## ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

# DYNAMIC MEMORY ALLOCATION (NEW, DELETE)

## CHAPTER 1: INTRODUCTION TO DYNAMIC MEMORY ALLOCATION

In programming, memory management is crucial for efficient execution and resource utilization. **Dynamic Memory Allocation (DMA)** is a mechanism that allows programs to allocate memory **at runtime**, rather than at compile time. This is particularly useful when dealing with data structures like linked lists, trees, and graphs, where the exact memory requirements are not known beforehand.

Unlike **static memory allocation**, where memory is assigned at compile time and remains fixed throughout execution, **dynamic memory allocation** enables flexible memory management. It allows a program to request memory when needed and release it when it is no longer required, preventing unnecessary memory wastage. In C++, **dynamic memory allocation** is achieved using the new and delete operators. The new operator is used to allocate memory dynamically, while the delete operator is used to deallocate memory once it is no longer needed. Proper memory management ensures that programs run efficiently and do not suffer from issues such as **memory leaks** or **dangling pointers**.

The primary benefits of **dynamic memory allocation** include:

- **Efficient Memory Utilization:** Memory is allocated and deallocated as needed, reducing wastage.

- **Scalability:** The program can handle varying amounts of data without needing to predefine fixed-size structures.
- **Data Structure Flexibility:** Dynamic memory enables the use of complex data structures like trees, graphs, and linked lists.

While dynamic memory allocation offers these advantages, it also comes with risks. Improper memory deallocation can lead to **memory leaks**, where unused memory remains allocated, consuming system resources. Additionally, **accessing freed memory** can cause undefined behavior, leading to program crashes. Thus, careful use of new and delete is essential for writing robust and efficient programs.

---

## CHAPTER 2: UNDERSTANDING THE NEW OPERATOR

The new operator in C++ is used to dynamically allocate memory for variables, arrays, and objects. It requests memory from the **heap** at runtime and returns a pointer to the allocated memory. The syntax for using new is:

### Syntax of new Operator

data_type* pointer_name = new data_type;

For example:

int* ptr = new int;  // Allocating memory for a single integer

In this case, ptr is a pointer that stores the address of the dynamically allocated memory. We can assign a value to this memory as follows:

*ptr = 10;  // Assigning value 10 to dynamically allocated memory

### Allocating Memory for Arrays

We can also allocate memory dynamically for arrays using new[]:

```
int* arr = new int[5];  // Allocating memory for an array of 5 integers
```

This enables the creation of arrays whose size is determined at runtime rather than compile-time.

**Example: Using new for Dynamic Memory Allocation**

```
#include <iostream>

using namespace std;


int main() {

   int* ptr = new int(25);  // Allocate and initialize an integer

   cout << "Value stored in dynamically allocated memory: " << *ptr
<< endl;

   delete ptr;  // Free memory

   return 0;

}
```

**Output:**

Value stored in dynamically allocated memory: 25

This example dynamically allocates memory for an integer, initializes it with 25, and then releases the memory using delete.

---

## CHAPTER 3: UNDERSTANDING THE DELETE OPERATOR

The delete operator in C++ is used to **deallocate memory** that was dynamically allocated using new. It prevents memory leaks by freeing up space that is no longer needed. If dynamically allocated memory is not explicitly freed, it remains occupied until the program terminates, potentially causing performance issues.

**Syntax of delete Operator**

delete pointer_name;

For dynamically allocated arrays, the syntax is:

delete[] pointer_name;

This ensures that all elements in the array are correctly deallocated.

**Example: Using delete to Free Memory**

```
#include <iostream>

using namespace std;


int main() {

   int* ptr = new int(100);  // Allocating memory

   cout << "Value: " << *ptr << endl;

   delete ptr;  // Deallocating memory

   return 0;

}
```

**Output:**

Value: 100

After the delete ptr statement, the allocated memory is freed, preventing memory leaks.

**Deleting Dynamically Allocated Arrays**

```
#include <iostream>

using namespace std;


int main() {

  int* arr = new int[5];  // Allocate an array of size 5


  for (int i = 0; i < 5; i++) {

    arr[i] = i * 10;  // Assign values

  }


  for (int i = 0; i < 5; i++) {

    cout << arr[i] << " ";

  }


  delete[] arr;  // Free memory allocated for array


  return 0;

}
```

**Output:**

0 10 20 30 40

Here, delete[] arr correctly deallocates the entire array, preventing memory leaks.

---

## Chapter 4: Case Study – Dynamic Memory Allocation in a Student Database System

### Problem Statement

A university needs to store and manage student records efficiently. Since the number of students can vary each semester, a **dynamic memory allocation** approach is required to handle varying data efficiently.

### Solution

Instead of using a fixed-size array, we can use new to allocate memory dynamically based on the actual number of students enrolling.

### Implementation

```
#include <iostream>

using namespace std;


class Student {

public:

    string name;
```

```
    int age;

    Student(string n, int a) {
        name = n;
        age = a;
    }

    void display() {
        cout << "Student: " << name << ", Age: " << age << endl;
    }
};

int main() {
    int num;
    cout << "Enter number of students: ";
    cin >> num;

    Student** students = new Student*[num];  // Array of Student
pointers

    for (int i = 0; i < num; i++) {
```

```
        string name;
        int age;
        cout << "Enter name and age for student " << i + 1 << ": ";
        cin >> name >> age;
        students[i] = new Student(name, age);
    }

    cout << "\nStudent Records:\n";
    for (int i = 0; i < num; i++) {
        students[i]->display();
        delete students[i];  // Free memory for each student object
    }

    delete[] students;  // Free the array of pointers

    return 0;
}
```

**Expected Output**

Enter number of students: 2

Enter name and age for student 1: Alice 20

Enter name and age for student 2: Bob 22

Student Records:

Student: Alice, Age: 20

Student: Bob, Age: 22

Here, dynamic memory allocation allows the program to store student records efficiently, adapting to varying data sizes dynamically.

---

## Chapter 5: Exercises

1. **Write a program** that dynamically allocates memory for a matrix (2D array) and performs matrix addition.
2. **Modify the student database program** to allow updating and deleting records dynamically.
3. **Create a linked list implementation** using dynamic memory allocation (new and delete).
4. **Detect memory leaks** using a memory profiling tool (e.g., Valgrind for Linux).

# SMART POINTERS (UNIQUE_PTR, SHARED_PTR, WEAK_PTR)

## CHAPTER 1: INTRODUCTION TO SMART POINTERS

In modern C++, **memory management** is a crucial aspect of efficient and safe programming. Traditional pointers require explicit **allocation (new)** and **deallocation (delete)**, which can lead to **memory leaks, dangling pointers, and undefined behavior** if not managed correctly. To address these issues, **smart pointers** were introduced in **C++11**, providing an automatic and efficient way to manage dynamically allocated memory.

**Smart pointers** are part of the **Standard Library (<memory>)** and are designed to **automatically manage memory resources** by utilizing **RAII (Resource Acquisition Is Initialization)**. Unlike raw pointers, smart pointers ensure that memory is released when it is no longer needed, thus preventing **memory leaks and ownership issues**.

### Advantages of Smart Pointers

- **Automatic Memory Management:** No need to explicitly use delete.
- **Exception Safety:** Smart pointers automatically free memory, even if an exception occurs.
- **Ownership Management:** Different types of smart pointers manage ownership differently (unique_ptr, shared_ptr, weak_ptr).
- **Improved Code Readability:** Easier to understand and debug.

There are three primary types of smart pointers:

1. **unique_ptr** – Exclusive ownership of a dynamically allocated resource.
2. **shared_ptr** – Shared ownership of a resource, with reference counting.
3. **weak_ptr** – A weak reference to a shared resource to prevent circular dependencies.

In the following chapters, we will explore each of these in detail with **examples, case studies, and exercises** to ensure a thorough understanding.

---

## CHAPTER 2: UNIQUE_PTR – EXCLUSIVE OWNERSHIP SMART POINTER

A unique_ptr is a smart pointer that **owns and manages** a dynamically allocated object **exclusively**. No other smart pointer can own the same object. When the unique_ptr goes out of scope, the object is automatically deleted.

### Key Features of unique_ptr

- **Single Ownership:** Only one unique_ptr can own a resource at a time.
- **Prevents Copying:** A unique_ptr **cannot be copied** to another unique_ptr.
- **Supports Move Semantics:** Ownership can be transferred using std::move().

### Example: Using unique_ptr

```
#include <iostream>
```

```
#include <memory>  // Include smart pointers

using namespace std;

class Example {
public:
    Example() { cout << "Resource acquired\n"; }
    ~Example() { cout << "Resource released\n"; }
};

int main() {
    unique_ptr<Example> ptr1 = make_unique<Example>(); // Creating a unique_ptr
    // unique_ptr<Example> ptr2 = ptr1; // ERROR: Copy not allowed
    unique_ptr<Example> ptr2 = move(ptr1); // Transferring ownership

    cout << "ptr1 is " << (ptr1 ? "not null" : "null") << endl;
    cout << "ptr2 is " << (ptr2 ? "not null" : "null") << endl;

    return 0;
}
```

### Expected Output

Resource acquired

ptr1 is null

ptr2 is not null

Resource released

Here, ptr1 transfers ownership to ptr2 using std::move(), ensuring that only one pointer owns the resource at a time.

### Use Cases of unique_ptr

- Managing **file handles** and **database connections**.
- Ensuring **proper cleanup of dynamically allocated memory**.
- **RAII pattern** to automatically handle resource allocation and deallocation.

---

## CHAPTER 3: SHARED_PTR – REFERENCE COUNTED SMART POINTER

A shared_ptr allows **multiple smart pointers** to share ownership of a dynamically allocated resource. The object is **deleted only when the last shared_ptr owning it is destroyed**.

### Key Features of shared_ptr

- **Reference Counting:** Keeps track of the number of shared_ptr instances sharing ownership.
- **Automatic Cleanup:** When the last shared_ptr is destroyed, the resource is deallocated.

- **Supports Copying:** Unlike unique_ptr, shared_ptr can be copied.

**Example: Using shared_ptr**

```
#include <iostream>

#include <memory>


using namespace std;


class Example {

public:

   Example() { cout << "Resource acquired\n"; }

   ~Example() { cout << "Resource released\n"; }

};


int main() {

   shared_ptr<Example> ptr1 = make_shared<Example>();  // Shared ownership

   shared_ptr<Example> ptr2 = ptr1;  // ptr2 also owns the resource


   cout << "Reference Count: " << ptr1.use_count() << endl;
```

```
    return 0;  // Resource is freed only when the last shared_ptr is destroyed
}
```

**Expected Output**

```
Resource acquired
Reference Count: 2
Resource released
```

Here, ptr1 and ptr2 share ownership. The reference count is 2, meaning two smart pointers own the resource. The resource is only released when both pointers go out of scope.

**Use Cases of shared_ptr**

- **Graph and Tree structures**, where multiple nodes may share ownership of a resource.
- **Multithreading**, where multiple threads might need access to shared data.
- **Factory functions**, returning a shared resource to multiple consumers.

---

## CHAPTER 4: WEAK_PTR – PREVENTING CIRCULAR REFERENCES

A weak_ptr is a smart pointer that **holds a weak reference** to a shared_ptr but **does not increase its reference count**. This is useful for breaking **circular dependencies** in shared ownership scenarios.

**Problem: Circular Reference in shared_ptr**

If two objects contain shared_ptr references to each other, they create a **circular dependency**, preventing the memory from being deallocated.

**Solution: Using weak_ptr**

A weak_ptr does not contribute to the reference count, allowing safe handling of cyclic references.

**Example: Using weak_ptr to Prevent Cyclic References**

```
#include <iostream>

#include <memory>


using namespace std;


class B;  // Forward declaration


class A {

public:

   shared_ptr<B> bptr;  // Shared ownership of B

   ~A() { cout << "A destroyed\n"; }

};


class B {

public:
```

```
    weak_ptr<A> aptr;  // Weak reference to A
    ~B() { cout << "B destroyed\n"; }
};

int main() {
    shared_ptr<A> a = make_shared<A>();
    shared_ptr<B> b = make_shared<B>();

    a->bptr = b;
    b->aptr = a;  // Weak reference prevents circular dependency

    return 0;
}
```

**Expected Output**

A destroyed

B destroyed

If aptr were a shared_ptr, both objects would persist indefinitely due to a circular reference. Using weak_ptr ensures proper memory deallocation.

**Use Cases of weak_ptr**

- **Observer Design Pattern**, where observers hold weak references to prevent ownership issues.

- **Graph Structures**, where nodes might have non-owning references to other nodes.
- **Cache Systems**, where temporary data should not prevent resource deallocation.

---

## Chapter 5: Exercises

1. **Convert a raw pointer program to use unique_ptr and shared_ptr.**
2. **Implement a factory function using shared_ptr to create objects.**
3. **Demonstrate a cyclic reference problem with shared_ptr, then fix it using weak_ptr.**
4. **Write a program that dynamically allocates an array using unique_ptr.**

---

## Conclusion

Smart pointers (unique_ptr, shared_ptr, weak_ptr) **simplify memory management** in C++, prevent **memory leaks**, and **improve program safety**. Understanding their differences and use cases ensures effective resource management in **real-world applications**.

# HANDLING MEMORY LEAKS AND OPTIMIZATION

## CHAPTER 1: INTRODUCTION TO MEMORY LEAKS AND OPTIMIZATION

Efficient memory management is a crucial aspect of software development. Poor memory handling can lead to **memory leaks**, which occur when a program allocates memory dynamically but fails to release it after use. Over time, these leaks accumulate, reducing available system memory, slowing down applications, and potentially causing crashes.

Memory leaks occur primarily in languages like **C and C++**, where developers manually allocate and deallocate memory using new and delete (or malloc and free). While modern languages like Java and Python use **garbage collection**, improper object references can still cause memory inefficiencies.

Optimizing memory usage is essential for improving **performance, efficiency, and stability** in software applications. Techniques such as **smart pointers, garbage collection, reference counting, and memory profiling tools** help mitigate memory leaks and ensure efficient resource usage.

In this chapter, we will explore memory leaks, their causes, detection methods, and optimization techniques to improve memory efficiency in applications.

## CHAPTER 2: UNDERSTANDING MEMORY LEAKS

### What is a Memory Leak?

A **memory leak** occurs when a program allocates memory but fails to deallocate it, causing a permanent increase in memory usage. This leads to **gradual depletion of available memory**, affecting system performance and eventually causing crashes.

### Causes of Memory Leaks

1. **Failure to Release Dynamically Allocated Memory**
   - Example: Forgetting to use delete after new in C++.
2. int* ptr = new int(10);
3. // Memory leak occurs as delete is not called
4. **Circular References in Smart Pointers**
   - shared_ptr with circular dependencies can prevent memory deallocation.
5. **Lost Pointers (Dangling References)**
   - Assigning a new value to a pointer before freeing the previous allocation.
6. int* ptr = new int(5);
7. ptr = new int(10); // Memory leak: Previous allocation is lost
8. **Improper Use of Static Variables**
   - Static variables persist for the entire program lifecycle and may retain memory unnecessarily.

---

## CHAPTER 3: DETECTING MEMORY LEAKS

Memory leaks can be difficult to detect manually, so developers use specialized tools and techniques to identify leaks.

**Methods for Detecting Memory Leaks**

**1. Using Valgrind (Linux)**

Valgrind is a powerful memory analysis tool that detects memory leaks, invalid memory access, and memory usage inefficiencies.

```
valgrind --leak-check=full ./a.out
```

**2. Using AddressSanitizer (GCC/Clang)**

GCC and Clang compilers support AddressSanitizer for memory error detection.

```
g++ -fsanitize=address -g program.cpp -o program

./program
```

**3. Using CRT Debug Heap (Windows)**

For Windows, _CrtDumpMemoryLeaks() in **Microsoft Visual Studio** helps identify leaks.

```
#define _CRTDBG_MAP_ALLOC

#include <cstdlib>

#include <crtdbg.h>


int main() {

   int* ptr = new int(10);

   _CrtDumpMemoryLeaks(); // Detects leaks at program exit

   return 0;
```

}

### 4. Using Smart Pointers for Automatic Memory Management

Replacing raw pointers with **unique_ptr** or **shared_ptr** prevents leaks by ensuring proper deallocation.

---

## CHAPTER 4: PREVENTING AND FIXING MEMORY LEAKS

### 1. Using delete and delete[] Properly

Every dynamically allocated memory (new) must be explicitly freed using delete.

```
int* ptr = new int(10);

delete ptr; // Free allocated memory
```

For arrays, use delete[]:

```
int* arr = new int[5];

delete[] arr; // Proper deallocation
```

### 2. Using Smart Pointers (unique_ptr, shared_ptr)

Smart pointers in C++ **automatically** manage memory and prevent leaks.

**Using unique_ptr (Single ownership)**

```
#include <memory>

int main() {

  std::unique_ptr<int> ptr = std::make_unique<int>(10); // Auto-deallocated
```

```
}
```

**Using shared_ptr (Reference counting)**

```
#include <memory>

std::shared_ptr<int> ptr1 = std::make_shared<int>(10);

std::shared_ptr<int> ptr2 = ptr1; // Shared ownership
```

**Using weak_ptr (Prevents circular dependencies)**

```
#include <memory>

class B;


class A {

public:

   std::shared_ptr<B> bptr;

};


class B {

public:

   std::weak_ptr<A> aptr; // Prevents circular reference

};
```

## 3. Avoiding Circular References in shared_ptr

When using shared_ptr, ensure circular references are avoided with weak_ptr.

### 4. Using RAII (Resource Acquisition Is Initialization)

Encapsulate resources in classes with proper destructors to automatically handle deallocation.

```
class Resource {
public:
    Resource() { ptr = new int(10); }
    ~Resource() { delete ptr; } // Automatically cleans up memory
private:
    int* ptr;
};
```

### 5. Using std::vector Instead of Raw Dynamic Arrays

Instead of manually allocating arrays, use std::vector for automatic memory management.

```
std::vector<int> vec = {1, 2, 3, 4, 5}; // No need for manual memory allocation
```

---

## CHAPTER 5: OPTIMIZING MEMORY USAGE

Besides preventing memory leaks, optimizing memory usage improves program performance.

### 1. Using Stack Allocation Instead of Heap Allocation

Stack allocation is faster and automatically cleaned up when out of scope.

```
void function() {
```

```
    int arr[10]; // Allocated on stack, automatically deallocated
}
```

**2. Using reserve() with std::vector**

By reserving memory in advance, std::vector avoids frequent reallocation.

```
std::vector<int> vec;
vec.reserve(100); // Allocates memory for 100 elements upfront
```

**3. Using Memory Pools for Frequent Allocations**

For frequently allocated objects, using a **memory pool** improves efficiency.

**4. Using std::move to Transfer Ownership Efficiently**

Instead of copying objects, use std::move to transfer ownership.

```
std::unique_ptr<int> ptr1 = std::make_unique<int>(10);
std::unique_ptr<int> ptr2 = std::move(ptr1); // Transfers ownership
```

---

## CHAPTER 6: CASE STUDY – OPTIMIZING A GAME ENGINE'S MEMORY USAGE

**Problem Statement**

A game engine dynamically allocates **thousands of objects** (players, enemies, and bullets) during gameplay, causing **high memory usage and potential leaks**.

**Solution**

1. **Use Smart Pointers:**

- unique_ptr for single-ownership resources like textures.
- shared_ptr for resources shared among multiple objects.

2. **Implement Object Pooling:**
    - Reuse frequently allocated objects instead of creating new ones each time.
3. **Minimize Heap Allocations:**
    - Use stack-allocated arrays and pre-allocated memory buffers.
4. **Profile and Detect Leaks:**
    - Use Valgrind or AddressSanitizer to detect leaks and optimize memory.

**Implementation**

```
#include <vector>

#include <memory>


class GameObject {

public:

  GameObject() { /* Load model */ }

  ~GameObject() { /* Cleanup */ }

};


int main() {
```

```
    std::vector<std::unique_ptr<GameObject>> gameObjects;

    for (int i = 0; i < 1000; ++i) {
        gameObjects.push_back(std::make_unique<GameObject>());
    } // Objects automatically deleted when vector goes out of scope
}
```

**Outcome**

- Reduced **memory leaks** using smart pointers.
- Improved **performance** by avoiding frequent allocations.
- Enhanced **scalability** by optimizing object reuse.

---

## Chapter 7: Exercises

1. Implement a program that demonstrates **memory leaks** and fixes them using **smart pointers**.
2. Use Valgrind to analyze a program for memory leaks.
3. Optimize a program using **memory pooling**.
4. Implement **circular reference handling** with weak_ptr.

---

## Conclusion

Efficient memory management through **smart pointers, profiling tools, and optimization techniques** is critical for developing **high-performance applications**. By implementing these best practices, developers can prevent **memory leaks, optimize resource usage, and ensure robust software performance**.

# MOVE SEMANTICS AND R-VALUE REFERENCES

## CHAPTER 1: INTRODUCTION TO MOVE SEMANTICS AND R-VALUE REFERENCES

In modern C++, **efficient memory management and performance optimization** are critical aspects of software development. **Move semantics** and **R-value references** were introduced in **C++11** to improve performance by **eliminating unnecessary copies** when dealing with **temporary objects**.

Traditionally, when objects were passed or returned by value, **deep copies** were made, leading to unnecessary memory allocation and deallocation. This process was inefficient, especially for **large objects** such as **vectors, strings, and files**. Move semantics optimize performance by **transferring ownership** of resources rather than copying them.

Move semantics leverage **R-value references (&&)**, which allow the compiler to distinguish **temporary objects** (R-values) from **persistent objects** (L-values). This optimization is particularly useful in **containers, string manipulation, and dynamic memory management**.

### Key Benefits of Move Semantics

- **Avoids Unnecessary Copies:** Instead of duplicating data, move semantics transfers ownership.
- **Optimized Performance:** Reduces memory allocations and deallocations.

- **Used in STL Containers:** Standard containers like std::vector, std::string, and std::unique_ptr use move semantics for efficiency.

In this chapter, we will explore **R-values, L-values, move constructors, move assignment operators, and best practices** for implementing move semantics in C++.

---

## CHAPTER 2: UNDERSTANDING R-VALUES AND L-VALUES

### 1. What are L-values?

An **L-value** refers to an object that **persists beyond a single expression** and can be assigned a new value. L-values **have a memory address** and exist throughout the program's execution.

### Examples of L-values

```
int x = 10;  // 'x' is an L-value

x = 20;      // L-values can be modified


std::string str = "Hello";  // 'str' is an L-value
```

### 2. What are R-values?

An **R-value** is a **temporary object** that does **not have a persistent memory address**. R-values are typically **returned by expressions or function calls** and **cannot be assigned new values**.

### Examples of R-values

```
int y = 5 + 3;   // '5 + 3' produces an R-value

std::string name = "John";  // "John" is an R-value (temporary string)
```

int&& z = 10;     // R-value reference

### 3. Difference Between L-value and R-value References

- **L-value Reference (T&)**: Binds to persistent objects.
- **R-value Reference (T&&)**: Binds only to temporary (R-value) objects.

**Example: L-value vs. R-value References**

int a = 10;   // 'a' is an L-value

int& refA = a;  // Valid: L-value reference

int&& rref = 20; // Valid: R-value reference (temporary object)

R-value references (&&) enable **move semantics** by distinguishing temporary objects from permanent ones.

---

## Chapter 3: Move Constructors and Move Assignment Operator

### 1. The Need for Move Semantics

Consider a class managing a **large dynamic array**. Copying this class **creates unnecessary duplication** of memory, leading to performance inefficiencies.

### 2. Move Constructor

A **move constructor** transfers ownership of a resource from one object to another **without creating a copy**.

**Syntax**

ClassName(ClassName&& other);

**Example: Move Constructor**

```
#include <iostream>
#include <cstring>

class String {
private:
   char* data;
public:
   // Constructor
   String(const char* str) {
      data = new char[strlen(str) + 1];
      strcpy(data, str);
      std::cout << "String Created\n";
   }

   // Move Constructor
   String(String&& other) noexcept {
      data = other.data;  // Transfer ownership
```

```
        other.data = nullptr;  // Prevent the old object from deleting the
data
        std::cout << "Move Constructor Called\n";
    }

    // Destructor
    ~String() { delete[] data; }

    void show() { std::cout << (data ? data : "Empty") << std::endl; }
};

int main() {
    String s1("Hello");
    String s2 = std::move(s1);  // Move constructor is called

    s1.show();  // Output: Empty (ownership transferred)
    s2.show();  // Output: Hello

    return 0;
}
```

**Expected Output**

String Created

Move Constructor Called

Empty

Hello

### 3. Move Assignment Operator

A **move assignment operator** is used when an existing object is assigned an R-value (temporary object). It releases existing resources before transferring ownership.

**Syntax**

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

**Example: Move Assignment Operator**

```
class String {
private:
    char* data;
public:
    // Constructor
    String(const char* str) {
        data = new char[strlen(str) + 1];
        strcpy(data, str);
    }

    // Move Assignment Operator
```

```
  String& operator=(String&& other) noexcept {
    if (this != &other) {  // Avoid self-assignment
      delete[] data;  // Free existing resource
      data = other.data;  // Transfer ownership
      other.data = nullptr;
    }
    return *this;
  }

  ~String() { delete[] data; }
};
```

---

## CHAPTER 4: OPTIMIZING PERFORMANCE WITH MOVE SEMANTICS

### 1. Using std::move()

The std::move() function **converts an L-value into an R-value**, enabling move semantics.

**Example: Using std::move()**

```
#include <iostream>
#include <vector>

int main() {
```

```
    std::vector<int> v1 = {1, 2, 3};
    std::vector<int> v2 = std::move(v1); // Transfers ownership


    std::cout << "v1 size: " << v1.size() << std::endl;
    std::cout << "v2 size: " << v2.size() << std::endl;


    return 0;
}
```

**Expected Output**

```
v1 size: 0
v2 size: 3
```

Here, v1 is emptied, and ownership is transferred to v2 without making a copy.

**2. Move Semantics in STL Containers**

STL containers like std::vector, std::string, and std::unique_ptr use move semantics for efficiency.

```
std::vector<std::string> vec;
vec.push_back(std::move(std::string("Hello")));
```

This avoids an extra copy operation by moving the temporary string.

---

## Chapter 5: Case Study – Move Semantics in Large Data Processing

## Problem Statement

A **data processing system** needs to process large datasets. Copying data slows down performance, and **memory usage becomes inefficient**.

## Solution

By using **move semantics**, the system can transfer ownership instead of copying large data structures.

## Implementation

```
#include <iostream>

#include <vector>


class Data {

public:

   std::vector<int> values;


   // Move Constructor

   Data(Data&& other) noexcept : values(std::move(other.values)) {}


   // Move Assignment Operator

   Data& operator=(Data&& other) noexcept {

      values = std::move(other.values);

      return *this;
```

```
    }
};

int main() {
    Data d1;
    d1.values = {1, 2, 3, 4, 5};

    Data d2 = std::move(d1); // Moves instead of copying

    std::cout << "d1 size: " << d1.values.size() << std::endl;
    std::cout << "d2 size: " << d2.values.size() << std::endl;

    return 0;
}
```

**Expected Output**

d1 size: 0

d2 size: 5

**Move semantics** reduce redundant copying, improving efficiency.

---

## Chapter 6: Exercises

1. **Implement a move constructor and move assignment operator for a Matrix class.**
2. **Modify a program to replace deep copies with move semantics and benchmark performance.**
3. **Analyze STL containers to see how move semantics improve efficiency.**

---

## CONCLUSION

Move semantics and R-value references **optimize memory management**, **reduce unnecessary copies**, and **improve performance**, making them essential for modern C++ development.

# Assignment Solution: Develop a Memory-Efficient C++ Application Using Smart Pointers

This step-by-step guide provides a **complete solution** for developing a **memory-efficient C++ application** using **smart pointers (unique_ptr, shared_ptr, and weak_ptr)**. Smart pointers ensure **automatic memory management**, preventing **memory leaks** and **dangling pointers** while improving overall **application efficiency**.

---

## Step 1: Understanding Smart Pointers

**Smart pointers** are part of the **C++ Standard Library (<memory>)** and manage dynamic memory automatically, ensuring proper deallocation when objects go out of scope.

**Types of Smart Pointers:**

1. **unique_ptr** - Exclusive ownership (only one pointer owns the resource).
2. **shared_ptr** - Shared ownership (multiple pointers can own the resource).
3. **weak_ptr** - Weak reference to a shared_ptr to avoid circular dependencies.

Using these smart pointers, we will develop a **memory-efficient Employee Management System**.

---

## Step 2: Define the Problem – Employee Management System

A company needs an application to **manage employee records**, ensuring **efficient memory usage** and **safe object handling**.

**Requirements:**

- Store **employee details (ID, name, department)**.
- Prevent **memory leaks** using **smart pointers**.
- Implement **unique ownership for each employee**.
- Allow **multiple references** to shared employees.
- Avoid **circular references** using weak_ptr.

---

## STEP 3: DESIGN THE APPLICATION STRUCTURE

The application consists of:

1. **Employee Class** (Stores employee details).
2. **Department Class** (Holds references to employees using shared_ptr).
3. **Company Class** (Manages all departments).

We will use:

- **unique_ptr** for Company ownership.
- **shared_ptr** for employees shared across departments.
- **weak_ptr** to prevent circular dependencies.

---

## STEP 4: IMPLEMENT EMPLOYEE CLASS

Each employee will have a **unique ID, name, and department**.

```
#include <iostream>

#include <memory>

#include <vector>

#include <string>


using namespace std;


class Employee {

private:

   int id;

   string name;

   string department;


public:

   // Constructor

   Employee(int empId, string empName, string dept)

      : id(empId), name(empName), department(dept) {

      cout << "Employee Created: " << name << endl;

   }


   // Destructor
```

```
  ~Employee() {
    cout << "Employee Deleted: " << name << endl;
  }

  // Display Employee Info
  void show() const {
    cout << "ID: " << id << ", Name: " << name
       << ", Department: " << department << endl;
  }
};
```

✅ **shared_ptr will be used to store employee objects**, allowing multiple departments to share employees.

---

## STEP 5: IMPLEMENT DEPARTMENT CLASS

Each department will **store employees** using shared_ptr.

```
class Department {
private:
  string deptName;
  vector<shared_ptr<Employee>> employees;  // Employees shared across departments

public:
```

```
    // Constructor
    Department(string name) : deptName(name) {}

    // Add Employee to Department
    void addEmployee(shared_ptr<Employee> emp) {
        employees.push_back(emp);
    }

    // Display Department Employees
    void showEmployees() {
        cout << "Department: " << deptName << endl;
        for (auto emp : employees) {
            emp->show();
        }
    }
};
```

✅ **Employees can be added to multiple departments without memory duplication**.

---

## Step 6: Implement Company Class

The Company class will manage departments **using unique_ptr**, ensuring exclusive ownership.

```
class Company {
private:
   string companyName;
   vector<unique_ptr<Department>> departments;  // Unique
ownership

public:
   // Constructor
   Company(string name) : companyName(name) {}

   // Add Department
   void addDepartment(unique_ptr<Department> dept) {
      departments.push_back(move(dept));  // Move ownership
   }

   // Display Departments
   void showDepartments() {
      cout << "Company: " << companyName << endl;
      for (auto& dept : departments) {
         dept->showEmployees();
      }
```

```
    }
};
```

✅ **Departments are managed uniquely by the company using unique_ptr.**

---

### Step 7: Prevent Circular References Using weak_ptr

If a department **holds references to a company** (to access company details), it can cause a **circular reference**.

**Fix:** Use weak_ptr instead of shared_ptr.

```
class Department {
private:
    string deptName;
    weak_ptr<Company> companyRef;  // Prevent circular dependency

public:
    Department(string name, shared_ptr<Company> comp) : deptName(name), companyRef(comp) {}

    void showCompany() {
        if (auto comp = companyRef.lock()) {  // Check if reference is still valid
            cout << "Department " << deptName << " belongs to " << comp->getCompanyName() << endl;
```

```
        } else {
            cout << "Company reference no longer exists" << endl;
        }
    }
};
```

✅ **Prevents memory leaks due to circular dependencies.**

---

## Step 8: Implement Main Function to Test Application

```
int main() {
    // Create a Company using unique_ptr
    unique_ptr<Company> myCompany = make_unique<Company>("TechCorp");

    // Create Employees using shared_ptr
    shared_ptr<Employee> emp1 = make_shared<Employee>(101, "Alice", "HR");
    shared_ptr<Employee> emp2 = make_shared<Employee>(102, "Bob", "IT");
    shared_ptr<Employee> emp3 = make_shared<Employee>(103, "Charlie", "Finance");

    // Create Departments
```

```
    unique_ptr<Department> hrDept =
make_unique<Department>("Human Resources");

    unique_ptr<Department> itDept =
make_unique<Department>("Information Technology");


    // Add Employees to Departments

    hrDept->addEmployee(emp1);

    itDept->addEmployee(emp2);

    itDept->addEmployee(emp3);


    // Add Departments to Company

    myCompany->addDepartment(move(hrDept));

    myCompany->addDepartment(move(itDept));


    // Display Company Structure

    myCompany->showDepartments();


    return 0;

}
```

---

## STEP 9: EXPECTED OUTPUT

Employee Created: Alice

Employee Created: Bob

Employee Created: Charlie

Company: TechCorp

Department: Human Resources

ID: 101, Name: Alice, Department: HR

Department: Information Technology

ID: 102, Name: Bob, Department: IT

ID: 103, Name: Charlie, Department: Finance

Employee Deleted: Alice

Employee Deleted: Bob

Employee Deleted: Charlie

**What Happens in Memory?**

1. **unique_ptr<Company>** ensures Company is deleted properly.
2. **shared_ptr<Employee>** shares ownership between departments.
3. **weak_ptr<Company>** prevents circular dependencies.
4. **When main() exits, all objects are safely deallocated**.

---

## Step 10: Benefits of Using Smart Pointers

| **Smart Pointer** | **Used In** | **Benefits** |
|---|---|---|

| | | |
|---|---|---|
| unique_ptr | Company | Ensures exclusive ownership of departments. |
| shared_ptr | Employee | Allows employees to be shared between departments. |
| weak_ptr | Company Reference in Department | Prevents circular reference memory leaks. |

---

## STEP 11: ADDITIONAL EXERCISES

1. **Modify the program** to allow dynamic hiring (add new employees at runtime).
2. **Use std::vector instead of manual dynamic memory allocation**.
3. **Benchmark the program** to compare performance with raw pointers.

---

## CONCLUSION

This step-by-step implementation **demonstrates memory-efficient C++ application development** using **smart pointers**. By combining **unique_ptr, shared_ptr, and weak_ptr**, we ensure **efficient memory management, prevent memory leaks, and improve application performance**.

# ASSIGNMENT SOLUTION: CREATE A RESOURCE MANAGEMENT SYSTEM UTILIZING MOVE SEMANTICS

This step-by-step guide provides a **complete solution** for developing a **Resource Management System** using **move semantics** in **C++**. Move semantics improve performance by **eliminating unnecessary copies** and **transferring ownership** of resources instead of duplicating them.

---

## STEP 1: UNDERSTANDING MOVE SEMANTICS

### 1. What is Move Semantics?

Move semantics in **C++11** optimize resource handling by **transferring ownership** of dynamically allocated memory instead of copying it. This reduces **memory allocation overhead** and prevents **unnecessary deep copies** of large objects.

### 2. Why Use Move Semantics?

- **Improves Performance** – Avoids expensive deep copies of large objects.
- **Efficient Memory Usage** – Transfers ownership instead of duplicating resources.
- **Prevents Unnecessary Allocations/Deallocations** – Avoids repeated memory allocation.

### 3. Key Components of Move Semantics

- **Move Constructor:** Transfers ownership from a temporary object.

- **Move Assignment Operator:** Ensures that an object safely transfers resources.
- **std::move() Function:** Converts an L-value into an R-value, enabling move operations.

In this assignment, we will **develop a resource management system** that efficiently manages memory using **move semantics**.

---

## Step 2: Define the Problem – Resource Management System

**Requirements:**

- Implement a **Resource class** that manages a dynamically allocated resource.
- Implement **move constructor and move assignment operator** to efficiently transfer ownership.
- Create a **ResourceManager class** that stores multiple resources efficiently.
- Demonstrate **move operations** with std::move().

---

## Step 3: Implement the Resource Class

The **Resource** class will manage a **dynamically allocated integer array**, ensuring proper **memory allocation and deallocation**.

```
#include <iostream>

#include <utility>  // For std::move
```

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

public:
    // Constructor
    Resource(size_t s) : size(s) {
        data = new int[size];
        std::cout << "Resource allocated for " << size << " elements.\n";
    }

    // Destructor
    ~Resource() {
        delete[] data;
        std::cout << "Resource deallocated.\n";
    }

    // Copy Constructor (Deleted to enforce move semantics)
    Resource(const Resource&) = delete;
```

```
// Copy Assignment Operator (Deleted to enforce move semantics)
Resource& operator=(const Resource&) = delete;

// Move Constructor
Resource(Resource&& other) noexcept {
    data = other.data;  // Transfer ownership
    size = other.size;
    other.data = nullptr;  // Prevent the old object from deleting the
resource
    other.size = 0;
    std::cout << "Resource moved (Move Constructor).\n";
}

// Move Assignment Operator
Resource& operator=(Resource&& other) noexcept {
    if (this != &other) {
        delete[] data;  // Free existing resource

        data = other.data;  // Transfer ownership
        size = other.size;
        other.data = nullptr;
```

```
        other.size = 0;
        std::cout << "Resource move-assigned (Move Assignment Operator).\n";
      }
      return *this;
    }

    // Function to display data size
    void showSize() {
      if (data)
        std::cout << "Resource size: " << size << " elements.\n";
      else
        std::cout << "Resource is empty.\n";
    }
};
```

### Key Features in Resource Class

✅ **Move Constructor:** Transfers ownership when an object is initialized from another temporary object.

✅ **Move Assignment Operator:** Handles reassignment of resources, preventing memory leaks.

✅ **Deleted Copy Constructor & Assignment Operator:** Enforces move semantics by preventing deep copies.

---

## STEP 4: IMPLEMENT THE RESOURCE MANAGER CLASS

The **ResourceManager** class will efficiently store and manage multiple Resource objects.

```
#include <vector>


class ResourceManager {

private:

   std::vector<Resource> resources;  // Stores multiple resources


public:

   // Add resources using move semantics

   void addResource(Resource&& res) {

      resources.push_back(std::move(res)); // Move instead of copy

   }


   // Display all resources

   void showResources() {

      for (auto& res : resources)

         res.showSize();

   }

};
```

### Key Features in ResourceManager

✅ **Stores Resource objects efficiently using std::vector.**
✅ **Uses std::move() to prevent unnecessary copying.**
✅ **Provides a showResources() function to display resource information.**

---

## STEP 5: TESTING THE SYSTEM

We test **move semantics** by creating Resource objects and moving them into the ResourceManager.

```
int main() {
    ResourceManager manager;


    // Creating resources and moving them to the manager
    Resource res1(10);
    Resource res2(20);


    manager.addResource(std::move(res1));
    manager.addResource(std::move(res2));


    // Display all resources managed
    manager.showResources();
```

```
    return 0;
}
```

---

## Step 6: Expected Output

Resource allocated for 10 elements.

Resource allocated for 20 elements.

Resource moved (Move Constructor).

Resource moved (Move Constructor).

Resource size: 10 elements.

Resource size: 20 elements.

Resource deallocated.

Resource deallocated.

**What Happens?**

1. **Two Resource objects are created**.
2. **Ownership is transferred to ResourceManager** via std::move().
3. **Original res1 and res2 become empty (nullified)**.
4. **Objects in ResourceManager are properly managed and deallocated at the end**.

---

## Step 7: Benefits of Move Semantics in this System

**1. Eliminates Unnecessary Copies**

- Without move semantics, every insertion in std::vector would require **deep copying** of the entire object, causing **memory inefficiencies**.
- Using std::move(), **ownership is transferred**, avoiding unnecessary memory allocation.

**2. Improves Performance**

- Copying large data structures (like arrays, vectors) is expensive.
- Move semantics **reduce overhead**, ensuring fast execution.

**3. Prevents Memory Leaks**

- The move constructor prevents duplicate memory allocations.
- The move assignment operator ensures previously allocated memory is properly freed before transfer.

---

## STEP 8: ADDITIONAL EXERCISES

1. **Modify Resource to manage a dynamically allocated string instead of an array.**
2. **Enhance ResourceManager to allow resource deletion.**
3. **Implement a function that returns a Resource using move semantics instead of copying.**
4. **Measure performance differences between copy-based and move-based implementations.**

---

## STEP 9: SUMMARY

| Feature | Without Move Semantics | With Move Semantics |
|---|---|---|
| **Memory Usage** | High (copies data) | Low (transfers ownership) |
| **Performance** | Slow (deep copies) | Fast (avoids extra allocation) |
| **Memory Leaks** | Likely if not managed | Prevented by automatic deallocation |

✅ **Move semantics enable efficient resource management, improve performance, and prevent memory leaks**.

---

## CONCLUSION

This assignment demonstrates how **move semantics (&& and std::move())** improve memory efficiency in a **Resource Management System**. Using **move constructors, move assignment operators, and smart ownership transfer**, we optimize performance and **eliminate unnecessary memory usage** in C++.