TASK 2

Irem Aslan

December 2024

0.1 C++ POINTERS

0.1.1 Pointers

In C++, pointers are variables that store the memory address of another variable. Instead of holding data directly, a pointer stores the location where the data is stored in memory.

Memory Address: Every variable in C++ is stored at a specific memory address. A pointer holds the address of a variable in memory rather than the variable's value.

Address	Value
100 "c"	5
101	
102 "a"	5
103	
104 "b"	102
105	

Table 1: How Do the Pointers Work?

```
/*assume that "a" ia located at the address 102, b is located at the
address 104 and c is located at the address 100. */


 int a= 5;
 pointer b= &a;
 int c=*b; //indirect addressing
```

Declaring a Pointer

Pointer Declaration: Pointers are declared using the * symbol in front of a variable type. The * tells the compiler that this variable is a pointer.

```
int* ptr;
```

Getting the Address of a Variable

Pointer Initialization: A pointer is initialized with the address of another variable using &.

```
int a=5;
int* ptr= &a;
```

Dereferencing a Pointer

Dereferencing: Dereferencing a pointer means accessing the value stored at the memory address that the pointer is pointing to. This is done using the * operator.

```
int a = 5;
int *ptr = &a;  // ptr holds the address of a
std::cout << *ptr;
```

Modifying the Value via Pointers

```
int a = 10;
int* ptr = &a;
*ptr = 20;
std::cout << a;
```

0.1.2 Garbage Collection Mechanism

Garbage collection is a memory management technique. It is a separate automatic memory management method that is used in programming languages where manual memory management is not preferred or done. In the manual memory management method, the user is required to mention the memory which is in use and which can be relocated, whereas the garbage collector collects the memory that is occupied by variables or objects that are no longer in use in the program. Only memory will be managed by garbage collectors; other resources such as destructors, user interaction window, or files will not be handled by the garbage collector.

Few languages need garbage collectors as part of the language for good efficiency. These languages are called garbage-collected languages. For example, Java, C, and most of the scripting languages need garbage collection as part of their functioning. However, languages such as C and C++ support manual memory management, which works similarly to the garbage collector.

0.1.3 Smart Pointers

1-void pointer

A void pointer in C++ is a special pointer that can point to objects of any data type. In other words, a void pointer is a general purpose pointer that can store the address of any data type and it can be typecasted to any type. A void pointer is not associated with any particular data type.

The size of a void pointer is different in different systems. In 16-bit systems, the size of a void pointer is 2 bytes. In a 32-bit system, the size of a void pointer is 4 bytes. And, in a 64-bit system, the size of a void pointer is 8 bytes.

```
int int_var = 63;
char char_var = 'f';

void *ptr = &int_var;   // Use the void keyword to declare a void pointer.

ptr = &char_var;  /*It is used for a void pointer to store the
                 address of an integer variable, and then the same
                pointer is used to store the address of a character variable..*/
```

Example

```
#include <iostream>
using namespace std;

enum class DataType {
    INT,
    FLOAT,
    STRING
};

void printData(void* ptr, DataType type) {
    switch (type) {
    case DataType::INT:
        cout << "Integer value: " << *(static_cast<int*>(ptr)) << endl;
        break;
    case DataType::FLOAT:
        cout << "Float value: " << *(static_cast<float*>(ptr)) << endl;
        break;
    case DataType::STRING:
        cout << "String value: " << static_cast<char*>(ptr) << endl;
        break;
    default:
        cout << "Invalid data type!" << endl;
    }
}

int main() {
    int intVar = 10;
    float floatVar = 20.5;
    char stringVar[] = "Hello!!";

    printData(&intVar, DataType::INT);
    printData(&floatVar, DataType::FLOAT);
    printData(&stringVar, DataType::STRING);

    return 0;
}
```

Advantages of Using Void Pointer

- Void pointers allow the creation of functions and data structures that can work with any data type. This makes them suitable for generic programming.

- The calloc() and malloc() functions return void* type (void pointers). This allows us to use these methods to allocate memory of

- In the C programming language, void pointers are used to implement generic functions. For example, the qsort() function in the standard C library .

2-nullptr

A NULL Pointer in C++ indicates the absence of a valid memory address in C++. It tells that the pointer is not pointing to any valid memory location In other words, it has the value "NULL" (or 'nullptr' since C++11). This is generally done at the time of variable declaration to check whether the pointer points to some valid memory address or not. It is also returned by several inbuilt functions as a failure response.

```
int* ptrName = nullptr // Creating a null pointer
```

Example

```
#include <iostream>
using namespace std;

int main()
{
    int* ptr = nullptr;

    if (ptr == nullptr) {
        cout << "Pointer is currently null." << endl;
    }
    else {
        cout << "Pointer is not null." << endl;
    }


    int value = 5;
    ptr = &value;


    if (ptr == nullptr) {
        cout << "Pointer is currently null." << endl;
    }
    else {
        cout << "Pointer is not null." << endl;
        cout << "Value at the memory location pointed to "
                "by the pointer: "
             << *ptr << endl;
    }
    return 0;
}
```

Advantages of using nullptr

- It is a good practice to initialize pointers to a null value as it helps avoid undefined behavior by explicitly indicating they are not pointing to valid memory locations.
- Null pointers act as default or initial values for pointers when no valid address is assigned to the pointers.
- They are useful in error conditions or to signify the absence of data that enables better handling of exceptional cases.
- To release the resources, like the destructor of a class, or to set pointers to NULL after deletion we can use a null pointer to avoid accidentally using or accessing the released memory.

3-auto_ptr

auto_ptr is a smart pointer that manages an object obtained via a new expression and deletes that object when auto_ptr itself is destroyed. Once the object is destroyed, it de-allocates the allocated memory. auto_ptr has ownership control over the object and it is based on the Exclusive Ownership Model, which says that a memory block can not be pointed by more than one pointer of the same type.

```
auto_ptr <type> pointer_name = value;
```

Example

```
void memLeak() {
  classA *ptr = new classA();
  // some code

  delete ptr;}
```

In this above code, we have used delete to deallocate the memory to avoid memory leaks. But what if an exception happens before reaching the delete statement? In this case, the memory will not be deallocated. Hence, there is a need for a pointer that can free the memory it is pointing to after the pointer itself gets destroyed.

```
void memLeakPrevented() {
  auto_ptr<classA> ptr(new classA());
  // some code
}
```

4-unique_ptr

std::unique_ptr is a smart pointer introduced in C++11. It automatically manages the dynamically allocated resources on the heap. Smart pointers are just wrappers around regular old pointers that help you prevent widespread bugs. Namely, forgetting to delete a pointer and causing a memory leak or accidentally deleting a pointer twice or in the wrong way. They can be used in a similar way to standard pointers. They automate some of the manual processes that cause common bugs.

```
unique_ptr<A> ptr1 (new A)

 /* unique_ptr<A>: It specifies the type of the std::unique_ptr.
In this case- an object of type A.

new A: An object of type A is dynamically allocated on the
heap using the new operator.

ptr1: This is the name of the std::unique_ptr variable. */
```

Example

```
#include <iostream>
#include <memory> //use the <memory> header file for using these smart pointers.

using namespace std;

struct A {
    void printA() { cout << "A struct...." << endl; }
};

int main()
{
    unique_ptr<A> p1(new A);
    p1->printA();

    cout << p1.get() << endl;
    return 0;
}
```

Example2

```
#include <iostream>
```

```
#include <memory>

using namespace std;

struct A {
    void printA() { cout << "A struct...." << endl; }
};
int main()
{
    unique_ptr<A> p1(new A);
    p1->printA();

    cout << p1.get() << endl;

    // will give compile time error
    unique_ptr<A> p2 = p1;
    p2->printA();
    return 0;
}
```

The above code will give compile time error as we cannot assign pointer p2 to p1 in case of unique pointers.

5-shared_ptr

std::shared_ptr is one of the smart pointers introduced in C++11. Unlike a simple pointer, it has an associated control block that keeps track of the reference count for the managed object. This reference count is shared among all the copies of the shared_ptr instances pointing to the same object, ensuring proper memory management and deletion.

```
std::shared_ptr <T> ptr_name;
```

Member Methods of shared_ptr

- use_count() :Returns the current reference count, indicating how many std::shared_ptr instances share ownership.
- unique() :Check if there is only one std::shared_ptr owning the object (reference count is 1).
- get() :Returns a raw pointer to the managed object. Be cautious when using this method.
- swap(shr_ptr2) :Swaps the contents (ownership) of two std::shared_ptr instances.

Example

```
#include <iostream>
#include <memory>
using namespace std;

class A {
public:
    void show() { cout << "A::show()" << endl; }
};

int main()
{

    shared_ptr<A> p1(new A);

    cout << p1.get() << endl;
    p1->show();

    shared_ptr<A> p2(p1);
    p2->show();

    cout << p1.get() << endl;
    cout << p2.get() << endl;

    cout << p1.use_count() << endl;
    cout << p2.use_count() << endl;

    p1.reset();
    cout << p1.get() << endl; // This will print nullptr or 0
    cout << p2.use_count() << endl;
    cout << p2.get() << endl;

    /*
    These lines demonstrate that p1 no longer manages an
    object (get() returns nullptr), but p2 still manages the
    same object, so its reference count is 1.
    */

    return 0;
}
```

6-weak_ptr

std::shared_ptr is one of the smart pointers introduced in C++11. Unlike a simple pointer, it has an associated control block that keeps track of the reference count for the managed object. This reference count is shared among all the copies of the shared_ptr instances pointing to the same object, ensuring proper memory management and deletion.

```
std::weak_ptr<type> name;
```

Member Methods of weak_ptr

- reset():Clear the weak_ptr.
- swap:It swaps the objects managed by weak_ptr.
- expired(): Check if the resource weak_ptr pointing to exists or not.
- lock():If the resource pointed by weak_ptr exists, this function returns a shared_ptr with ownership of that resource. If the resource does not exist, it returns default constructed shared_ptr.
- use_count():Tells about how many shared_ptr owns the resource.

Applications of weak_ptr

- Preventing Circular References: The primary application of weak_ptr is to prevent circular references. When an object wants to reference another object without owning it, it can use weak_ptr. This ensures that no circular references are created and objects can be safely freed when they are no longer needed.
- Cache Systems: weak_ptr is commonly used in cache implementations. Caches often need to temporarily store references to objects without preventing the deletion of those objects when they are no longer in use. weak_ptr provides an elegant solution for this use case.

Cyclic Dependency

A cyclic dependency occurs when two or more components (such as classes or modules) are mutually dependent on each other. In the context of software development, this often refers to situations where object A depends on object B, and object B, in turn, depends on object A, directly or indirectly. This can create a "cycle," where the system's dependencies form a loop.

This creates a circular dependency, and if not properly managed, can lead to issues such as:

-Memory Leaks: Objects in the cycle can be held in memory because they never get destroyed (because their reference counts never reach zero).
-Difficulty in Design: Such dependencies can complicate the system design and make it harder to modify or extend the system in the future.
-Tight Coupling: It often leads to tight coupling, where changes to one class can necessitate changes to the other, making code maintenance difficult.

In C++ (and other languages that support smart pointers), weak_ptr is used to avoid cyclic dependencies and memory management problems related to them. It works alongside shared_ptr, which is a smart pointer that automatically manages the lifetime of an object via reference counting.A shared_ptr keeps an object alive by maintaining a reference count, which means that the object will not be destroyed as long as there are active shared_ptr references to it. If two objects hold shared_ptr references to each other, they will never be destroyed, even if no other references to the objects exist. This leads to a memory leak. By using a weak_ptr, one of the objects can hold a non-owning reference to the other, preventing the cycle. weak_ptr does not increase the reference count, so it does not prevent the object from being destroyed.A weak_ptr can be converted to a shared_ptr using its lock() method. If the object still exists (i.e., if it has not been destroyed), lock() returns a shared_ptr to the object. If the object has already been deleted, lock() returns a nullptr. This ensures that you are always working with a valid object when accessing it.

0.1.4 Raw Pointers

A raw pointer in C++ refers to a basic pointer that holds the memory address of a variable or an object, without any additional management or safety mechanisms built into it. It's the simplest type of pointer in C++, and it points directly to a memory location. Raw pointers do not provide automatic memory management. This means that the programmer is responsible for allocating and deallocating memory manually (using new/delete or malloc/free). If memory is not properly deallocated, this can result in memory leaks. If a pointer is used after memory is freed, this can lead to dangling pointers or undefined behavior.

Advantages of Using Raw Pointers:

- Manual Memory Allocation and Deallocation: With raw pointers, you have complete control over when and how memory is allocated and deallocated. This can be particularly useful in scenarios where performance is critical, and you want to manage memory in a very specific manner.

- Minimal Overhead: Raw pointers typically incur less overhead than smart pointers. Smart pointers (such as std::unique_ptr or std::shared_ptr) introduce additional features (like reference counting, scope-based ownership, etc.), which can slightly reduce performance in certain cases.

- No Runtime Dependencies: Raw pointers are part of the core C++ language and do not require additional libraries or runtime dependencies. In contrast, smart pointers, like std::shared_ptr or std::unique_ptr, rely on the C++ Standard Library. For highly specialized or embedded systems with limited resources, using raw pointers can help keep the program lightweight.

- Cross-Language Interoperability: Raw pointers are fundamental in C and C++, so they are often used in scenarios where you're working with code that interfaces with C libraries or other low-level system code. Raw pointers can also be passed to functions written in other languages or legacy codebases, where smart pointers may not be compatible.

Problems with Raw Pointers:

- Memory Leaks: If you allocate memory using new or malloc but forget to free it using delete or free, the allocated memory will never be released, causing a memory leak.

  ```
  int* ptr = new int(10); // dynamically allocated memory
  // Forgetting to delete the memory will cause a memory leak.
  ```

- Dangling Pointers:

  A raw pointer that points to a memory location which has already been freed or deleted is called a dangling pointer. Using a dangling pointer leads to undefined behavior.

  ```
  int* ptr = new int(10);
  delete ptr;
  std::cout << *ptr; // Accessing deleted memory (dangling pointer)
  ```

0.1.5 Wild Pointers

A wild pointer refers to a pointer in programming that is pointing to a memory location that is either invalid, uninitialized, or has already been deallocated.This can lead to undefined behavior, such as program crashes, data corruption, or memory access violations.

```
int* p;  // Some unknown memory location is being corrupted.
*p = 12;
```

If a pointer points to a known variable then it's not a wild pointer.

```
int main()
{
        int* p; // wild pointer
        int a = 10;
        p = &a;  // p is not a wild pointer now

}
```

0.1.6 Data Inconsistency

Data inconsistency refers to a situation in which data across multiple locations or systems does not match or align. This can occur in databases, software applications, or any other system where data is stored, processed, or communicated.

-Mismatched Data:

Different copies of the same data may not match. For example, in a system with replicated databases, if the same customer's address is updated in one database but not in others, data inconsistency occurs.

-Different Versions of Data:
Inconsistent data can arise when different versions of the same information exist, such as one part of the system using an outdated value while another uses a newer one.

-Data Duplication:
When the same data is stored in multiple places, updating one instance but failing to update the others results in inconsistency. For example, a person's phone number being different in two systems.

-Lack of Synchronization:
In distributed systems, if data isn't properly synchronized across different locations, it can lead to inconsistency. For instance, if two users try to update the same record at the same time, one of the updates may be lost or overwritten.

-Concurrency Problems:
In multi-user systems, concurrent access and modifications to the same data without proper locking mechanisms can cause data inconsistency.

0.1.7 Buffer Overflow

A buffer overflow in C++ occurs when a program writes more data to a buffer (an allocated memory block) than it can hold. This leads to writing data beyond the allocated memory boundaries, causing adjacent memory to be overwritten. This can result in unpredictable behavior, including crashes, data corruption, or security vulnerabilities.

Consequences of Buffer Overflow

- Memory Corruption: The overflow can overwrite adjacent memory locations, potentially corrupting other variables, function return addresses, or control structures.
- Program Crashes: Buffer overflows can cause the program to crash by overwriting important data or control structures such as function pointers, which are critical for program execution.
- Security Vulnerabilities: Buffer overflows are often exploited in attacks to: Overwrite function return addresses, allowing attackers to take control of the program flow (e.g., redirect execution to malicious code). Execute arbitrary code by writing into buffer overflowed areas (this is called code injection).

0.1.8 Ownership Models

1- std::unique_ptr

Ownership Model:
Exclusive ownership: A unique_ptr is responsible for the lifetime of the object it points to, and it cannot be shared or copied. Only one unique_ptr can own a given resource at any point in time. When the unique_ptr goes out of scope, it automatically deletes

the associated object. Transferring ownership: You can transfer ownership of the object by moving the unique_ptr to another unique_ptr using std::move(). After the move, the original unique_ptr becomes null.

When to use:
Use unique_ptr when you want exclusive ownership of a resource and the resource should not be shared with other parts of the program. It's ideal when you need to guarantee that there is only one owner of a particular resource (e.g., a dynamically allocated object). It's often used for managing resources that have a single owner, such as when a resource needs to be cleaned up as soon as it goes out of scope.

2-std::shared_ptr

Ownership Model:
Shared ownership: Multiple shared_ptr objects can share ownership of the same resource. The resource is deleted when the last shared_ptr owning it goes out of scope (i.e., reference count drops to zero). It uses a reference count to track how many shared_ptr objects are managing the same resource.

When to use:
Use shared_ptr when you want shared ownership of a resource, meaning multiple parts of the program need access to the same object and it should not be deleted until all owners have released it. shared_ptr is useful in cases where you have complex ownership scenarios where resources might be shared across various parts of a program, like in a graph structure, or managing objects in multithreaded environments.

3-std::weak_ptr

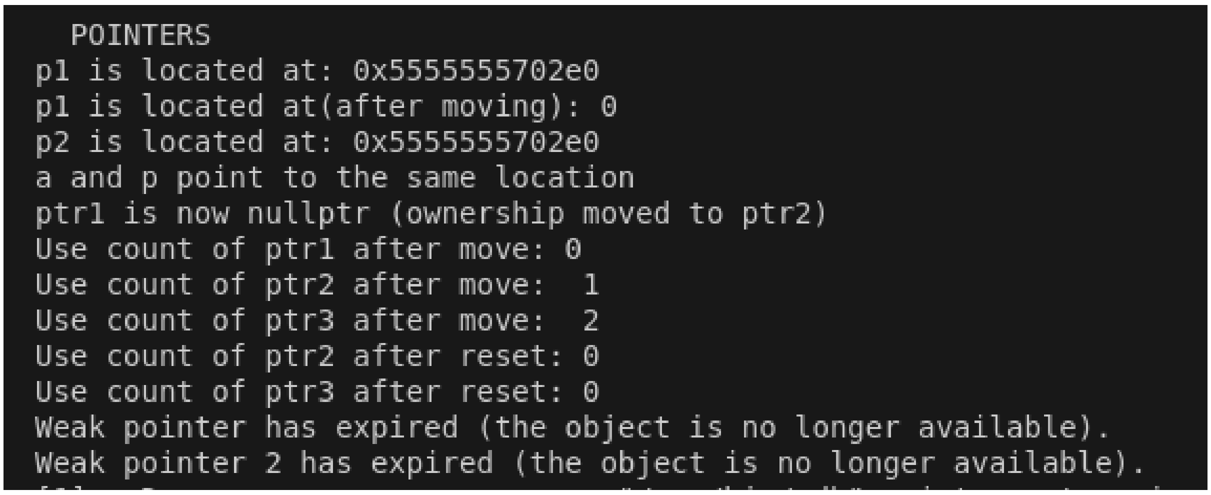
Ownership Model:
Non-owning reference: A weak_ptr does not own the resource it points to. It is used to observe an object that is managed by a shared_ptr without affecting its reference count. Preventing circular references: weak_ptr is used to break cycles in situations where two shared_ptr objects might reference each other, causing memory leaks.

When to use:
Use weak_ptr when you want to observe or reference an object managed by a shared_ptr without taking ownership. It's especially useful when you need to track objects but avoid strong references that would prevent their destruction. weak_ptr is also used in situations where you want to prevent circular references (common in object graphs or when objects reference each other).

0.1.9 Creating Smart Pointer

```
#include <iostream>
#include <memory>

using namespace std;
```

```
class ptrClass {
public:
    void showMessage() {
        cout << "  POINTERS  " << endl;
    }

};

class SmartPtr {
    ptrClass* ptr;

public:
    explicit SmartPtr(ptrClass* p = nullptr) : ptr(p) {}


    ~SmartPtr() { delete ptr; }


    ptrClass& operator*() {
        return *ptr;
    }


    ptrClass* operator->() {
        return ptr;
    }
};

int main() {
    SmartPtr smartPtr(new ptrClass());
    (*smartPtr).showMessage();


/*
//auto ptr

 auto_ptr<int> p (new int);
 *p.get() = 100;
 cout << "p points to " << *p.get() << '\n';
 auto_ptr<ptrClass> ptr1(new ptrClass());

 auto_ptr<ptrClass> ptr2 = move(ptr1);

 if (ptr1 == nullptr) {
 cout << "ptr1 is now nullptr (ownership moved to ptr2)" << endl;
    }
 ptr2->showMessage();


std::auto_ptr was deprecated in C++11 and removed in C++17 because it
had significant issues with move semantics, implicit ownership transfer,
and unclear ownership rules.
It was replaced by std::unique_ptr and std::shared_ptr, which provide clearer,
safer, and more predictable ownership semantics in modern C++ code.
If you're writing code in C++11 or later, it's best to use std::unique_ptr for
exclusive ownership and std::shared_ptr for shared ownership.
*/


//unique ptr

    unique_ptr<int> p1(new int);
    *p1.get() = 10;
    cout << p1.get() << endl;

    unique_ptr<int> p2 = move(p1);
    *p2.get();
    cout << p1.get() << endl;
    cout << p2.get() << endl;

//shared ptr

int* p= new int (10);
shared_ptr<int> a (p);

 if (a.get()==p){
    std::cout << "a and p point to the same location\n";

 }
 else{
    std::cout << "a and p do not point to the same location\n";
    }

/*three ways of accessing the same address:
  std::cout << *a.get() << "\n";
  std::cout << *a << "\n";
  std::cout << *p << "\n"; */

   shared_ptr<ptrClass> ptr1 = make_shared<ptrClass>();

    shared_ptr<ptrClass> ptr2 =move(ptr1);

    if (ptr1 == nullptr) {
        cout << "ptr1 is now nullptr (ownership moved to ptr2)" << endl;
    }

else{

}

  cout << "Use count of ptr1 after move: " << ptr1.use_count() << endl;

  cout << "Use count of ptr2 after move:  " << ptr2.use_count() << endl;

  shared_ptr<ptrClass> ptr3 = ptr2;

  cout << "Use count of ptr3 after move:  " << ptr3.use_count() << endl;


   ptr2.reset();

   cout << "Use count of ptr2 after reset: " << ptr2.use_count() << endl;

   ptr3.reset();

   cout << "Use count of ptr3 after reset: " << ptr3.use_count() << endl;


//weak ptr

 weak_ptr<ptrClass> weakPtr = ptr3;

    shared_ptr<ptrClass> lockedPtr = weakPtr.lock();

    if (lockedPtr) {
        cout << "Weak pointer has successfully locked the object."
        << endl;
        lockedPtr->showMessage();
    } else {
        cout << "Weak pointer has expired (the object is no longer available)."
        << endl;
    }

    weak_ptr<ptrClass> weakPtr2 = move(weakPtr);

    shared_ptr<ptrClass> lockedPtr2 = weakPtr2.lock();

    if (lockedPtr2) {
        cout << "Weak pointer 2 has successfully locked the object after move."
        << endl;
        lockedPtr2->showMessage();
    } else {
        cout << "Weak pointer 2 has expired (the object is no longer available)."
        << endl;
    }


}
```



```
  POINTERS
p1 is located at: 0x5555555702e0
p1 is located at(after moving): 0
p2 is located at: 0x5555555702e0
a and p point to the same location
ptr1 is now nullptr (ownership moved to ptr2)
Use count of ptr1 after move: 0
Use count of ptr2 after move:  1
Use count of ptr3 after move:  2
Use count of ptr2 after reset: 0
Use count of ptr3 after reset: 0
Weak pointer has expired (the object is no longer available).
Weak pointer 2 has expired (the object is no longer available).
```