Lecture: Binary Trees in C++

1. Introduction

Today, we will explore binary trees, one of the most fundamental and widely used data structures in computer science. Binary trees are essential for understanding more complex structures like Binary Search Trees (BSTs), heaps, and expression trees, and they form the basis for many algorithms in searching, sorting, and parsing.

2. What is a Binary Tree?

A binary tree is a hierarchical data structure in which each node has at most two children, referred to as the left child and the right child.

Key Properties:

- Each node contains:
  - Data (value)
  - Pointer to left child
  - Pointer to right child
- The topmost node is called the root.
- Nodes with no children are called leaves.
- The height of a binary tree is the length of the longest path from the root to a leaf.

3. Binary Tree Node Structure in C++

In C++, we typically represent a binary tree node using a struct or a class.

```
template<class T>
struct tree_node {
      T data;
      tree_node<T>* left;
      tree_node<T>* right;
      // ctor
      tree_node(const T& data, tree_node<T>* left =
nullptr,
                tree_node<T>* right = nullptr) :
                data(data), left(left), right(right) {}
};
```

4. Creating a Simple Binary Tree

Let's manually build a small binary tree:

```
// Create root
tree_node<int>* root = new tree_node<int> (1);
// Attach children
root->left = new tree_node<int> (2);
root->right = new tree_node<int> (3);
// Add grandchildren
root->left->left = new tree_node<int> (4);
root->left->right = new tree_node<int> (5);
// The tree now looks like:
//         1
//        / \
//       2   3
//      / \
//     4   5
```

5. Applications of Binary Trees

- Binary Search Trees (BSTs): Enable fast search, insertion, and deletion (average O(log n)).
- Expression Trees: Represent arithmetic expressions (e.g., (a + b) * c).
- Huffman Coding Trees: Used in data compression.
- Syntax Trees: Used by compilers to parse source code.
- Decision Trees: Used in machine learning and AI.

6. Memory Management & Best Practices

- Always free dynamically allocated memory to avoid leaks (or use smart pointers in modern C++).
- Consider using std::unique_ptr for automatic memory management:

```
#include <memory>

struct TreeNode {
    int data;
    std::unique_ptr<TreeNode> left;
    std::unique_ptr<TreeNode> right;
    TreeNode(int val) : data(val) {}
};
```

However, for learning purposes, raw pointers are often used to understand pointer manipulation.

7. Summary

- A binary tree is a recursive data structure where each node has up to two children.
- Implemented in C++ using structs/classes with pointers.
- Traversals (inorder, preorder, postorder) are foundational for tree algorithms.
- Binary trees are the basis for many advanced data structures and algorithms.

Destructors in C++ — Purpose and Necessity

1. Introduction

In C++, objects have a lifetime: they are created, used, and eventually destroyed. While constructors initialize objects, destructors are responsible for cleaning up when an object goes out of scope or is explicitly deleted.

Today, we'll explore:

- What a destructor is
- When it is called automatically
- Why it's essential in resource management
- Common scenarios requiring a custom destructor
- Pitfalls like resource leaks and double-deletion

2. What Is a Destructor?

A destructor is a special member function that:

- Has the same name as the class, preceded by a tilde (~)
- Takes no parameters and cannot be overloaded
- Cannot have a return type (not even void)
- Is automatically invoked when an object's lifetime ends

```
class MyClass {
public:
    ~MyClass() {
        // Cleanup code here
    }
};
```

◈ The compiler generates a default destructor if you don't define one.

◈ The default destructor calls destructors of member objects and base classes—but does not release resources like raw memory, file handles, or network connections.

3. When Is a Destructor Called?

SCENARIO	EXAMPLE

Local object goes out of scope	`void f()`
	`{`
	`  MyClass obj;`
	`} // ~MyClass() called at }`

Object is deleted (for dynamically allocated objects)	`MyClass* p = new MyClass();` `delete p; // ~MyClass() called`
Program terminates (for global/static) objects	Global objects are destroyed after main() ends
Temporary object is no longer needed	MyClass().doSomething(); // destroyed after statement

4. When Do You Need a Custom Destructor?

You must define a custom destructor only if your class manages resources that are not automatically released.

This follows the Rule of Three/Five:

you need to define any of destructor, copy constructor, or copy assignment operator, you likely need to define all three (or five, in C++11+ with move operations).

4. When Do You Need a Custom Destructor?

Common resources requiring manual cleanup:

- Dynamically allocated memory (new / new[])
- File handles (fopen, std::fstream)
- Threads, mutexes, sockets, database connections
- External libraries (e.g., OpenGL contexts, SDL windows)

5. Examples

✅ Example 1: Without Custom Destructor → MEMORY LEAK

```
class BadString {
    char* data;
public:
    BadString(const char* s) {
        data = new char[strlen(s) + 1];
        strcpy(data, s);
    }
    // ❌ No destructor → memory leak!
};
void test() {
    BadString s("Hello");
} // 's' destroyed, but 'data' memory is never freed!
```

Result: Every BadString object leaks memory.

5. Examples

✅ Example 2: With Proper Destructor

```
class GoodString {
    char* data;
public:
    GoodString(const char* s) {
        data = new char[strlen(s) + 1];
        strcpy(data, s);
    }

    ~GoodString() {
        delete[] data;  // ✅ Release memory
        data = nullptr; // Optional: defensive programming
    }
};
```

Now, when GoodString goes out of scope, its memory is safely freed.

5. Examples

✅ Example 3: File Handle Management

```
#include <cstdio>
class LogFile {
    FILE* file;
public:
    LogFile(const char* filename) {
        file = fopen(filename, "w");
        if (!file) throw "Cannot open file!";
    }
    ~LogFile() {
        if (file) {
            fclose(file);  // ✅ Close file on destruction
        }
    }
    void write(const char* msg) {
        fprintf(file, "%s\n", msg);
    }
};
```

Without the destructor, the file might remain open, risking data loss or resource exhaustion

6. Important Rules & Best Practices

🚫 Never call a destructor manually (unless using placement new)

```
MyClass obj;
obj.~MyClass();
// Dangerous! Destructor will be
// called again at scope end → UB
```

🔄 Order of destruction

- Members are destroyed in the reverse order of declaration.
- Base class destructors are called after derived class destructors.
- Always make base class destructors `virtual` if the class is meant to be inherited:

6. Important Rules & Best Practices

```
class Base {
public:
    virtual ~Base() = default; // ✅ Enables proper
                               // cleanup in inheritance
};
class Derived : public Base {
    int* ptr;
public:
    Derived() : ptr(new int(42)) {}
    ~Derived() { delete ptr; }
};
```

Without virtual ~Base(), deleting a Derived object via a Base* pointer causes undefined behavior (only Base destructor runs!).

7. Modern C++ Alternative: Smart Pointers

Instead of manually managing resources, prefer smart pointers:

```
#include <memory>
#include <string>

class ModernString {
    std::unique_ptr<char[]> data; // Automatically freed
    size_t len;
public:
    ModernString(const char* s) : len(strlen(s)) {
        data = std::make_unique<char[]>(len + 1);
        strcpy(data.get(), s);
    }
    // ✅ No custom destructor needed!
};
```

💡 Best practice: If you can avoid new/delete, do so. Use std::string, std::vector, std::unique_ptr, etc.

8. When Are a Copy Constructor and Assignment Operator Necessary?

- In C++, the compiler automatically generates a default copy constructor and a default copy assignment operator for every class if you don’t define them yourself. These perform member-wise copying (shallow copy).
- However, you must explicitly define both when your class manages resources (such as raw pointers, file handles, or other external entities). This is known as the Rule of Three (or Rule of Five in C++11 and later).

8. When Are a Copy Constructor and Assignment Operator Necessary?

The Rule of Three

If your class defines any one of the following, it likely needs all three:

- Destructor
- Copy constructor
- Copy assignment operator

Why?

Because the default (compiler-generated) versions perform shallow copying, which can lead to serious bugs when your class owns resources like dynamically allocated memory.

🔍 Common Scenario: Classes with Raw Pointers

❌ Problem: Shallow Copy → Double Deletion

```
class BadArray {
    int* data;
    size_t size;
public:
    BadArray(size_t n) : size(n) {
        data = new int[size];
    }
    // ❌ No custom copy constructor or assignment
    // ❌ Default destructor (or even with destructor, still unsafe)
    ~BadArray() { delete[] data; }
};
```

🔍 Common Scenario: Classes with Raw Pointers

Now consider:

```
BadArray a(5);
BadArray b = a; // Uses default copy constructor
                // → copies pointer!
```

Both a.data and b.data point to the same memory.

When both objects are destroyed, delete[] is called twice on the same address → undefined behavior (usually a crash).

✅ Solution: Define All Three

```
class GoodArray {
    int* data;
    size_t size;
public:
    // Constructor
    GoodArray(size_t n) : size(n) {
        data = new int[size]();
    }
    // Destructor
    ~GoodArray() {
        delete[] data;
    }
    // Copy constructor
    GoodArray(const GoodArray& other) : size(other.size) {
        data = new int[size];
        std::copy(other.data, other.data + size, data);
    }
```

✅ Solution: Define All Three

```
    // Copy assignment operator
    GoodArray& operator=(const GoodArray& other) {
        if (this == &other) return *this;  // Self-assignment
                                           // check
        delete[] data;                     // Free existing
                                           // resource
        size = other.size;
        data = new int[size];
        std::copy(other.data, other.data + size, data);
        return *this;
    }
};
```

Now copying creates independent copies (deep copy) — safe and correct.

🛑 When Are They Not Needed?

You do NOT need to define them if your class:

- Uses only automatic storage (e.g., int, std::string, std::vector, std::unique_ptr)
- Relies on RAII types from the standard library

Example of a class that does NOT need custom copy/assign:

```
class SafeStringHolder {
    std::string name;      // Manages its own memory
    std::vector<int> nums; // Manages its own memory
public:
    // ✅ Compiler-generated copy constructor and
    // assignment work perfectly
};
```

💡 Modern C++ Tip: Prefer std::string, std::vector, std::unique_ptr, etc., to avoid manual memory management entirely.

Bonus: Prevent Copying (C++11+)

If your class should not be copied (e.g., represents a unique resource like a thread or file):

```
class UniqueResource {
public:
    UniqueResource(const UniqueResource&) = delete;
    UniqueResource& operator=(const UniqueResource&) = delete;
};
```

The Copy-and-Swap Idiom in C++

1. What Is Copy-and-Swap?

Copy-and-swap is a powerful and elegant C++ idiom used to implement the copy assignment operator in a way that is:

- Exception-safe
- Strongly exception-safe (provides the strong guarantee)
- Avoids code duplication
- Automatically handles self-assignment

Instead of manually copying data and managing resources inside the assignment operator, copy-and-swap leverages the copy constructor and a swap function.

The Copy-and-Swap Idiom in C++

1. What Is Copy-and-Swap?

Copy-and-swap is a powerful and elegant C++ idiom used to implement the copy assignment operator in a way that is:

- Exception-safe
- Strongly exception-safe (provides the strong guarantee)
- Avoids code duplication
- Automatically handles self-assignment

Instead of manually copying data and managing resources inside the assignment operator, copy-and-swap leverages the copy constructor and a swap function.

2. How It Works

The idea is simple:

To assign object a = b:

1. Make a copy of b (using the copy constructor)
2. Swap the contents of a with that copy
3. The temporary copy is destroyed automatically, taking the old resources of a with it

This approach delegates resource management to the copy constructor (which you already need to write correctly) and a non-throwing swap function.

3. Implementation

Step 1: Define a swap member function (or a friend swap)

```
class MyClass {
private:
    int* data;
    size_t size;

public:
    // Constructor, destructor, copy
constructor... (Rule of Three/Five)
```

3. Implementation

```
    // Swap member function (should never throw)
    void swap(MyClass& other) noexcept {
        std::swap(data, other.data);
        std::swap(size, other.size);
    }

    // Copy assignment using copy-and-swap
    MyClass& operator=(MyClass other) {  // Note: passed by VALUE!
        swap(other);  // Swap this with the copy
        return *this;
    }
};
```

🔍 Key Detail:

- The parameter other is passed by value → this triggers the copy constructor.
- If the user writes a = b, b is copied into other.
- If the user writes a = std::move(b), and a move constructor exists, other is move-constructed (even better!).

Thus, one assignment operator supports both copy and move assignment (if move constructor is defined)!

4. Why Is It Exception-Safe?

- All risky operations (like memory allocation) happen during the copy construction of the parameter.
- If an exception is thrown during copying, the original object (*this) remains unchanged.
- The swap function itself must not throw (hence noexcept) — swapping pointers/integers is safe.

This gives the strong exception guarantee:

Either the assignment succeeds completely, or the object is left exactly as it was.

5. Self-Assignment? No Problem!

With copy-and-swap, self-assignment (a = a) works correctly without any extra checks:

- other becomes a copy of a
- swap(a, other) just exchanges identical values
- No double-delete, no leaks, no bugs

Compare this to manual assignment, where you often need:

```
if (this == &other) return *this;
// Not needed with copy-and-swap!
```

6. Final Thought

“Do the work in the right place.”

Let the copy constructor handle copying, the destructor handle cleanup, and the assignment operator just orchestrate the swap.

The copy-and-swap idiom embodies this principle beautifully.

Move Semantics in C++ — Move Constructor and Move Assignment Operator

1. Introduction

In C++11, a powerful feature called move semantics was introduced to improve performance by avoiding unnecessary deep copies of objects. This is especially important for objects that manage expensive resources like dynamic memory, files, or network connections.

At the heart of move semantics are:

- Rvalue references (T&&)
- Move constructor
- Move assignment operator

Together, they enable efficient transfer of resources from temporary (or no-longer-needed) objects.

2. Lvalues vs. Rvalues

Before understanding move semantics, we must distinguish between lvalues and rvalues:

Type	Description	Example
Lvalue	An object that has a name and persists beyond a single expression. Has an address.	int x = 5; → x is an lvalue
Rvalue	A temporary, unnamed value that is about to be destroyed. Cannot be assigned to.	5, func(), MyClass()

💡 Key idea: We can safely "steal" resources from rvalues because they won’t be used again.

3. Rvalue References (T&&)

C++11 introduced rvalue references, denoted by T&&, which bind only to rvalues.

```
int&& r = 42;                  // OK: 42 is an rvalue
int x = 10;
int&& r2 = x;                  // ❌ Error: x is an
                               // lvalue
int&& r3 = std::move(x); // ✅ OK: std::move
                                 // casts x to rvalue
```

4. Move Constructor

Purpose:

Transfer ownership of resources from a temporary object (rvalue) to a new object — without copying.

Syntax:

```
ClassName(ClassName&& other) noexcept;
```

4. Move Constructor

```
class MyString {
    char* data;
    size_t len;
public:
    // Regular constructor
    MyString(const char* s) {
        len = strlen(s);
        data = new char[len + 1];
        strcpy(data, s);
    }
    // Move constructor
    MyString(MyString&& other) noexcept
        : data(other.data), len(other.len) {
        // "Steal" the resource
        other.data = nullptr;  // Leave source in valid but empty state
        other.len = 0;
    }
    ~MyString() { delete[] data; }
};
```

4. Move Constructor

When is it called?

- When initializing an object from an rvalue:

```
MyString s1 = MyString("Hello"); // Move
                                 // constructor
                                 // (if available)
MyString s2 = std::move(s1); // Explicit move
std::vector<MyString> v;
v.push_back(MyString("World")); // Move, not copy!
```

5. Move Assignment Operator

Purpose:

Efficiently assign one object to another by moving resources instead of copying.

Syntax:

```
ClassName& operator=(ClassName&& other) noexcept;
```

Example:

```
MyString& operator=(MyString&& other) noexcept {
    if (this != &other) {
        delete[] data;           // Free current resource
        data = other.data;       // Steal resource
        len = other.len;
        other.data = nullptr;    // Reset source
        other.len = 0;
    }
    return *this;
}
```

If you already have a move constructor, copy-and-swap *can* work for move assignment—but providing an explicit move assignment operator is usually recommended for performance, clarity, and adherence to C++ best practices.

5. Move Assignment Operator

Consider this without move semantics:

```
std::vector<MyString> v;
v.push_back(MyString("Temp")); // Without move:
                               // deep copy +
                               // delete temp
```

With move semantics:

- The temporary MyString("Temp") is moved into the vector.
- No memory allocation or copying — just pointer transfer!
- Much faster, especially for large objects.

📊 Real-world impact: Moving a std::vector of 1 million elements is O(1) instead of O(n)!

7. The Rule of Five

If you define any of the following, you probably need all five:

- Destructor
- Copy constructor
- Copy assignment operator
- Move constructor
- Move assignment operator

 If you manage resources (raw pointers, etc.), define all five or disable copying/moving as needed.

Modern alternative: Use smart pointers and standard containers to avoid manual implementation.

8. When Are Move Operations Automatically Suppressed?

The compiler will not generate move operations if you define:

- A custom destructor
- A custom copy constructor or copy assignment

So if you want move semantics, declare them explicitly (or use = default if safe).

If you already have a move constructor, copy-and-swap can work for move assignment—but providing an explicit move assignment operator is usually recommended for performance, clarity, and adherence to C++ best practices.