Alright, let's delve into the world of Memory Management in C++.

IV. Memory Management in C++

- **Stack vs. Heap Memory:**
  - **Explanation:** C++ programs use two main areas of memory: the stack and the heap.
    - **Stack:** Used for static memory allocation. Memory is allocated and deallocated automatically by the system, typically when functions are called and return. Local variables declared within a function are usually stored on the stack. Stack memory is managed in a LIFO (Last-In, First-Out) manner. It's generally faster for allocation and deallocation. The size of the stack is usually limited.
    - **Heap:** Used for dynamic memory allocation. Memory is explicitly allocated and deallocated by the programmer using operators like new and delete. The heap provides more flexibility in terms of memory size and lifetime. Objects created on the

heap persist until they are explicitly deallocated. Managing heap memory requires careful attention to avoid memory leaks.

- **Example:**

```
C++
#include <iostream>

int globalVar = 10; // Stored in static memory (not stack or heap in the same way)

void myFunction() {
    int localVar = 20; // Stored on the stack
    int* heapVar = new int(30); // Stored on the heap

    std::cout << "Local variable on stack: " << localVar << std::endl;
    std::cout << "Variable on heap: " << *heapVar << std::endl;

    delete heapVar; // Deallocate memory from the heap
}

int main() {
    myFunction();
    return 0;
}
```

- **Dynamic Memory Allocation (new, new, delete, delete):**
  - **Explanation:** C++ provides operators to allocate and deallocate memory on the heap at runtime.
    - **new:** Allocates memory for a single object of

a specified type and returns a pointer to the allocated memory. If allocation fails, it throws a std::bad_alloc exception (unless the no-throw version new(std::nothrow) is used, which returns nullptr on failure).

- **new:** Allocates memory for an array of objects of a specified type and returns a pointer to the first element of the array.
- **delete:** Deallocates memory that was allocated using new for a single object. It's crucial to only delete memory that was allocated with new and to ensure that delete is called exactly once for each allocation.
- **delete:** Deallocates memory that was allocated using new for an array of objects. You must use delete to deallocate memory allocated with new. Using delete instead can lead to undefined behavior.

- **Examples:**

```
C++
#include <iostream>

int main() {
    // Allocate memory for a single integer
    int* singleIntPtr = new int;
    *singleIntPtr = 100;
```

```
    std::cout << "Value of single integer on heap: " << *singleIntPtr << std::endl;
    delete singleIntPtr; // Deallocate the memory

    // Allocate memory for an array of 5 integers
    int* intArrayPtr = new int[5];
    for (int i = 0; i < 5; ++i) {
        intArrayPtr[i] = i * 10;
    }
    std::cout << "Values in the integer array on heap: ";
    for (int i = 0; i < 5; ++i) {
        std::cout << intArrayPtr[i] << " ";
    }
    std::cout << std::endl;
    deleteintArrayPtr; // Deallocate the array memory

    return 0;
}
```

- **Pointers:**
  - **Explanation:** Variables that store memory addresses. They allow you to indirectly access and manipulate data stored at those addresses.
  - **Declaration:** data_type* pointer_name;
  - **Address-of Operator (&):** Used to get the memory address of a variable.
  - **Dereference Operator (*):** Used to access the value stored at the memory address held by a pointer.
  - **Pointer Arithmetic:** You can perform certain

arithmetic operations on pointers (e.g., incrementing or decrementing a pointer to move to the next or previous element in an array).

- **Null Pointers:** A pointer that does not point to any valid memory location (often represented by nullptr in C++11 and later).
- **Dangling Pointers:** Pointers that hold the address of memory that has already been deallocated. Dereferencing a dangling pointer leads to undefined behavior.
- **Example:**

C++

```
#include <iostream>

int main() {
    int number = 50;
    int* ptr = &number; // ptr stores the address of number

    std::cout << "Address of number: " << ptr << std::endl;
    std::cout << "Value at the address in ptr: " << *ptr << std::endl;

    *ptr = 100; // Modifying the value of number through the pointer
    std::cout << "New value of number: " << number << std::endl;

    int arr= {1, 2, 3};
    int* arrPtr = arr; // arr decays to a pointer to the first element
    std::cout << "First element of arr: " << *arrPtr << std::endl;
    arrPtr++; // Move to the next element
    std::cout << "Second element of arr: " << *arrPtr << std::endl;
```

```
    int* nullPtr = nullptr;
    // std::cout << *nullPtr << std::endl; // Dereferencing a null pointer is undefined behavior

    int* danglingPtr = new int(5);
    delete danglingPtr;
    // std::cout << *danglingPtr << std::endl; // Dereferencing a dangling pointer is undefined behavior

    return 0;
}
```

- **References:**
  - **Explanation:** An alias for an existing variable. Once a reference is initialized to a variable, it cannot be made to refer to another variable. References are often used for function parameters and return types to avoid copying.
  - **Declaration:** data_type& reference_name = variable_name;
  - **Properties:**
    - Must be initialized when declared.
    - Cannot be reassigned to refer to a different variable.
    - Acts as an alias for the original variable.
  - **Example:**

```
C++
#include <iostream>

int main() {
    int num = 25;
```

```
    int& ref = num; // ref is a reference to num

    std::cout << "Value of num: " << num << std::endl;
    std::cout << "Value of ref: " << ref << std::endl;

    ref = 30; // Modifying ref also modifies num
    std::cout << "New value of num: " << num << std::endl;
    std::cout << "New value of ref: " << ref << std::endl;

    // int anotherNum = 40;
    // ref = anotherNum; // Error: Cannot rebind a reference

    return 0;
}
```

- **Smart Pointers (C++11):**
  - **Explanation:** Class templates that behave like regular pointers but provide automatic memory management. They help prevent memory leaks by automatically deallocating the memory they manage when they are no longer in use.
  - **std::unique_ptr:** Represents exclusive ownership of a dynamically allocated object. Only one unique_ptr can point to an object at a time. When the unique_ptr goes out of scope, the object it manages is automatically deleted. unique_ptr cannot be copied, but it can be moved using std::move.

```C++
#include <iostream>
#include <memory>

int main() {
    std::unique_ptr<int> ptr1(new int(75)); // ptr1 owns the integer
    std::cout << "Value pointed to by ptr1: " << *ptr1 << std::endl;

    // std::unique_ptr<int> ptr2 = ptr1; // Error: Cannot copy unique_ptr
    std::unique_ptr<int> ptr3 = std::move(ptr1); // Ownership is transferred to ptr3
    if (ptr1) {
        std::cout << "ptr1 still points to: " << *ptr1 << std::endl; // This won't happen
    } else {
        std::cout << "ptr1 is now null." << std::endl;
    }
    std::cout << "Value pointed to by ptr3: " << *ptr3 << std::endl;

    // The integer will be automatically deleted when ptr3 goes out of scope
    return 0;
}
```

- **std::shared_ptr:** Allows multiple shared_ptr objects to own and manage the same dynamically allocated object. It uses a reference count to keep track of how many shared_ptr objects are pointing to the object. When the last shared_ptr that owns the object goes out of scope, the object is automatically deleted.

```C++
#include <iostream>
#include <memory>

int main() {
```

```
    std::shared_ptr<int> ptr1(new int(88));
    std::cout << "Value pointed to by ptr1: " << *ptr1 << ", Count: " << ptr1.use_count() << std::endl;

    std::shared_ptr<int> ptr2 = ptr1; // ptr2 also owns the same integer
    std::cout << "Value pointed to by ptr2: " << *ptr2 << ", Count: " << ptr2.use_count() << std::endl;

    {
        std::shared_ptr<int> ptr3 = ptr1;
        std::cout << "Value pointed to by ptr3: " << *ptr3 << ", Count: " << ptr3.use_count() << std::endl;
    } // ptr3 goes out of scope, count decreases

    std::cout << "Count after ptr3 goes out of scope: " << ptr1.use_count() << std::endl;

    ptr1.reset(); // Releases ownership
    std::cout << "Count after ptr1.reset(): " << ptr2.use_count() << std::endl;

    // The integer will be deleted when ptr2 goes out of scope (count becomes 0)
    return 0;
}
```

- **std::weak_ptr:** A non-owning pointer that holds a weak reference to an object managed by a std::shared_ptr. It does not increase the reference count. A weak_ptr can be used to break circular dependencies between shared_ptr objects, which can prevent the object from being deleted. You need to convert a weak_ptr to a shared_ptr (using .lock()) before you can access the managed

object. If the object has already been deleted, .lock() will return a null shared_ptr.

```C++
#include <iostream>
#include <memory>

int main() {
    std::shared_ptr<int> sharedPtr(new int(99));
    std::weak_ptr<int> weakPtr = sharedPtr;

    {
        std::shared_ptr<int> lockedPtr = weakPtr.lock(); // Try to get a shared_ptr
        if (lockedPtr) {
            std::cout << "Value from weak_ptr: " << *lockedPtr << ", Count: " << lockedPtr.use_count() << std::endl;
        } else {
            std::cout << "Object no longer exists." << std::endl;
        }
    }

    sharedPtr.reset(); // Object is deleted

    std::shared_ptr<int> lockedPtr2 = weakPtr.lock();
    if (lockedPtr2) {
        std::cout << "Value from weak_ptr again: " << *lockedPtr2 << std::endl;
    } else {
        std::cout << "Object no longer exists." << std::endl; // This will be printed
    }

    return 0;
}
```

- **Memory Leaks and Dangling Pointers:**
  - **Memory Leak:** Occurs when memory is allocated on the heap but is never deallocated,

even when it's no longer needed by the program. Over time, this can consume all available memory and cause the program to crash or the system to slow down. Memory leaks are a common problem in languages with manual memory management like C++. Smart pointers help to mitigate this.

- **Dangling Pointer:** A pointer that holds the address of memory that has already been freed (deallocated). Dereferencing a dangling pointer leads to undefined behavior, which can manifest as crashes, incorrect results, or security vulnerabilities.
- **Prevention:**
  - Always use delete for memory allocated with new and delete for memory allocated with new.
  - Set pointers to nullptr after deleting the memory they point to.
  - Prefer using smart pointers (std::unique_ptr, std::shared_ptr) to manage dynamically allocated memory automatically.

- **RAII (Resource Acquisition Is Initialization):**

- **Explanation:** A programming idiom (not a specific language feature) where resources (like memory, file handles, network connections, etc.) are acquired during the initialization of an object and are automatically released when the object goes out of scope (typically in the object's destructor). Smart pointers are a prime example of RAII in action for memory management. By tying the lifetime of a resource to the lifetime of an object, RAII helps to ensure that resources are always properly managed, even in the presence of exceptions.
- **Example (using std::unique_ptr):**

```C++
#include <iostream>
#include <fstream>
#include <memory>

class FileWriter {
private:
    std::unique_ptr<std::ofstream> fileStream;
public:
    FileWriter(const std::string& filename) : fileStream(new std::ofstream(filename))
{
        if (!fileStream->is_open()) {
            throw std::runtime_error("Could not open file: " + filename);
        }
    }
```

```
    void writeLine(const std::string& line) {
        *fileStream << line << std::endl;
    }

    // No explicit destructor needed, std::unique_ptr will close the file automatically
};

int main() {
    try {
        FileWriter writer("output.txt");
        writer.writeLine("This is the first line.");
        writer.writeLine("This is the second line.");
        // The file will be automatically closed when 'writer' goes out of scope
    } catch (const std::runtime_error& e) {
        std::cerr << "Error: " << e.what() << std::endl;
    }
    return 0;
}
```

- **Placement New:**
  - **Explanation:** A version of the new operator that allows you to construct an object at a specific, pre-allocated memory location. It does not allocate new memory from the heap; it simply calls the constructor of the object at the provided memory address. You need to be very careful when using placement new, as you are responsible for ensuring that the memory you are using is valid and large enough for the object. You also need to explicitly call the destructor of the object

when you are done with it before the memory is reused or deallocated.

- **Example:**

```C++
#include <iostream>
#include <new> // For placement new

class MyClass {
public:
    MyClass() { std::cout << "MyClass constructor called." << std::endl; }
    ~MyClass() { std::cout << "MyClass destructor called." << std::endl; }
    void printMessage() const { std::cout << "Hello from MyClass!" << std::endl; }
};

int main() {
    // Allocate raw memory on the stack
    alignas(MyClass) unsigned char buffer[sizeof(MyClass)];

    // Use placement new to construct an object of MyClass in the buffer
    MyClass* objPtr = new (buffer) MyClass();

    objPtr->printMessage();

    // Explicitly call the destructor
    objPtr->~MyClass();

    // The memory in 'buffer' is not deallocated by the destructor call
    // It will be reclaimed when 'buffer' goes out of scope

    return 0;
}
```

- **Overloading new and delete Operators:**
  - **Explanation:** You can overload the global new and delete operators, as well as the

class-specific versions, to customize memory allocation and deallocation behavior. This can be useful for things like custom memory management schemes, memory pooling, or debugging memory usage.

- Global Overloading: Affects all uses of `new` and `delete` in your program (unless[1] a class-specific version is defined).
- **Class-Specific Overloading:** Only affects the allocation and deallocation of objects of that particular class.
- **Example (Global Overloading - for demonstration purposes, use with caution):**

```C++
#include <iostream>
#include <cstdlib>

void* operator new(size_t size) {
    std::cout << "Custom global new operator called, size: " << size << " bytes." << std::endl;
    void* p = std::malloc(size);
    if (p == nullptr) {
        throw std::bad_alloc();
    }
    return p;
}

void operator delete(void* ptr) noexcept {
    std::cout << "Custom global delete operator called." << std::endl;
    std::free(ptr);
}
```

```
int main() {
    int* arr = new int[5];
    deletearr; // Note: You'd need to overload deleteas well for proper array handling
    return 0;
}
```

- **Memory Alignment:**
  - **Explanation:** The way data is arranged and accessed in computer memory. Modern processors can access memory more efficiently if data is aligned at certain boundaries (e.g., a 4-byte integer might be accessed faster if it starts at an address that is a multiple of 4). The compiler usually handles memory alignment automatically. However, in some low-level programming or when dealing with specific hardware requirements, you might need to control memory alignment. C++11 introduced the alignof operator (to get the alignment requirement of a type) and alignas specifier (to request a specific alignment for a variable or type).
  - **Example:**

```
C++
#include <iostream>
#include <cstddef> // For std::align_val_t
#include <new>     // For operator new with alignment
```

```cpp
struct alignas(16) AlignedData {
    int x;
    char y;
};

int main() {
    std::cout << "Alignment of AlignedData: " << alignof(AlignedData) << std::endl;

    // Allocating aligned memory using operator new with alignment (C++17)
    AlignedData* alignedPtr = new AlignedData;
    std::cout << "Address of alignedPtr: " << static_cast<void*>(alignedPtr) << std::endl;
    delete alignedPtr;

    return 0;
}
```

This concludes the section on Memory Management in C++. It's a critical area in C++ that requires careful understanding and practice. We will move on to Advanced C++ Concepts in the next response. Please let me know if you have any questions or would like any of these topics clarified further.

Sources

1. https://developer.aliyun.com/article/1463228