Chapter 7: Raw Memory Management (`new` and `delete`)

This chapter dives into manual memory management. While modern C++ provides powerful tools (smart pointers) to automate this, understanding how `new` and `delete` work is essential for comprehending the language's memory model, for working with legacy code, and for certain advanced, low-level optimizations.

7.1 Dynamic Allocation Operators

Why do we need dynamic allocation? C++ has two primary memory regions: the **stack** and the **heap** (or free store). * **Stack:** Used for local variables, function parameters, etc. It's extremely fast. Memory is allocated and deallocated automatically when a variable enters or leaves scope. However, its size is limited, and its lifetime is strictly tied to scope. * **Heap:** A large pool of memory available to the programmer. It's more flexible; an object allocated on the heap can live as long as you want it to, independent of any scope. However, this flexibility comes at a cost: it's slightly slower, and **you** are responsible for telling the system when you are finished with the memory.

You use dynamic allocation with `new` when an object's lifetime must extend beyond the scope that creates it.

The Two-Step Process of `new` and `delete`

What makes `new` different from `malloc` in C? `new` and `delete` are not simple memory functions; they are fundamental operators that understand object lifetime.

- **`new MyClass()` does two things:**
  1. **Allocates Memory:** It calls an underlying function (like `malloc`) to get a block of raw memory of the correct size from the heap.
  2. **Constructs Object:** It calls the `MyClass` constructor on that raw memory to turn it into a living object.
- **`delete p` does two things:**
  1. **Destroys Object:** It calls the object's destructor (`~MyClass()`) to allow the object to clean up its own resources.
  2. **Deallocates Memory:** It calls an underlying function (like `free`) to return the memory to the system.

This constructor/destructor integration is the key feature that makes `new` and `delete` fundamental to C++.

The Unforgivable Sin: Mismatched `new[]` and `delete`

This is one of the most critical rules in manual memory management. * `new` is paired with `delete`. * `new[]` is paired with `delete[]`.

Mismatching them is **Undefined Behavior**.

Why? When you write `new MyClass[10]`, the memory allocation system typically stores the number of objects (10) in a hidden location just before the allocated array. * When you call `delete[] p_array`, it looks for that hidden number and calls the destructor 10 times before freeing the whole block. * When you call plain `delete p_array`, it assumes it's a single object. It calls the destructor only once and frees a smaller amount of memory, leading to a definite leak of the other 9 objects and likely heap corruption.

```cpp
class MyClass { public: ~MyClass(){/*...*/} };

MyClass* p_array = new MyClass[3];

// WRONG! Undefined Behavior. Only one destructor is called.
delete p_array;

// CORRECT! Three destructors are called.
delete[] p_array;
```

7.2 Placement New and Custom Allocation

Why would you use this? Placement `new` is an advanced feature for performance-critical applications where you want to separate memory allocation from object construction. Common use cases include: * **Memory Pools:** Pre-allocating a large chunk of memory and then rapidly constructing/destructing objects within it to avoid the overhead of many separate `new`/`delete` calls. * **Hardware Interaction:** Constructing an object at a specific memory address that corresponds to a hardware register.

What is it? Placement `new` does **not** allocate any memory. It is an overload of `new` that takes a pointer to a pre-allocated buffer. It only performs step 2 of the `new` process: it calls the constructor on the memory you provided.

How to use it (and its dangers):

```cpp
#include <new> // Required for placement new

// 1. Allocate a buffer of raw memory (e.g., on the stack)
alignas(MyClass) char buffer[sizeof(MyClass)];

// 2. Use placement new to construct an object in the buffer
MyClass* p_placed = new (buffer) MyClass(); // Calls constructor

// ... use p_placed ...

// DANGER: You cannot `delete` a pointer from placement new.
// You must manually and explicitly call the destructor.

// 1. Manually call the destructor
p_placed->~MyClass();

// 2. The buffer memory is freed automatically since it was on the stack.
```

7.3 Handling Memory Allocation Failure

Why handle failures? The heap is finite. `new` can fail if the system runs out of memory. If you don't have a strategy for this, your program will crash.

What are the strategies? 1. **The Default: `std::bad_alloc` Exception** By default, if `new` fails, it throws an exception of type `std::bad_alloc`. This is the preferred modern C++ approach, as it allows for centralized error handling in a `catch` block.

```
```cpp
try {
    // Attempt to allocate an impossibly large amount of memory
    int* huge_array = new int[1000000000000ULL];
} catch (const std::bad_alloc& e) {
    std::cerr << "Allocation failed: " << e.what() << std::endl;
    // Attempt to recover or terminate gracefully
}
```
```

2. **The `nothrow` Alternative** For codebases that do not use exceptions (common in some embedded systems or older C++ styles), you can request that `new` return `nullptr` on failure instead of throwing.

   ```cpp
   // This will not throw an exception on failure
   int* p_nothrow = new (std::nothrow) int[1000000000000ULL];

   if (p_nothrow == nullptr) {
       std::cerr << "Allocation failed, returned nullptr."
       // Handle the error without using exceptions
   }
   ```

Projects for Chapter 7

Project 1: The Leaky Array

- **Problem Statement:** Write a program containing a function that allocates an array of 1000 `int`s on the heap using `new int[1000]`. Call this function inside a `for` loop that runs 10,000 times. Do **not** deallocate the memory. Run your program and observe its memory usage climb using your operating system's task manager. Then, run it under a memory checker like Valgrind to see the leak report. Finally, fix the code by adding the correct `delete[]` call and verify that the leak is gone.
- **Core Concepts to Apply:** `new[]`, `delete[]`, memory leaks, using diagnostic tools.

Project 2: The Destructor Mismatch

- **Problem Statement:** Create a simple `struct Tracker` that prints `"Constructed\n"` in its constructor and `"Destroyed\n"` in its destructor. In `main`, allocate an array of 5 `Tracker` objects using `new[]`. Then, deallocate the array using plain `delete` instead of `delete[]`. Count the number of "Destroyed" messages printed. In comments, explain why the count is wrong. Fix the code to use `delete[]` and verify that 5 "Destroyed" messages are printed.
- **Core Concepts to Apply:** `new[]`, the difference between `delete` and `delete[]`, constructors, destructors, resource leaks.

Project 3: The Placement `new` Custom Buffer

- **Problem Statement:** Inside `main`, allocate a raw character buffer on the stack large enough to hold three instances of a `Tracker` struct (from Project 2). Use a `for` loop and placement `new` to construct three `Tracker` objects within this buffer at different offsets. After the loop, print a message. Then, write another `for` loop that explicitly calls the destructor for each of the three objects in reverse order of construction. Run the program and verify that the constructor and destructor messages are balanced.
- **Core Concepts to Apply:** Placement `new`, manual destructor calls (`ptr->~MyClass()`), stack-based buffers, pointer arithmetic.
- **Hint:** `alignas(Tracker) char buffer[3 * sizeof(Tracker)];` will ensure your buffer has the correct alignment for `Tracker` objects.

"'