C++ Pointers and Memory Management: A Study Guide

Generated by Gemini

November 15, 2025

1. Summary of Pointers and Memory Management

A **pointer** is a variable that stores the **memory address** of another variable. Understanding memory management is critical in C++ to prevent leaks and runtime errors.

Pointers Fundamentals and Operators

The table below summarizes the core operators used with raw pointers.

Concept	Operator	Description
Declaration	`*`	Declares a variable as a pointer (e.g., `int *pint;`).
Address-of	`&`	Returns the memory address of a variable (e.g., `pint = &z;`).
Dereference	`*`	Accesses the value stored at the memory address the pointer holds (e.g., `cout << *pint;`).
Null Pointer	`nullptr`	A constant indicating the pointer points to nothing valid.
Size	`sizeof()`	The size of the pointer variable itself is constant (e.g., 8 bytes) regardless of the data type it points to.

Passing Data to Functions

- **Pass by Reference (`&`):** Uses an alias in the function header (e.g., `void swap(int &a)`). Modifies the original data directly. Cannot handle nullability.
- **Pass by Address (`*`):** Uses pointers (e.g., `void swap(int *px)`). Requires passing the address (`&x`) and using dereference (`*px`). Can safely check for `nullptr`.

Dynamic Memory (Raw Pointers)

Memory allocated on the Heap at runtime. This requires manual management.

- **Allocation:** Use the `new` keyword (e.g., `int *p = new int[size];`).
- **Deallocation:** Use `delete` (for single objects) or `delete[]` (for arrays) to manually free memory.
- **Safety:** After deleting, the pointer **must** be set to `nullptr` to avoid a **dangling pointer**.

Modern Smart Pointers

Smart pointers are introduced in modern C++ (<memory>) to automate memory cleanup.

- `unique_ptr`**: Exclusive ownership**. Only one pointer can own the resource. Automatic cleanup when scope ends. Ownership is transferred using `std::move()`.
- `shared_ptr`**: Shared ownership**. Multiple pointers can point to the same resource. Uses a reference counter; memory is freed only when the last `shared_ptr` goes out of scope.

2. C++ Code Examples for Study

The following code snippets are complete and runnable, demonstrating the concepts summarized above. They all use `using namespace std;`.

2.1. Pointer Basics

```cpp
#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

// Define a simple structure to demonstrate pointer size regardless of object size
struct Employee {
    string name;
    int id;
    double salary;
};

// Main function demonstrating basic pointer concepts
int main() {
    cout << "--- 1. Pointer Basics ---\n\n";

    // 1. Basic Declaration, Address-Of (&), and Dereferencing (*)
    int z = 99;

    // Declare an integer pointer and store the address of 'z'
    int *pint = &z;

    cout << "Value of z: " << z << "\n";
    cout << "Address of z (&z): " << &z << "\n";
    cout << "Value in pint (address of z): " << pint << "\n";
    cout << "Value at address *pint (dereferencing): " << *pint << "\n\n";

    // 2. Modifying the variable using the pointer
    *pint = 10;
    cout << "Action: Set *pint = 10.\n";
    cout << "New value of z: " << z << "\n\n";

    // 3. Pointer Size Check (Size is constant, regardless of type pointed to)
    char ch = 'a';
    char *pch = &ch;
    Employee emp;
    Employee *pemp = &emp;

    cout << "--- Size Check ---\n";
    cout << "Size of char* (pch): " << sizeof(pch) << " bytes\n";
    cout << "Size of Employee* (pemp): " << sizeof(pemp) << " bytes\n";
    cout << "Size of Employee struct: " << sizeof(Employee) << " bytes\n\n";

    // 4. Null Pointer
    pint = nullptr; // Setting the pointer to null (points to nothing)
    if (pint == nullptr) {
        cout << "'pint' is now a nullptr and does not point to a valid object.\n
           ";
```

```
    }

    return 0;
}
```

Listing 1: pointer_basics.cpp

2.2. Function Parameters (Value, Reference, Address)

```
#include <iostream>
#include <utility>

using namespace std;

// 1. Pass by Value: Cannot change original variables.
void swap_by_value(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
    cout << "  -> Inside swap_by_value: a=" << a << ", b=" << b << " (Originals
       NOT changed)\n";
}

// 2. Pass by Reference: Uses an alias (&) to directly modify the originals.
void swap_by_reference(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
    cout << "  -> Inside swap_by_reference: a=" << a << ", b=" << b << " (
       Originals ARE changed)\n";
}

// 3. Pass by Address (Pointers): Takes addresses and uses * to modify originals
    .
void swap_by_address(int *px, int *py) {
    // Pointers allow for checking if an address is valid
    if (px == nullptr || py == nullptr) {
        cout << "  -> ERROR: Cannot swap. A null pointer was passed.\n";
        return;
    }

    int temp = *px; // Dereference to read the value
    *px = *py;      // Dereference to write the new value
    *py = temp;     // Dereference to write the new value
    cout << "  -> Inside swap_by_address: *px=" << *px << ", *py=" << *py << " (
       Originals ARE changed)\n";
}

int main() {
    int x, y;

    cout << "--- 2. Function Parameters ---\n\n";

    // Pass by Value Test
    x = 5; y = 7;
    cout << "1. Pass by Value (Original: x=" << x << ", y=" << y << ")\n";
    swap_by_value(x, y);
```

```
    cout << "   After swap_by_value: x=" << x << ", y=" << y << " (No Change)\n\
        n";

    // Pass by Reference Test
    x = 5; y = 7;
    cout << "2. Pass by Reference (Original: x=" << x << ", y=" << y << ")\n";
    swap_by_reference(x, y);
    cout << "   After swap_by_reference: x=" << x << ", y=" << y << " (Swapped!)
        \n\n";

    // Pass by Address Test
    x = 5; y = 7;
    cout << "3. Pass by Address (Original: x=" << x << ", y=" << y << ")\n";
    // Must pass the address using the & operator
    swap_by_address(&x, &y);
    cout << "   After swap_by_address: x=" << x << ", y=" << y << " (Swapped!)\n
        \n";

    // Pass by Address with Null
    cout << "4. Pass by Address with Null (Safe Pointer Check):\n";
    swap_by_address(&x, nullptr);

    return 0;
}
```

Listing 2: function_parameters.cpp

2.3. Raw Pointers and Dynamic Memory (The Heap)

```cpp
#include <iostream>
#include <numeric>

using namespace std;

int main() {
    cout << "--- 3. Raw Dynamic Memory (Heap) ---\n\n";

    int size;
    cout << "Enter the size of the dynamic array: ";
    if (!(cin >> size) || size <= 0) {
        cerr << "Invalid size entered.\n";
        return 1;
    }

    // 1. Allocation on the Heap using 'new[]'
    int *parr = new int[size];
    cout << "Allocated array of size " << size << " on the Heap.\n\n";

    // Use a temporary pointer to traverse the array
    int *pcurrent = parr;

    // 2. Input and Traversal using Pointer Arithmetic
    cout << "Enter " << size << " integer values:\n";
    for (int i = 0; i < size; i++) {
        cout << "Element " << i + 1 << ": ";
        cin >> *pcurrent; // Read value into the memory pointed to by pcurrent
        pcurrent++;       // Move pointer to the next element
    }

    // Reset pcurrent to the start to calculate sum
    pcurrent = parr;

    // 3. Summation
    long long sum = 0;
    for (int i = 0; i < size; i++) {
        sum += *pcurrent;
        pcurrent++;
    }
    cout << "\nTotal Sum: " << sum << "\n\n";

    // 4. Deallocation and Safety (CRITICAL MANUAL STEP)
    // You MUST use delete[] for memory allocated with new[]
    delete[] parr;
    cout << "Memory DEALLOCATED using delete[].\n";

    // Set the pointer to nullptr to prevent a "dangling pointer"
    parr = nullptr;
    cout << "Pointer 'parr' set to nullptr (safe).\n";

    return 0; // No memory leak!
}
```

Listing 3: raw_dynamic_memory.cpp

2.4. Modern Smart Pointers (`unique_ptr` and `shared_ptr`)

```cpp
#include <iostream>
#include <memory>  // Required for smart pointers
#include <utility> // Required for move

using namespace std;

int main() {
    cout << "--- 4. Modern Smart Pointers ---\n\n";

    // 1. UNIQUE_PTR (Exclusive Ownership)
    cout << "1. unique_ptr (Exclusive Ownership)\n";

    unique_ptr<int> p1 = make_unique<int>(40);
    cout << "   Initial p1 value: " << *p1 << "\n";

    // Ownership can only be transferred (moved)
    unique_ptr<int> p2 = move(p1);

    cout << "   Ownership moved from p1 to p2.\n";

    // Check if p1 still owns the resource
    if (!p1) {
        cout << "   p1 is now nullptr and no longer owns the resource.\n";
    }
    cout << "   p2 value: " << *p2 << "\n\n";

    // Resource is automatically deleted when p2 goes out of scope.


    // 2. SHARED_PTR (Shared Ownership)
    cout << "2. shared_ptr (Shared Ownership)\n";

    shared_ptr<int> s1 = make_shared<int>(50);
    cout << "   s1 value: " << *s1 << "\n";
    cout << "   s1 use count: " << s1.use_count() << "\n";

    // Copying is allowed, which increments the reference counter
    auto s2 = s1;
    auto s3 = s2;

    cout << "   s2 and s3 created (shared ownership).\n";
    cout << "   Current use count: " << s1.use_count() << "\n"; // Should be 3

    // Modifying through any pointer affects the single shared object
    *s2 = 88;
    cout << "   Value modified to 88. s1 value: " << *s1 << ", s3 value: " << *
        s3 << "\n";

    // The resource is AUTOMATICALLY DELETED only when the last shared pointer (
        s1, s2, or s3) goes out of scope.

    return 0;
}
```

Listing 4: smart_pointers.cpp