that is automatically called when an object is destroyed.

Object Lifecycle



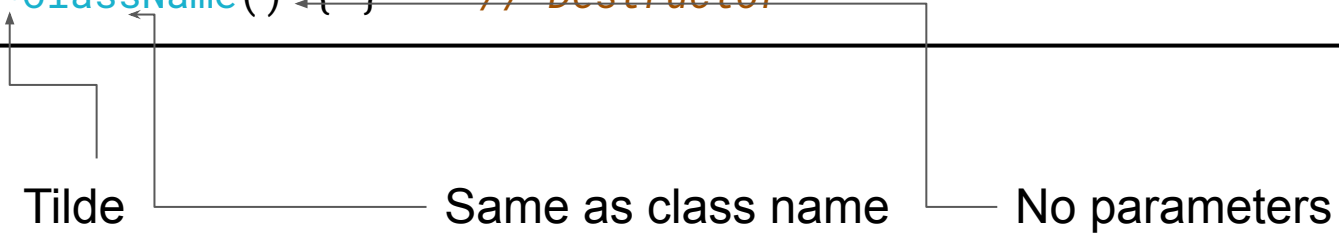
Destructor

- Same name as the class, preceded by tilde ~
- Called automatically - you don't call it yourself
- Cleans up resources before the object is removed from memory

Destructor

- Destructor Syntax

```
~ClassName() { }     // Destructor
```




```
class MyClass {
public:
    MyClass() { }     // Constructor - called at birth
    ~MyClass() { }    // Destructor - called at death
};
```

- Only one destructor per class

Source

Destructor Example

```
class Student {
public:
    // Constructor
    Student(string name) {
        studentName = name;
        cout << "Student created: " << studentName << endl;
    }

    // Destructor - note: NO parameters, NO return type
    ~Student() {
        cout << "Student destroyed: " << studentName << endl;
    }

private:
    string studentName;
};
```

Source

Destructor Trigger Events

1. **Out of Scope** Local object reaches end of block{}	2. **`delete` Operator** Explicit deleting heap object
3. **Program ends** Global/static objects destroyed	4. **Temporary dies** Temporary object lifetime ends

Source

Example 1: Local Object Goes Out of Scope

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

class Demo {
public:
    Demo()  { cout << "Constructor called" << endl; }
    ~Demo() { cout << "Destructor called" << endl; }
};

int main() {
    cout << "--- Entering block ---" << endl;
    {
        Demo obj;  // Constructor called here
        cout << "Inside block" << endl;
    }  // <-- Destructor called here (obj goes out of scope)
    cout << "--- After block ---" << endl;
    return 0;
}
```

Source

Example 1: Local Object Goes Out of Scope

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

class Demo {
public:
    Demo()  { co                    << endl; }
    ~Demo() { co                    endl; }
};

int main() {
    cout << "--- Entering block ---" << endl;
    {
        Demo obj;  // Constructor called here
        cout << "Inside block" << endl;
    }  // <-- Destructor called here (obj goes out of scope)
    cout << "--- After block ---" << endl;
    return 0;
}
```

```
--- Entering block ---
Constructor called
Inside block
Destructor called
--- After block ---
```

Source

Example 2: Using `delete` Operator

```
int main() {
    Demo* ptr = new Demo();  // Constructor called
    cout << "Using object..." << endl;
    delete ptr;              // Destructor called explicitly
    cout << "After delete" << endl;
    return 0;
}
```

Source

Stack vs Heap - Destructor Behavior

STACK Memory

- Automatic Cleanup
- Destructor called automatically when scope ends

- Fast Allocation
- Memory managed by compiler

```
Demo obj; // Stack
```

HEAP Memory

Manual Cleanup

- You must call delete or memory leaks

- Flexible size
- Allocate at runtime

```
Demo *p = new Demo();
```

Source

Example 3

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

class Box {
    int id;
public:
    Box(int i) : id(i) {
        cout << "Box " << id << " created" << endl;
    }
    ~Box() {
        cout << "Box " << id << " destroyed" << endl;
    }
};

int main() {
    Box stackBox(1);           // Stack allocation

    Box* heapBox = new Box(2); // Heap allocation

    cout << "--- End of main ---" << endl;

    delete heapBox;            // Must delete manually!

    return 0;
}  // stackBox destructor called automatically
```

Source

Example 3

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

class Box {
    int id;
public:
    Box(int i) : id(i) {
        cout << "Box " << id << " created" << endl;
    }
    ~Box() {
        cout << "Box " << 
    }
};

int main() {
    Box stackBox(1);

    Box* heapBox = new Bo

    cout << "--- End of main ---" << endl;

    delete heapBox;             // Must delete manually!

    return 0;
}  // stackBox destructor called automatically
```

```
Box 1 created
Box 2 created
--- End of main ---
Box 2 destroyed
Box 1 destroyed
```

Source

Why Do We Need Custom Destructors?

- 💾 Dynamic Memory - Used new/new[ ]
- 📁 File Handles - Opened files need closing
- 🔌 Network Connections - Sockets need closing
- 🔒 Locks/Mutexes - Must be released

Source

BAD: No Custom Destructor (Memory Leak!)

```
class BadString {
    char* text;
public:
    BadString(const char* str) {
        text = new char[strlen(str) + 1];
        strcpy(text, str);
    }
    // No destructor! Memory is LEAKED when object dies
};
```

Source

With Custom Destructor

```
class GoodString {
    char* text;
public:
    GoodString(const char* str) {
        text = new char[strlen(str) + 1];
        strcpy(text, str);
        cout << "Allocated memory for: " << text << endl;
    }

    ~GoodString() {
        cout << "Freeing memory for: " << text << endl;
        delete[] text;  // Clean up dynamic memory!
    }
};

int main() {
    GoodString greeting("Hello World");
    // Memory automatically freed when greeting goes out of scope
    return 0;
}
```

Source

The Default Destructor

- Default is **ENOUGH**
- Class has only basic types
- Class has only objects that manage themselves
- No pointers to dynamically allocated memory

- Need **CUSTOM**
- Class uses **new / new [ ]**
- Class opens files/sockets
- Class acquires any resource

Source

Default Destructor is Fine Here

```
class Point {
    int x, y;  // Basic types - automatically cleaned up
public:
    Point(int a, int b) : x(a), y(b) { }
    // No destructor needed - compiler provides default
};

class Person {
    string name;    // string cleans up itself
    int age;        // basic type
    vector<int> scores; // vector cleans up itself
public:
    Person(string n, int a) : name(n), age(a) { }
    // No destructor needed!
};
```

Source

Custom Destructor Required Here

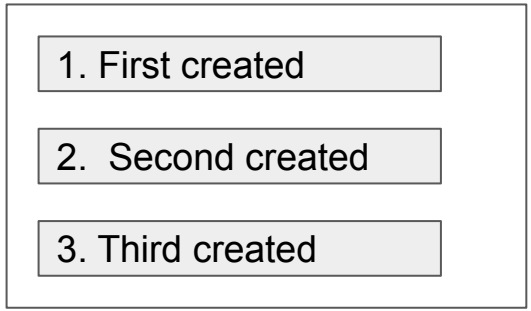
```
class DynamicArray {
    int* data;
    int size;
public:
    DynamicArray(int s) : size(s) {
        data = new int[size];  // Dynamic allocation!
    }

    ~DynamicArray() {
        delete[] data;  // MUST clean up!
    }
};
```

Source

Order of Destruction

Destruction = **REVERSE** of Construction

Construction order



Destruction order

Last In, First Out (LIFO)

Source

Example: Order of Destruction

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

class Object {
    int id;
public:
    Object(int i) : id(i) {
        cout << "Object " << id << " constructed" << endl;
    }
    ~Object() {
        cout << "Object " << id << " destroyed" << endl;
    }
};

int main() {
    Object a(1);  // First constructed
    Object b(2);  // Second constructed
    Object c(3);  // Third constructed

    cout << "--- End of main ---" << endl;
    return 0;
}  // Destruction happens in REVERSE order
```

Source

Example: Order of Destruction

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

class Object {
    int id;
public:
    Object(int i) : id(i) {
        cout << "Object " << id << " constructed" << end
    }
    ~Object() {
        cout << "Object " << id << " destroyed" << endl;
    }
};

int main() {
    Object a(1);  // First constructed
    Object b(2);  // Second constructed
    Object c(3);  // Third constructed

    cout << "--- End of main ---" << endl;
    return 0;
}  // Destruction happens in REVERSE order
```

```
Object 1 constructed
Object 2 constructed
Object 3 constructed
--- End of main ---
Object 3 destroyed
Object 2 destroyed
Object 1 destroyed
```

Source

Member Destruction Sequence

1. **Destructor body executes:** your cleanup code runs first
2. **Member objects destroyed:** In reverse order of declaration
3. **Base class destructor called:** if class inherits from another

Source

Example: Order of Destruction

```
class Member {
    string name;
public:
    Member(string n) : name(n) {
        cout << "  Member " << name << " constructed" <<
endl;
    }
    ~Member() {
        cout << "  Member " << name << " destroyed" <<
endl;
    }
};

class Container {
    Member first;   // Constructed 1st, destroyed 2nd
    Member second;  // Constructed 2nd, destroyed 1st
public:
    Container() : first("A"), second("B") {
        cout << "Container constructed" << endl;
    }
    ~Container() {
        cout << "Container destructor body" << endl;
        // After this, members destroyed in REVERSE order
    }
};
```

```
int main() {
    Container c;
    cout << "--- End of main ---" << endl;
    return 0;
}
```

```
  Member A constructed
  Member B constructed
Container constructed
--- End of main ---
Container destructor body
  Member B destroyed
  Member A destroyed
```

Source

Virtual Destructors

If a class has **virtual functions** OR will be inherited from, make destructor virtual

- Without virtual

  Only base destructor called = **MEMORY LEAK**

- With virtual

  Correct constructor called

Virtual functions are member functions whose behavior can be overridden in derived classes

Source

The Problem: Non-Virtual Destructor

```
class Base {
public:
    ~Base() {  // NOT virtual - PROBLEM!
        cout << "Base destructor" << endl;
    }
};

class Derived : public Base {
    int* data;
public:
    Derived() {
        data = new int[100];
        cout << "Derived allocated memory" << endl;
    }
    ~Derived() {
        delete[] data;
        cout << "Derived destructor - freed memory"
<< endl;
    }
};
```

```
int main() {
    Base* ptr = new Derived();  // Pointer
to base, object is derived
    delete ptr;  // ONLY Base destructor
called! Memory leaked!
    return 0;
}
```

```
Derived allocated memory
Base destructor
```

Source

The Problem: Non-Virtual Destructor

```
class Base {
public:
    ~Base() {  // NOT virtual - PROBLEM!
        cout << "Base destructor" << endl;
    }
};

class Derived : public Base {
    int* data;
public:
    Derived() {
        data = new int[100];
        cout << "Derived allocated memory" << endl;
    }
    ~Derived() {
        delete[] data;
        cout << "Derived destructor - freed memory"
<< endl;
    }
};
```

```
int main() {
    Base* ptr = new Derived();  //
Pointer to base, object is derived
    delete ptr;  // ONLY Base
destructor called! Memory leaked!
    return 0;
}
```

```
Derived allocated memory
Base destructor
```

- Derived destructor never called!
- Memory leaked!

Source

The Solution: Virtual Destructor

```
class Base {
public:
    virtual ~Base() {  // VIRTUAL - correct!
        cout << "Base destructor" << endl;
    }
};

class Derived : public Base {
    int* data;
public:
    Derived() {
        data = new int[100];
        cout << "Derived allocated memory" << endl;
    }
    ~Derived() {
        delete[] data;
        cout << "Derived destructor - freed memory"
<< endl;
    }
};
```

```
int main() {
    Base* ptr = new Derived();
    delete ptr;  // BOTH destructors
called correctly!
    return 0;
}
```

```
Derived allocated memory
Derived destructor - freed memory
Base destructor
```

Source

Destructor Summary

- **No Arguments:** Cannot accept any parameters
- **No Return:** No return type (not even void)
- **One Per Class:** Cannot be overloaded
- **Not Inherited:** Derived classes don’t inherit base destructor
- **Can be Virtual:** Essential for polymorphic base classes

Source

new/delete vs new[ ] / delete [ ]

CORRECT	WRONG
new → delete	new → delete [ ]
new [ ] → delete[ ]	new [ ] → delete

Mismatching causes Undefined Behavior (UB)

Source

The Solution: Virtual Destructor

```
class Base {
public:
    virtual ~Base() {  // VIRTUAL - correct!
        cout << "Base destructor" << endl;
    }
};

class Derived : public Base {
    int* data;
public:
    Derived() {
        data = new int[100];
        cout << "Derived allocated memory" << endl;
    }
    ~Derived() {
        delete[] data;
        cout << "Derived destructor - freed memory"
<< endl;
    }
};
```

```
int main() {
    Base* ptr = new Derived();
    delete ptr;  // BOTH destructors
called correctly!
    return 0;
}
```

```
Derived allocated memory
Derived destructor - freed memory
Base destructor
```

Source

The Solution: Virtual Destructor

```cpp
class Item {
public:
    Item()  { cout << "Item created" << endl; }
    ~Item() { cout << "Item destroyed" << endl; }
};

int main() {
    // Single object
    Item* single = new Item();
    delete single;           // Correct: new → delete

    cout << "---" << endl;

    // Array of objects
    Item* array = new Item[3];
    delete[] array;          // Correct: new[] → delete[]

    return 0;
}
```

```
Item created
Item destroyed
---
Item created
Item created
Item created
Item destroyed
Item destroyed
Item destroyed
```

Source

What Happens with Mismatch (UNDEFINED BEHAVIOR)

```
Item* arr = new Item[5];
delete arr;    // WRONG! Only first destructor called (or crash!)

Item* obj = new Item();
delete[] obj;  // WRONG! Undefined behavior!
```

Source

The Rule of Three

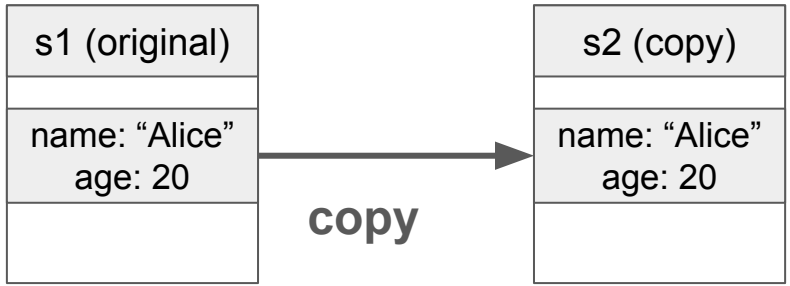
If you define **ANY** of these, you probably need **ALL THREE**		
1. Destructor	**2. Copy Constructor**	**3. Copy Assignment**

Source

Copy Constructor: Making Duplicates

```
// 1. Direct initialization
Student s1("Alice", 20);
// Copy constructor Student
Student s2(s1);
 s3 = s1; // Copy constructor

// 2. Pass by value
void print(Student s); // s is a copy!
print(s1); // Copy constructor

// 3. Return by value
Student create() {
Student temp("Bob", 21);
return temp;
// Copy constructor
}
```

```
Student s2(s1); // Copy s1 to s2
```

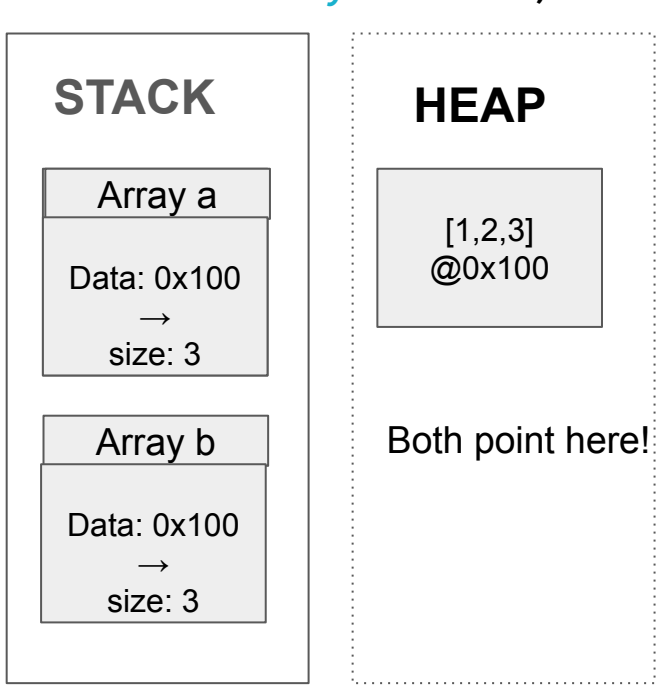


- Both objects are **INDEPENDENT**
- changing s2 won’t affect s1

Source

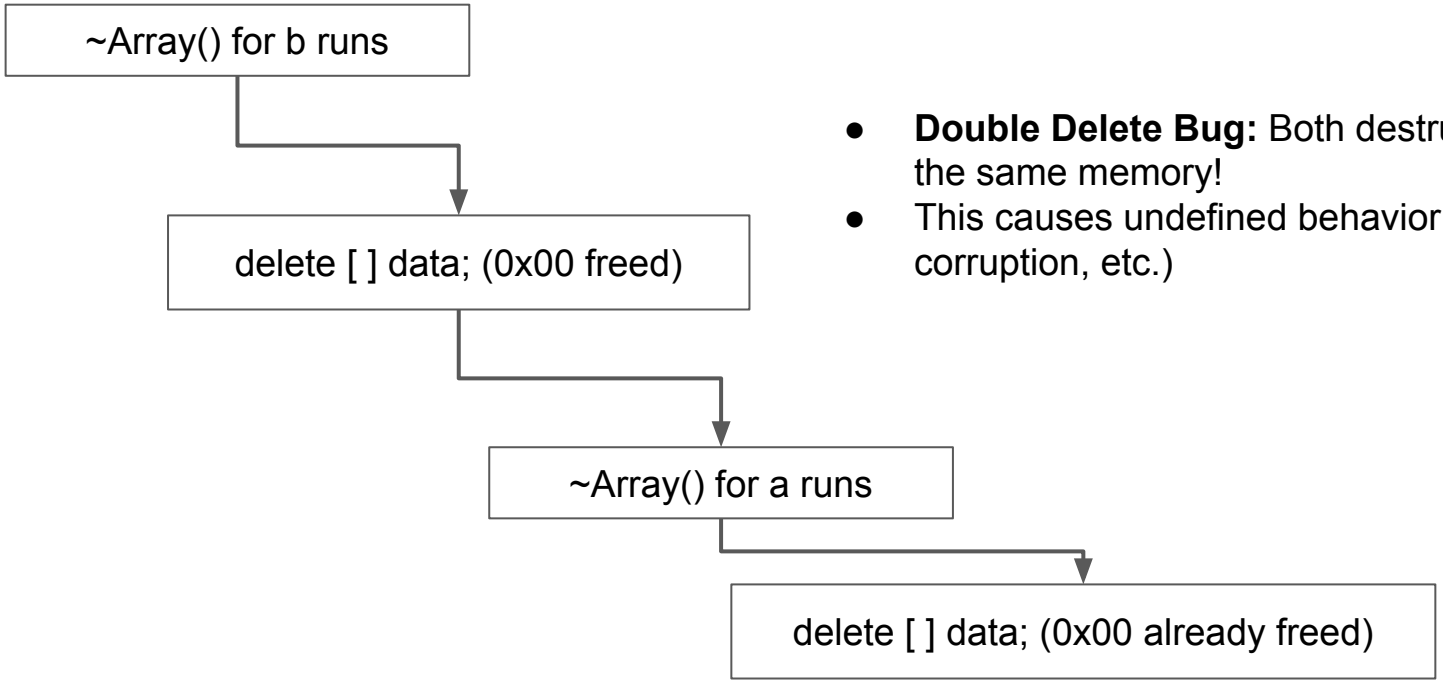
The Shallow Copy Problem

```
class Array {
int* data; int size;
public:
Array(int n) : size(n), data(new
int[n]) {}
~Array() { delete[] data; }
// No copy constructor defined -
compiler creates shallow copy! };
```



Source

When b is destroyed



- **Double Delete Bug:** Both destructors try to free the same memory!
- This causes undefined behavior (crash, corruption, etc.)

CRASH

Source

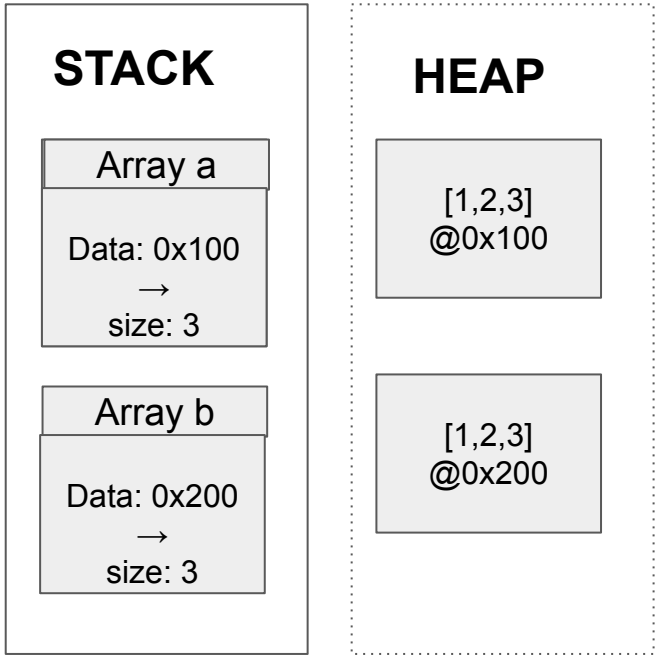
Deep Copy: The Solution

```
class Array {
int* data; int size;
public: Array(int n) : size(n),data(new int[n]) {}
// Deep Copy Constructor

Array(const Array& other) : size(other.size), data(new int[other.size]) // NEW
memory!
{
for(int i = 0; i < size; i++)
data[i] = other.data[i]; // Copy values }
~Array() { delete[] data; }
};
```

Source

Deep Copy: The Solution

AFTER: `Array b = a;`



- Each object has its **own memory**
- Deleting one doesn't affect the other

Rule: If your class has a pointer member that owns memory, you **MUST** write your own copy constructor

Source

Copy Assignment Operator

```
Array a(5); // Constructor
Array b(a); // Copy Constructor (new object)
Array c(3); // Constructor
c = a; // Copy Assignment (existing object)
```

Copy Constructor	Copy Assignment
Create **NEW** object	Modifies **EXISTING** object
No cleanup needed	Must free old data first

Source

Copy Assignment Operator

```
class Array {
// ... members ...

public: // Copy Assignment Operator
    Array& operator=(const Array& other) {
    // 1. Check self-assignment
    if (this == &other) return *this;
    // 2. Free old memory
    delete[] data;
    // 3. Allocate new memory
    size = other.size;
    data = new int[size];
    // 4. Copy data
    for(int i = 0; i < size; i++)
    data[i] = other.data[i];
    return *this;
}
};
```

Source

Copy Assignment Operator

```
class Array {
// ... members ...

public: // Copy Assignment Operator
    Array& operator=(const Array& other) {
    // 1. Check self-assignment
    if (this == &other) return *this;
    // 2. Free old memory
    delete[] data;
    // 3. Allocate new memory
    size = other.size;
    data = new int[size];
    // 4. Copy data
    for(int i = 0; i < size; i++)
    data[i] = other.data[i];
    return *this;
}
};
```

- **Never forget self-assignment check**
- a = a; would delete data before copying it

Source

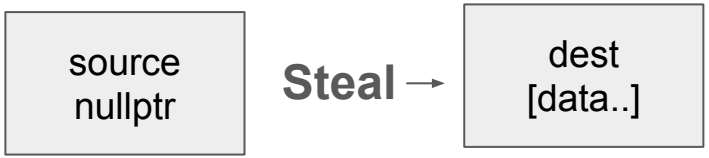
Move Semantics: Why Copy Isn't Always Best

COPY = ***Photocopy a Book***, page by page, slow and expensive
MOVE = ***Hand Over the Book***, Instant, but you no longer have it

```
vector<int> createBigVector() {
vector<int> v(1000000); // 1 million ints
// ... fill vector ...
return v; // Copy 1 million ints?
}

vector<int> result = createBigVector();
```

Without move: Return copies 1 million integers. The original v is about to be destroyed anyway

Both have data (expensive)



Only dest has data (just pointer swap!)

Source

Move Constructor

```
class Array {
int* data; int size;
public: // Move Constructor
Array(Array&& other) // && = rvalue reference
: data(other.data), // Steal the pointer!
size(other.size) {
other.data = nullptr;
// Leave source empty
other.size = 0;
} }; // Usage: Array a(1000);
Array b(std::move(a)); // Move, don't copy! // Now 'a' is empty, 'b' has the
data
```

Source

Rvalue Reference (&&) in C++

- Introduced in **C++11**
- && means **rvalue reference**
- Binds to **temporary objects** or objects marked with **std::move**
- Enables **move semantics** (steal resources instead of copying)

& → borrow

&& → steal

`std::move` → permission to steal

```
int&& a = 10;          // OK
int&& a = x;           // ERROR
int&& a = std::move(x); // OK
```

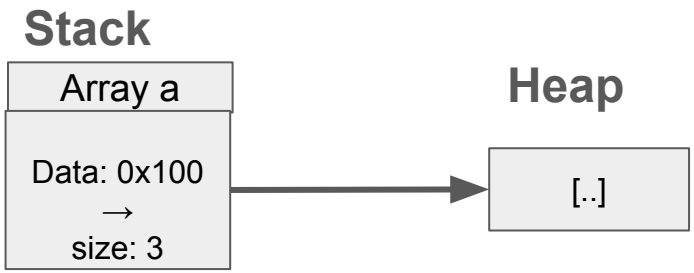
Source

Move Assignment

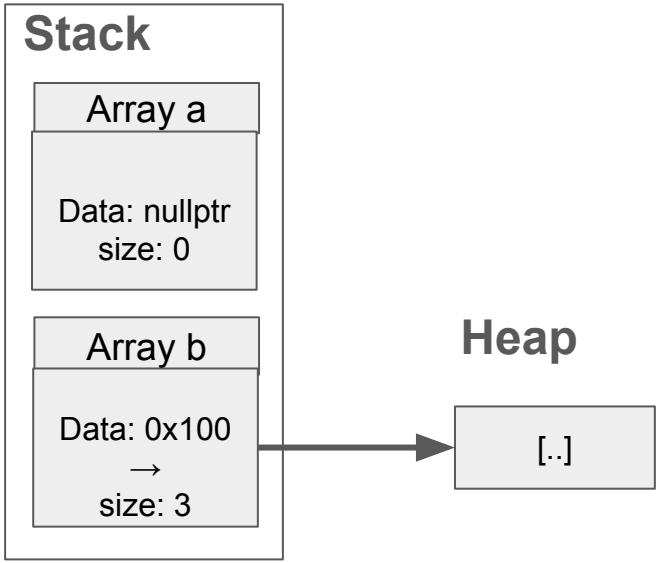
```
class Array {
int* data; int size;
public: // Move Constructor

// Move assignment
    Array& operator=(Array&& other){
        if (this != &other) {
            delete[] data;          // free current resource
            data = other.data;      // steal
            size = other.size;

            other.data = nullptr;   // leave source safe
            other.size = 0;
        }
        return *this;
```

Source

Move Constructor & Assignment




```
std::move(a)
```



The Rule of Three/Five

Rule of Three (C++98)

```
class MyClass {

public:
// 1. Destructor
~MyClass();
// 2. Copy Constructor
MyClass(const MyClass&);
// 3. Copy Assignment
MyClass& operator=(const
MyClass&);
};
```

Rule of Five (C++11)

```
class MyClass {

public:
// 1. Destructor
~MyClass();
// 2. Copy Constructor
MyClass(const MyClass&);
// 3. Copy Assignment
MyClass& operator=(const MyClass&);
// 4. Move Constructor
MyClass(MyClass&&);
// 5. Move Assignment
MyClass& operator=(MyClass&&);
};
```