C++ Pointers and Smart Pointers Cheat Sheet

1. Pointers (Raw Pointers)

A pointer is a variable that stores the memory address of another variable. Instead of holding a direct value, it "points" to the location in memory where the actual data is stored. Pointers are fundamental to C++ as they allow for direct memory manipulation, dynamic memory allocation, and efficient handling of data structures.

Key Concepts of Raw Pointers:

- **Memory Address:** Every variable in a program is stored at a specific memory location, which has a unique address. A pointer holds this address.
- **Declaration:** To declare a pointer, you use the asterisk (`*`) operator. The type of the pointer must match the type of the variable it points to.
- **Address-of Operator (`&`):** This operator is used to get the memory address of a variable.
- **Dereference Operator (`*`):** This operator is used to access the value stored at the memory address pointed to by a pointer.
- **Null Pointer:** A pointer that does not point to any valid memory location. It's good practice to initialize pointers to `nullptr` (or `NULL` in older C++) if they are not immediately assigned a valid address.

Declaring and Using Pointers:

```cpp
#include <iostream>

int main() {
    int var = 10;  // Declare an integer variable

    // Declare a pointer to an integer and initialize it with the address of 'var'
    int* ptr = &var;

    std::cout << "Value of var: " << var << std::endl;        // Output: 10
    std::cout << "Address of var: " << &var << std::endl;      // Output: (e.g., 0x7ffeefbff53c)
    std::cout << "Value of ptr (address of var): " << ptr << std::endl; // Output: (e.g., 0x7ffeefbff53c)
    std::cout << "Value pointed to by ptr: " << *ptr << std::endl; // Output: 10

    // Change the value through the pointer
    *ptr = 20;
    std::cout << "New value of var: " << var << std::endl;    // Output: 20

    // Null pointer
    int* nullPtr = nullptr;
    // std::cout << *nullPtr; // Dereferencing a null pointer leads to undefined behavior/crash

    return 0;
}
```

Pointers and Arrays:

In C++, array names can often be treated as pointers to their first element. This close relationship allows for flexible memory access.

```cpp
#include <iostream>

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int* p = arr; // p points to the first element of arr (arr[0])

    std::cout << "Value of arr[0] using pointer: " << *p << std::endl; // Output: 10

    p++; // Increment pointer to point to the next element (arr[1])
    std::cout << "Value of arr[1] using pointer: " << *p << std::endl; // Output: 20

    // Accessing elements using pointer arithmetic
    std::cout << "Value of arr[2] using pointer arithmetic: " << *(arr + 2) << std::endl; // Output: 30

    return 0;
}
```

Dynamic Memory Allocation with Pointers (`new` and `delete`):

Raw pointers are commonly used for dynamic memory allocation, where memory is allocated during program execution (runtime) from the heap. The `new` operator is used to allocate memory, and the `delete` operator is used to deallocate it.

```cpp
#include <iostream>

int main() {
    // Allocate memory for a single integer
    int* dynamicInt = new int;
    *dynamicInt = 100;
    std::cout << "Dynamically allocated int: " << *dynamicInt << std::endl;
    delete dynamicInt; // Deallocate memory
    dynamicInt = nullptr; // Good practice to set to nullptr after deletion

    // Allocate memory for an array of integers
    int* dynamicArray = new int[5];
    for (int i = 0; i < 5; ++i) {
        dynamicArray[i] = (i + 1) * 10;
    }
    std::cout << "Dynamically allocated array elements: ";
    for (int i = 0; i < 5; ++i) {
        std::cout << dynamicArray[i] << " ";
    }
    std::cout << std::endl;
    delete[] dynamicArray; // Deallocate array memory
    dynamicArray = nullptr;

    return 0;
}
```

Issues with Raw Pointers:

While powerful, raw pointers come with significant responsibilities and potential pitfalls:

- **Memory Leaks:** Forgetting to `delete` dynamically allocated memory leads to memory leaks, where memory is consumed but never released, eventually exhausting system resources.
- **Dangling Pointers:** Pointers that point to memory that has already been deallocated. Dereferencing a dangling pointer leads to undefined behavior.
- **Double Free:** Attempting to `delete` the same memory twice, also leading to undefined behavior.
- **Wild Pointers:** Uninitialized pointers that contain garbage values, pointing to arbitrary memory locations. Dereferencing them can cause crashes or data

corruption.

- **Ownership Ambiguity:** It's often unclear which part of the code is responsible for `deleting` a raw pointer, especially when pointers are passed between functions.

These issues make raw pointer management error-prone and are a common source of bugs in C++ programs. This is where smart pointers come into play.

2. Smart Pointers

Smart pointers are objects that act like pointers but provide automatic memory management, significantly reducing the risk of memory leaks and dangling pointers. They are defined in the `<memory>` header and were introduced to the C++ Standard Library in C++11. Smart pointers implement the Resource Acquisition Is Initialization (RAII) principle, meaning that resources (like dynamically allocated memory) are acquired during object creation and automatically released when the object goes out of scope.

Advantages of Smart Pointers:

- **Automatic Memory Management:** No need for manual `delete` calls.
- **Exception Safety:** Memory is automatically released even if exceptions occur.
- **Clear Ownership Semantics:** Each type of smart pointer clearly defines its ownership model.
- **Reduced Bugs:** Eliminates common pointer-related errors like memory leaks, dangling pointers, and double frees.

C++ provides three main types of smart pointers:

- `std::unique_ptr`
- `std::shared_ptr`
- `std::weak_ptr`

2.1. `std::unique_ptr`

`std::unique_ptr` is a smart pointer that owns the object it points to exclusively. This means that only one `unique_ptr` can point to a particular object at any given time. When the `unique_ptr` goes out of scope, the object it owns is automatically deleted.

Key Characteristics:

- **Exclusive Ownership:** Cannot be copied, only moved. This ensures that there's always a single owner.
- **Lightweight:** Has minimal overhead, similar to a raw pointer.
- **Automatic Deallocation:** The owned object is automatically deleted when the `unique_ptr` is destroyed.
- **Use Case:** Ideal for managing dynamically allocated objects that have a single, clear owner.

Declaration and Usage:

```cpp
#include <iostream>
#include <memory> // Required for unique_ptr

class MyClass {
public:
    MyClass() { std::cout << "MyClass Constructor" << std::endl; }
    ~MyClass() { std::cout << "MyClass Destructor" << std::endl; }
    void greet() { std::cout << "Hello from MyClass!" << std::endl; }
};

int main() {
    // Create a unique_ptr
    std::unique_ptr<MyClass> ptr1(new MyClass()); // C++11 way
    // Or using make_unique (preferred in C++14 and later for safety and
efficiency)
    // std::unique_ptr<MyClass> ptr1 = std::make_unique<MyClass>();

    ptr1->greet(); // Access member function

    // Transfer ownership (ptr1 becomes null, ptr2 now owns the object)
    std::unique_ptr<MyClass> ptr2 = std::move(ptr1);

    if (ptr1 == nullptr) {
        std::cout << "ptr1 is now null." << std::endl;
    }

    ptr2->greet();

    // When ptr2 goes out of scope, MyClass object will be destructed
automatically

    return 0;
}
```

2.2. `std::shared_ptr`

`std::shared_ptr` is a smart pointer that manages shared ownership of an object. Multiple `shared_ptr` instances can point to the same object. It maintains a reference count, which tracks how many `shared_ptr` instances are currently pointing to the object. When the last `shared_ptr` pointing to the object is destroyed or reset, the object is automatically deleted.

Key Characteristics:

- **Shared Ownership:** Multiple `shared_ptr`s can own the same object.
- **Reference Counting:** Internally manages a reference count. The object is deleted when the count drops to zero.
- **Automatic Deallocation:** Object is deleted when no `shared_ptr`s refer to it.

- **Use Case:** Ideal for scenarios where multiple parts of the code need to share ownership of an object, and the object should only be destroyed when all owners are done with it.

Declaration and Usage:

```
 #include <iostream>
#include <memory> // Required for shared_ptr

class Resource {
public:
    Resource() { std::cout << "Resource Created" << std::endl; }
    ~Resource() { std::cout << "Resource Destroyed" << std::endl; }
    void doSomething() { std::cout << "Doing something with Resource" <<
std::endl; }
};

void processResource(std::shared_ptr<Resource> res) {
    std::cout << "Inside processResource. Reference count: " << res.use_count()
<< std::endl;
    res->doSomething();
}

int main() {
    // Create a shared_ptr using make_shared (preferred for safety and
efficiency)
    std::shared_ptr<Resource> s_ptr1 = std::make_shared<Resource>();
    std::cout << "Initial reference count: " << s_ptr1.use_count() <<
std::endl; // Output: 1

    // Create another shared_ptr that shares ownership
    std::shared_ptr<Resource> s_ptr2 = s_ptr1;
    std::cout << "Reference count after copy: " << s_ptr1.use_count() <<
std::endl; // Output: 2

    // Pass by value to a function (increases reference count)
    processResource(s_ptr1);
    std::cout << "Reference count after function call: " << s_ptr1.use_count()
<< std::endl; // Output: 2

    // Reset one shared_ptr (decreases reference count)
    s_ptr2.reset();
    std::cout << "Reference count after reset: " << s_ptr1.use_count() <<
std::endl; // Output: 1

    // When s_ptr1 goes out of scope, the object will be destroyed

    return 0;
}
```

2.3. `std::weak_ptr`

`std::weak_ptr` is a non-owning smart pointer. It holds a "weak" reference to an object managed by a `std::shared_ptr`. It does not contribute to the reference count of the `shared_ptr`, meaning it won't prevent the object from being deleted. `weak_ptr` is primarily used to break circular references between `shared_ptr`s, which would otherwise lead to memory leaks.

Key Characteristics:

- **Non-Owning:** Does not increase the reference count of the managed object.
- **Observing:** Used to observe an object without affecting its lifetime.
- **Cannot be Dereferenced Directly:** Must be converted to a `shared_ptr` using the `lock()` method before accessing the managed object. If the object has already been deleted, `lock()` will return a `nullptr`.
- **Use Case:** Breaking circular dependencies between `shared_ptr`s, caching, and optional object access.

Declaration and Usage:

```cpp
 #include <iostream>
#include <memory> // Required for shared_ptr and weak_ptr

class Node {
public:
    std::shared_ptr<Node> next;
    std::weak_ptr<Node> prev; // Use weak_ptr to break circular dependency
    int value;

    Node(int val) : value(val) { std::cout << "Node " << value << " Created" <<
std::endl; }
    ~Node() { std::cout << "Node " << value << " Destroyed" << std::endl; }
};

int main() {
    std::shared_ptr<Node> node1 = std::make_shared<Node>(1);
    std::shared_ptr<Node> node2 = std::make_shared<Node>(2);

    node1->next = node2; // node1 owns node2
    node2->prev = node1; // node2 weakly observes node1

    // Accessing through weak_ptr
    if (std::shared_ptr<Node> lockedNode1 = node2->prev.lock()) {
        std::cout << "Node2's previous node value: " << lockedNode1->value <<
std::endl;
    } else {
        std::cout << "Node2's previous node is no longer valid." << std::endl;
    }

    // When node1 and node2 go out of scope, they will be properly destroyed
    // without weak_ptr, node1 and node2 would form a circular reference,
    // preventing their destruction and causing a memory leak.

    return 0;
}
```

3. Choosing the Right Pointer

Feature	Raw Pointer	`std::unique_ptr`	`std::shared_ptr`	`std::weak_ptr`
Ownership	Manual	Exclusive	Shared (reference counted)	Non-owning (observing)
Memory Mgmt.	Manual (`new` / `delete`)	Automatic	Automatic	None (observes `shared_ptr`)
Copyable	Yes (copies address)	No (move-only)	Yes	Yes
Circular Ref.	N/A	N/A	Yes (can cause leaks if not managed)	Breaks circular refs with `shared_ptr`
Overhead	Minimal	Minimal	Moderate (control block for ref count)	Minimal
Use Cases	Legacy code, low-level hardware interaction, when ownership is clear and simple. Generally avoid in modern C++ for heap objects.	Single ownership, function returns unique object, PIMPL idiom.	Multiple owners, object lifetime tied to last owner, data structures like trees/graphs.	Breaking circular references, caching, observing objects without extending their lifetime.

Conclusion

Pointers are a fundamental and powerful feature in C++ for direct memory management. However, raw pointers introduce complexities and potential pitfalls like memory leaks and dangling pointers. Smart pointers (`std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`) were introduced to address these issues by providing automatic memory management and clear ownership semantics. By leveraging smart pointers, C++ developers can write safer, more robust, and more maintainable code, significantly reducing common memory-related bugs. Understanding when and how to use each type of pointer is crucial for effective modern C++ programming.

References

[1] W3Schools. "C++ Pointers". Available at: https://www.w3schools.com/cpp/cpp_pointers.asp [2] GeeksforGeeks. "C++ Pointers". Available at: https://www.geeksforgeeks.org/cpp-pointers/ [3] Microsoft Learn. "Pointers (C++)". Available at: https://learn.microsoft.com/en-us/cpp/cpp/pointers-cpp?view=msvc-170 [4] Microsoft Learn. "Smart pointers (Modern C++)". Available at: https://learn.microsoft.com/en-us/cpp/cpp/smart-pointers-modern-cpp?view=msvc-170 [5] GeeksforGeeks. "Smart Pointers in C++". Available at: https://www.geeksforgeeks.org/cpp/smart-pointers-cpp/ [6] cppreference.com. "std::unique_ptr". Available at: https://en.cppreference.com/w/cpp/memory/unique_ptr.html [7] cppreference.com. "std::shared_ptr". Available at: https://en.cppreference.com/w/cpp/memory/shared_ptr.html [8] cppreference.com. "std::weak_ptr". Available at: https://en.cppreference.com/w/cpp/memory/weak_ptr.html