**POINTERS ::- >**

<https://www.geeksforgeeks.org/c-language-2-gq/pointers-gq/>

<https://www.geeksforgeeks.org/c-language-2-gq/advanced-pointer-c-gq/>

<https://www.geeksforgeeks.org/what-are-near-far-and-huge-pointers/>

// C++ program to illustrate Pointer Arithmetic

#include <bits/stdc++.h>

using namespace std;

void geeks()

{

// Declare an array

int v[3] = { 10, 100, 200 };

// declare pointer variable

int\* ptr;

// Assign the address of v[0] to ptr

ptr = v;

for (int i = 0; i < 3; i++) {

cout << "Value at ptr = " << ptr << "\n";

cout << "Value at \*ptr = " << \*ptr << "\n";

// Increment pointer ptr by 1

ptr++;

}

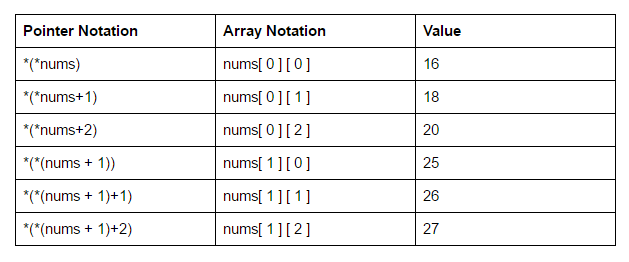
}

// Driver program

int main() { geeks(); }

int nums[2][3] = { { 16, 18, 20 }, { 25, 26, 27 } };

**In general, nums[ i ][ j ] is equivalent to \*(\*(nums+i)+j)**



## **Invalid pointers**

A pointer should point to a valid address but not necessarily to valid elements (like for arrays). These are called invalid pointers. Uninitialized pointers are also invalid pointers.

int \*ptr1;

int arr[10];

int \*ptr2 = arr+20;

Here, ptr1 is uninitialized so it becomes an invalid pointer and ptr2 is out of bounds of arr so it also becomes an invalid pointer. (Note: invalid pointers do not necessarily raise compile errors)

## **NULL Pointers**

A [null pointer](https://www.geeksforgeeks.org/few-bytes-on-null-pointer-in-c/) is a pointer that point nowhere and not just an invalid address. Following are 2 methods to assign a pointer as NULL;

int \*ptr1 = 0;

int \*ptr2 = NULL;

An opaque pointer in C++ refers to a programming technique where the implementation details of a data structure are hidden from the users of that data structure. The term "opaque" implies that the users cannot see or directly manipulate the internal details of the data structure; they can only interact with it through a pointer.

**Declaration in the Header File:**

* In the header file, only the declaration of the structure (or class) is provided, without revealing the details of its members.

// Header file (e.g., MyLibrary.h)

// Declaration of the structure with an opaque pointer

struct MyStruct;

// Function declarations that operate on MyStruct

MyStruct\* createMyStruct();

void doSomethingWithMyStruct(MyStruct\* ptr);

void destroyMyStruct(MyStruct\* ptr);

**Definition in the Source File:**

* In the source file, the actual definition of the structure is provided, but it is not exposed to the users of the library. Instead, users work with a pointer to the structure.

// Source file (e.g., MyLibrary.cpp)

// Definition of the structure

struct MyStruct {

int data;

// Other members...

};

// Function implementations

MyStruct\* createMyStruct() {

return new MyStruct();

}

void doSomethingWithMyStruct(MyStruct\* ptr) {

// Implementation details...

}

void destroyMyStruct(MyStruct\* ptr) {

delete ptr;

}

**Usage by Clients:**

* Clients of the library include the header file and interact with the data structure only through the opaque pointer, without knowing the internal details.

// Client code

#include "MyLibrary.h"

int main() {

MyStruct\* myObject = createMyStruct();

doSomethingWithMyStruct(myObject);

destroyMyStruct(myObject);

return 0;

}

In Summary opaque pointers are a technique that can be used to hide the implementation details of an object and provide a level of abstraction in C++. They are useful for hiding the implementation details of an object from the client code, and also for providing a level of abstraction. However, it’s important to use them judiciously and to consider other options as well.

**how to pass a function as an argument to another function exapmple in c++**

#include <iostream>

// Function that takes a function pointer as an argument

void PerformOperation(int x, void (\*operation)(int)) {

std::cout << "Performing operation on " << x << ": ";

operation(x);

}

// Functions that can be passed as arguments

void Square(int x) {

std::cout << x << " squared is " << x \* x << std::endl;

}

void Cube(int x) {

std::cout << x << " cubed is " << x \* x \* x << std::endl;

}

int main() {

// Passing Square function as an argument

PerformOperation(5, Square);

// Passing Cube function as an argument

PerformOperation(3, Cube);

return 0;

}

-------------------------------------------------------------------

#include <iostream>

#include <functional>

// Function that takes a std::function as an argument

void PerformOperation(int x, std::function<void(int)> operation) {

std::cout << "Performing operation on " << x << ": ";

operation(x);

}

// Functions that can be passed as arguments

void Square(int x) {

std::cout << x << " squared is " << x \* x << std::endl;

}

void Cube(int x) {

std::cout << x << " cubed is " << x \* x \* x << std::endl;

}

int main() {

// Passing Square function as an argument

PerformOperation(5, Square);

// Passing Cube function as an argument

PerformOperation(3, Cube);

return 0;

}

**std::function** is a part of the C++ Standard Template Library (STL) and is defined in the **<functional>** header. It is a general-purpose polymorphic function wrapper that can store, copy, and invoke any callable object (function, lambda expression, functor, etc.) with a compatible signature.

### Why std::function is Used:

1. **Polymorphism:**
   * **std::function** provides a way to achieve polymorphism with functions. You can create a **std::function** object and assign various callable entities to it, allowing you to write more flexible and generic code.
2. **Callback Mechanism:**
   * It is commonly used in scenarios where you need to pass functions or function-like objects as arguments to functions. This is particularly useful in implementing callback mechanisms or strategies.
3. **Function Pointers Replacement:**
   * **std::function** can replace traditional function pointers and is more versatile. It can store different types of callable entities and provides type safety.
4. **Container Usage:**
   * **std::function** can be stored in standard containers, allowing you to create collections of callable objects with different types.

// Syntax for declaring a std::function

std::function<returnType(parameterTypes)> functionName;

### Advantages:

1. **Flexibility:**
   * **std::function** provides a flexible way to work with functions, allowing you to use different types of callable entities interchangeably.
2. **Type Safety:**
   * It provides type safety by allowing you to specify the signature of the callable entity at the time of declaration.

In C++, a callable object refers to an object that can be invoked as if it were a function. Callable objects include functions, function pointers, function objects (also known as functors), and lambda expressions. These objects can be used in contexts where a function is expected, such as when passed as arguments to functions, stored in containers, or assigned to function pointers.

Function Pointer in C++

As we know that pointers are used to point some variables; similarly, the function pointer is a pointer used to point functions. It is basically used to store the address of a function. We can call the function by using the function pointer, or we can also pass the pointer to another function as a parameter.

They are mainly useful for event-driven applications, callbacks, and even for storing the functions in arrays

### **Syntax for Declaration**

The following is the syntax for the declaration of a function pointer:

**int** (\*FuncPtr) (**int**,**int**);

**POINTER TO A FUNCTION ----------------------------**

"function pointer" and "pointer to a function" refer to the same thing: a pointer that stores the memory address of a function. The former term usually refers to a variable that holds the address of a function, while the latter term usually refers to the type of the function pointer.

In C++, `const` can be used with pointers to create two different types: a "const pointer" and a "pointer to a const". Here's what each of these means:

1. Const pointer:

A const pointer is a pointer whose value (i.e. the memory address it points to) cannot be changed. However, the object that it points to can be changed. In other words, a const pointer is a pointer that is itself constant, but the object it points to is not necessarily constant.

Here's an example of a const pointer:

```cpp

int x = 5;

int y = 10;

int\* const ptr = &x;

\*ptr = 7; // OK, changes x to 7

ptr = &y; // ERROR, ptr is a const pointer and cannot be changed

```

2. Pointer to a const:

A pointer to a const is a pointer that can change its value (i.e. the memory address it points to), but the object it points to cannot be changed. In other words, a pointer to a const is a pointer that can be changed, but the object it points to is constant.

Here's an example of a pointer to a const:

```cpp

int x = 5;

const int\* ptr = &x;

\*ptr = 7; // ERROR, ptr is a pointer to a const and cannot change the object it points to

ptr = &y; // OK, ptr can be changed to point to a different object

```

Note that the `const` keyword can be placed either before or after the `\*` in a pointer declaration, and this affects whether the pointer itself is const or the object it points to is const. If the `const` keyword is placed before the `\*`, then the pointer is const (i.e. it cannot be changed). If the `const` keyword is placed after the `\*`, then the object being pointed to is const.

IMP points about ARRAY::

<https://www.geeksforgeeks.org/pointer-array-array-pointer/>

#include <iostream>

using namespace std;

int main()

{

// Pointer to an array of five numbers

int(\*a)[5];

int b[5] = { 1, 27, 3, 4, 5 };

int i = 0;

// Points to the whole array b

a = &b;

for (i = 0; i < 5; i++)

cout << \*(\*a+i) << endl;

return 0;

}

Void pointer is a specific pointer type – void \* – a pointer that points to some data location in storage, which doesn’t have any specific type. Void refers to the type. Basically the type of data that it points to is can be any.

**Important Points**

1. void pointers **cannot be dereferenced**. It can however be done using typecasting the void pointer
2. Pointer arithmetic is not possible on pointers of void due to lack of concrete value and thus size.

**NULL vs Uninitialized pointer –**An uninitialized pointer stores an undefined value. A null pointer stores a defined value, but one that is defined by the environment to not be a valid address for any member or object.

**NULL vs Void Pointer** – Null pointer is a value, while void pointer is a type

A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer. There are **three** different ways where Pointer acts as dangling pointer

1. **De-allocation of memory**
2. **Variable goes out of scope**
3. [Function Call](https://www.geeksforgeeks.org/what-happens-when-we-call-a-function/)

**SMART POINTERS :: -🡪>**

A *Smart Pointer* is a wrapper class over a pointer with an operator like **\*** and

**->** overloaded. The objects of the smart pointer class look like normal pointers. But, unlike *Normal Pointers* it can deallocate and free destroyed object memory.

The idea is to take a class with a pointer, [destructor,](https://www.geeksforgeeks.org/destructors-c/)and [overloaded operators](https://www.geeksforgeeks.org/operator-overloading-c/) like **\*** and **->**. Since the destructor is automatically called when an object goes out of scope, the dynamically allocated memory would automatically be deleted

// C++ program to demonstrate the working of Smart Pointer

#include <iostream>

using namespace std;

class SmartPtr {

int\* ptr; // Actual pointer

public:

// Constructor: Refer https:// www.geeksforgeeks.org/g-fact-93/

// for use of explicit keyword

explicit SmartPtr(int\* p = NULL) { ptr = p; }

// Destructor

~SmartPtr() { delete (ptr); }

// Overloading dereferencing operator

int& operator\*() { return \*ptr; }

};

int main()

{

SmartPtr ptr(new int(20));

cout << \*ptr;

// We don't need to call delete ptr: when the object

// ptr goes out of scope, the destructor for it is automatically

// called and destructor does delete ptr.

return 0;

}

Int \*a = new int(34);

Smartptr sm = a;

Cout<<\*sm;

// this can work without explicit but some issues:

While this might seem convenient, it can introduce subtle issues. For example:

1. **Ownership and Memory Management:**
   * The **SmartPtr** class is responsible for managing the memory allocated to the **int**. If implicit conversions are allowed, users might forget that the **SmartPtr** is involved in memory management, leading to potential memory leaks or double deletions.
2. **Unexpected Conversions:**
   * Implicit conversions can sometimes lead to unexpected behavior, making the code less readable and more error-prone.
3. **Compiler Warnings:**
   * Without **explicit**, the compiler might not issue warnings for certain types of implicit conversions, making it harder to catch potential issues during development.

By keeping the **explicit** keyword in this context, you explicitly require users to use the constructor in a more intentional manner, making the code safer and more self-documenting. If the **explicit** keyword is removed, it's essential for developers using the class to be aware of the potential consequences and manage memory correctly.

// C++ program to demonstrate the working of Template and

// overcome the issues which we are having with pointers

#include <iostream>

using namespace std;

// A generic smart pointer class

template <class T> class SmartPtr {

T\* ptr; // Actual pointer

public:

// Constructor

explicit SmartPtr(T\* p = NULL) { ptr = p; }

// Destructor

~SmartPtr() { delete (ptr); }

// Overloading dereferencing operator

T& operator\*() { return \*ptr; }

// Overloading arrow operator so that

// members of T can be accessed

// like a pointer (useful if T represents

// a class or struct or union type)

T\* operator->() { return ptr; }

};

int main()

{

SmartPtr<int> ptr(new int());

\*ptr = 20;

cout << \*ptr;

return 0;

}

The **explicit** keyword in C++ is used to prevent implicit type conversions. It can be applied to constructors that take a single argument. When a constructor is marked as **explicit**, it disallows the compiler from automatically converting the argument type to the class type. This is particularly useful in preventing unintended type conversions and improving code safety.

A screenshot of a computer

Description automatically generated

## Problems with Normal Pointers

Some Issues with normal pointers in C++ are as follows:

* **Memory Leaks:**This occurs when memory is repeatedly allocated by a program but never freed. This leads to excessive memory consumption and eventually leads to a system crash.
* **Dangling Pointers:** A [dangling pointer](https://www.geeksforgeeks.org/difference-between-dangling-pointer-and-void-pointer) is a pointer that occurs at the time when the object is de-allocated from memory without modifying the value of the pointer.
* **Wild Pointers:**Wild pointers are pointers that are declared and allocated memory but the pointer is never initialized to point to any valid object or address.
* **Data Inconsistency:**Data inconsistency occurs when some data is stored in memory but is not updated in a consistent manner.
* **Buffer Overflow:**When a pointer is used to write data to a memory address that is outside of the allocated memory block. This leads to the corruption of data which can be exploited by malicious attackers.

## auto\_ptr

Using auto\_ptr, you can manage objects obtained from new expressions and delete them when auto\_ptr itself is destroyed.

std::auto\_ptr<int> myval = new int(34); // invalid

std::auto\_ptr<int> myval (new int(23)) ; // valid

but In normal pointers:

int \* ar = new int (23) // valid

int \*ar (new int(33)) //valid

#include <iostream>

#include <memory>

int main() {

// Create an auto\_ptr to manage a dynamically-allocated integer

std::auto\_ptr<int> myInt(new int(42));

// Print out the value of the integer

std::cout << \*myInt << '\n';

// Reset the auto\_ptr to manage a different dynamically-allocated integer

myInt.reset(new int(99));

// Print out the value of the new integer

std::cout << \*myInt << '\n';

// When the auto\_ptr goes out of scope, it deletes the dynamically-allocated integer

return 0;

}

Note that **auto\_ptr** has some limitations and potential issues, such as its lack of support for custom deleters, and the possibility of accidental copying or assignment. Therefore, it is recommended to use **unique\_ptr** or **shared\_ptr** instead, depending on the situation.

## **unique\_ptr**

*unique\_ptr* stores one pointer only. We can assign a different object by removing the current object from the pointer.

0. unique\_ptr is a class template.

1. unique\_ptr is one of the smart pointer provided by c++11 to prevent memory leaks.

2. unique\_ptr wraps a raw pointer in it, and de-allocates the raw pointer, when unique\_ptr object goes out of scope.

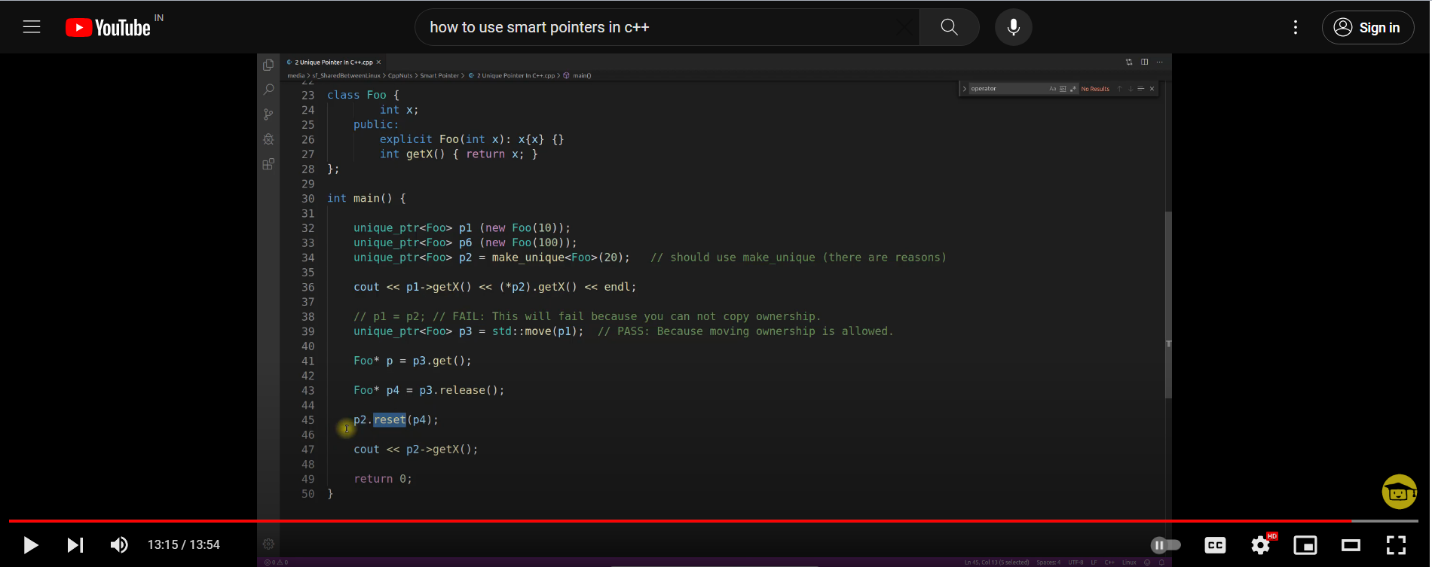
3. similar to actual pointers we can use arrow and \* on the object of unique\_ptr, because it is overloaded in unique\_ptr class.

4. When exception comes then also it will de-allocate the memory hence no memory leak.

5. Not only object we can create array of objects of unique\_ptr.

Operations : release, reset, swap, get, get\_deleter

<https://www.geeksforgeeks.org/smart-pointers-cpp/>



example rectangle::

## **shared\_ptr**

By using *shared\_ptr* more than one pointer can point to this one object at a time and it’ll maintain a **Reference Counter** using the ***use\_count()* method.**

There are few points about it:

0. shared\_ptr is a smart pointer which can share the ownership of object (managed object).

1. Several shared\_ptr can point to the same object (managed object).

2. It keep a reference count to maintain how many shared\_ptr are pointing to the same object. and once last shared\_ptr goes out of scope then the managed object gets deleted.

3. shared\_ptr is threads safe and not thread safe. [what is this??] a. control block is thread safe b. managed object is not

4. There are three ways shared\_ptr will destroyed managed object.

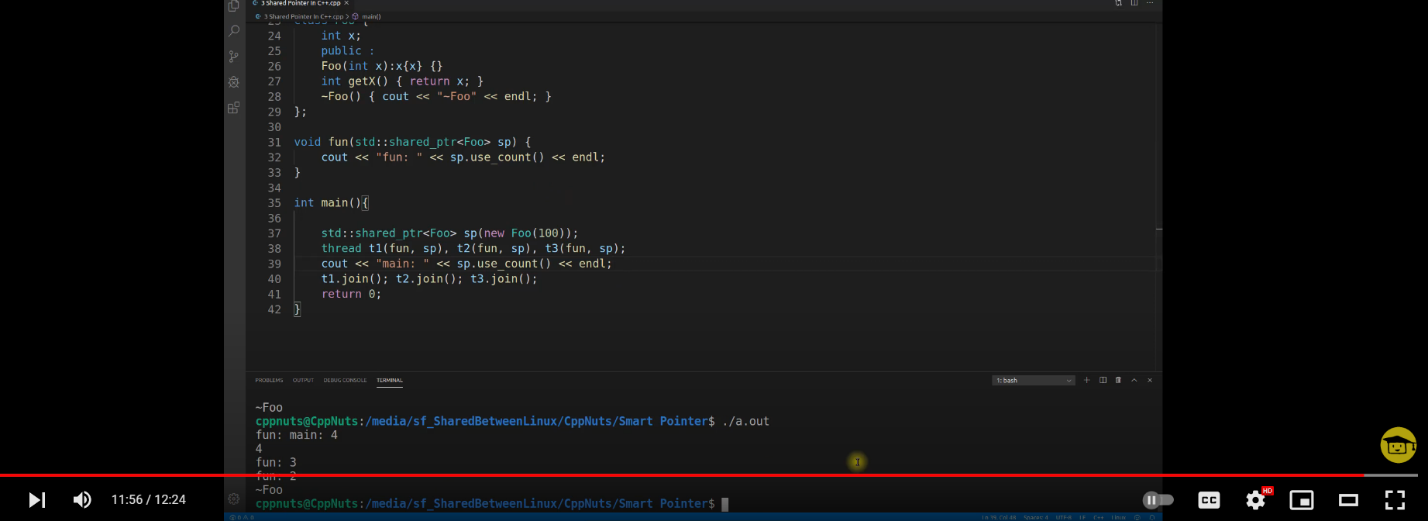
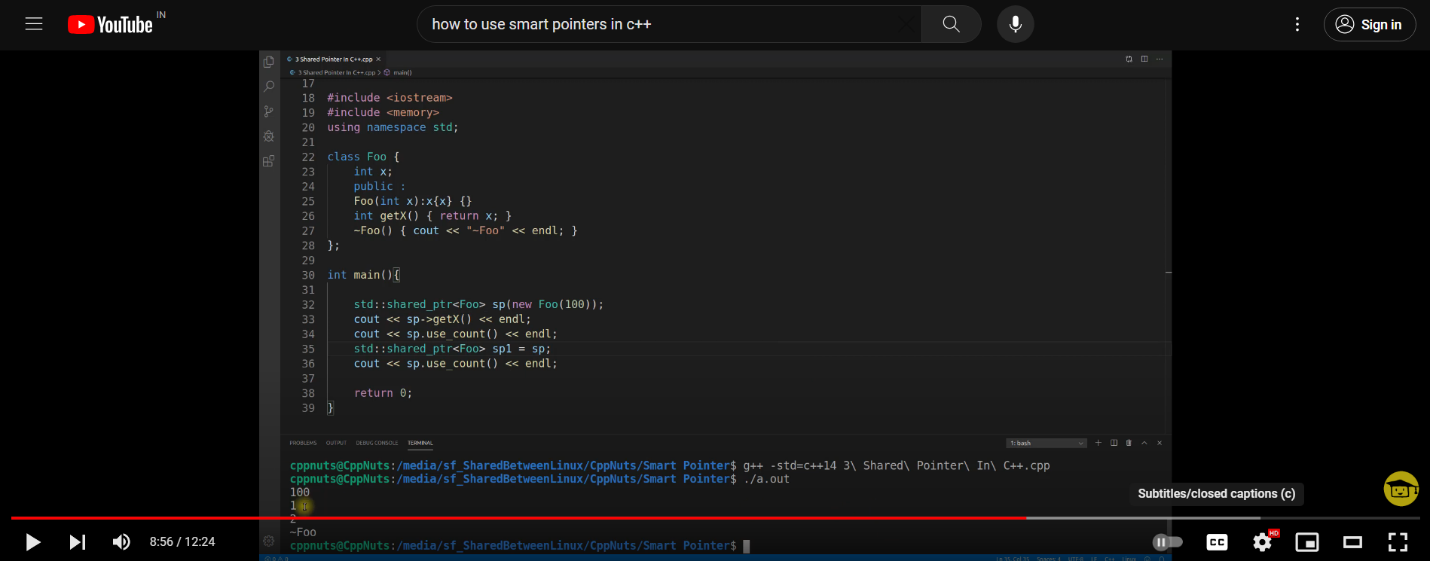
a. If the last shared\_ptr goes out of scope.

b. If you initialise shared\_ptr with some other shared\_ptr.

c. If you reset shared\_ptr.

5. Reference count doesn't work when we use reference or pointer of shared\_ptr.

I.e. means shared ptr should always be passed by value



Applications of pointers

<https://www.geeksforgeeks.org/smart-pointers-cpp/>

## **weak\_ptr**

Weak\_ptr is a smart pointer that holds a non-owning reference to an object. It’s much more similar to shared\_ptr except it’ll not maintain a **Reference Counter**. In this case, a pointer will not have a stronghold on the object. The reason is if suppose pointers are holding the object and requesting for other objects then they may form a **Deadlock.**

0. If we say unique\_ptr is for unique ownership and shared\_ptr is for shared ownership then weak\_ptr is for non-ownership smart pointer.

1. It actually reference to an object which is managed by shared\_ptr.

2. A weak\_ptr is created as a copy of shared\_ptr.

3. We have to convert weak\_ptr to shared\_ptr in order to use the managed object.

4. It is used to remove cyclic dependency between shared\_ptr.

A screen shot of a computer

Description automatically generated

<https://www.geeksforgeeks.org/applications-of-pointers-in-c-cpp/>

OBJECT SLICING IN C++

<https://www.geeksforgeeks.org/object-slicing-in-c/>

# Difference between passing pointer to pointer and address of pointer to any function

<https://www.geeksforgeeks.org/difference-between-passing-pointer-to-pointer-and-address-of-pointer-to-any-function/>

In C, data type of character constants is int, but in C++, data type of same is char.

**Passing pointer to a function**

If a pointer is passed to a function as a parameter and tried to be modified then the changes made to the pointer does not reflects back outside that function. This is because only a copy of the pointer is passed to the function. It can be said that “pass by pointer” is [passing a pointer by value](https://www.geeksforgeeks.org/passing-by-pointer-vs-passing-by-reference-in-c/).

**Passing “pointer to a pointer” as a parameter to function**

The above problem can be resolved by passing the address of the pointer to the function instead of a copy of the actual function. For this, the function parameter should accept a “pointer to pointer” as shown in the below program:

<https://www.geeksforgeeks.org/passing-reference-to-a-pointer-in-c/>

# Difference between Type Casting and Type Conversion

<https://www.geeksforgeeks.org/difference-between-type-casting-and-type-conversion/>

# Difference between data type and data structure

<https://www.geeksforgeeks.org/difference-between-data-type-and-data-structure/>

References

<https://www.geeksforgeeks.org/references-in-cpp/>

Another way to iterate through the array:

#include <iostream>

using namespace std;

int main()

{

int b[5] = { 1, 27, 3, 4, 5 };

for (int& x : b) {

x = x + 5;

}

for (int x : b) {

cout << x << " ";

}

return 0;

}

int fun(int& x) { return x; }

int main()

{

//int a=10; // this is correct

cout << fun(10);

return 0;}

The error in the program is that the function **fun** takes an integer reference parameter **x**, but it is called with a constant integer argument **10**. A reference parameter expects a variable as an argument, not a constant value.

To fix this error, you can either declare a variable and pass its reference to the **fun** function or modify the **fun** function to take a constant reference parameter instead of a non-constant reference parameter

Or

int fun(const int& x) { return x; }

# Reference to a pointer in C++ with examples and applications

<https://www.geeksforgeeks.org/reference-to-a-pointer-in-c-with-examples-and-applications/>

Reference to a pointer to a pointer

#include <iostream>

**using** **namespace** std;

**int** main()

{

    // Variable

**int** i = 10;

    // Pointer to i

**int**\* ptr\_i = &i;

    // Reference to a Pointer to a ptr\_i

**int**\*& ptr\_ref = ptr\_i;

    cout << \*ptr\_ref;

**return** 0;

}

# Different ways to use Const with Reference to a Pointer in C++

<https://www.geeksforgeeks.org/different-ways-to-use-const-with-reference-to-a-pointer-in-c/>

[References to pointers](https://www.geeksforgeeks.org/reference-to-a-pointer-in-c-with-examples-and-applications/) is a modifiable value that’s used same as a normal pointer.( it can also be called as alias of a pointer variable)

datatype \*&var\_name;

**int** i = 10;

    // Pointer to i

**int**\* ptr\_i = &i;

    // Reference to a Pointer ptr\_i

**int**\*& ptr\_ref = ptr\_i;

    cout << \*ptr\_ref;

It just act as a normal pointer

**Reference to a Const Pointer** is a reference to a constant pointer.

datatype const \*&var\_name;

  // Reference to a Const Pointer

**int** **const**\*& ptr\_ref = ptr;

**Const Reference to a pointer** is a non-modifiable value that’s used same as a const pointer.

datatype\* const &var\_name;

// Const Reference to a Pointer

**int**\* **const**& ptr\_ref = ptr;

LAVLUE AND RVALUE

In C++, an lvalue (left value) refers to an expression that can appear on the left-hand side of an assignment statement and has an identifiable memory address. On the other hand, an rvalue (right value) refers to an expression that can only appear on the right-hand side of an assignment statement and does not have an identifiable memory address.

Examples of lvalues include variables, arrays, and dereferenced pointers, while examples of rvalues include constants, literals, and the results of expressions.

LValue and Rvalue::

<https://www.geeksforgeeks.org/lvalue-and-rvalue-in-c-language/>

# In C++, lvalue references (&) and rvalue references (&&) are used to express different kinds of references to objects.

### Lvalue References (&):

An lvalue is an expression that refers to an object that persists beyond a single expression. Lvalue references (**&**) are used to bind to lvalues. They provide a way to refer to an existing object and can be used as function parameters or return types.

# #include <iostream>

# void modifyValue(int& x) {

# x \*= 2;

# }

# int main() {

# int value = 5;

# 

# // lvalue reference

# int& ref = value;

# std::cout << "Original value: " << value << std::endl;

# modifyValue(ref);

# std::cout << "Modified value: " << value << std::endl;

# return 0;

# }

### Rvalue References (&&):

An rvalue is an expression that is either a temporary object (e.g., a literal or the result of an expression) or an object that the program is allowed to treat as temporary. Rvalue references (**&&**) are used to bind to rvalues. They are commonly used in move semantics and perfect forwarding

# #include <iostream>

# #include <cstring>

# using namespace std;

# class MyString {

# private:

# char\* data;

# public:

# // Constructor

# MyString(const char\* str) {

# std::cout << "Constructor" << std::endl;

# // Allocate memory and copy the string

# data = new char[strlen(str) + 1];

# strcpy(data, str);

# }

# // Move constructor

# MyString(MyString&& other) noexcept {

# std::cout << "Move constructor" << std::endl;

# data = other.data;

# other.data = nullptr; // Make sure to leave other in a valid state

# }

# // Destructor

# ~MyString() {

# delete[] data;

# }

# void print() const {

# if (data != nullptr) {

# cout << data << endl;

# } else {

# cout << "Empty string" << endl;

# }

# }

# };

# int main() {

# MyString str1("Hello");

# MyString str2 = std::move(str1);

# str1.print(); // This should print an empty string

# str2.print(); // This should print "Hello"

# return 0;

# }

# Void fn(int& ref)

# {

# Cout<<”normal”<<endl;

# }

# Void fn(int && ref)

# {

# Cout<<”rval callaed”<<endl;

# }

# Main()

# {

# Int val = 10;

# Fn(val) - >normal;

# Fn(100) - >rval called;

# }

# What is move const and why it is used:

Move constructors are special member functions in C++ introduced to efficiently transfer the ownership of resources (such as dynamic memory allocations) from one object to another. They are a key part of move semantics, a feature introduced in C++11.

In a move operation, the contents of an object are "moved" to another object, leaving the source object in a valid but unspecified state. This is in contrast to a copy operation, where a duplicate of the contents is created, leaving both the source and destination objects independent of each other.

### Purpose of Move Constructors:

1. **Efficiency:**
   * Move constructors provide a more efficient way to transfer ownership of resources compared to deep copying in copy constructors.
   * Instead of duplicating the resources, move constructors can efficiently "steal" the resources from the source object.
2. **Reducing Redundant Allocations:**
   * Move semantics is particularly beneficial in scenarios where temporary objects are created during expressions, such as when returning objects from functions or during temporary object creation in expressions.
3. **Avoiding Deep Copying:**
   * When dealing with large data structures, avoiding unnecessary copying of data can significantly improve performance.
   * Move semantics allows for efficient transfer of resources without duplicating the data.
4. **Enabling Rvalue References:**
   * Move constructors are closely tied to rvalue references (**&&**), which represent temporary or expiring objects.
   * Rvalue references allow you to distinguish between objects that are candidates for efficient moves and those that should be copied.

# lvalues references and rvalues references in C++ with Examples

<https://www.geeksforgeeks.org/lvalues-references-and-rvalues-references-in-c-with-examples/>

**int** a = 10;

    // Declaring lvalue reference

    // (i.e variable a)

**int**& lref = a;

    // Declaring rvalue reference

**int**&& rref = 20;

# Function overloading and const keyword

<https://www.geeksforgeeks.org/function-overloading-and-const-functions/>

fn definition with const keyword extra

# Difference between const int\*, const int \* const, and int const \*

<https://www.geeksforgeeks.org/difference-between-const-int-const-int-const-and-int-const/>

**int const\*** is equivalent to **const int\***

# Passing Reference to a Pointer in C++

 It is allowed to use “pointer to pointer” in both C and C++, but we can use “Reference to pointer” only in C++.

# Undefined Behavior in C and C++

<https://www.geeksforgeeks.org/undefined-behavior-c-cpp/>

# Name Mangling and extern “C” in C++

<https://www.geeksforgeeks.org/extern-c-in-c/>

**#define**: This keyword is used to define a macro. Macros are preprocessor directives that allow you to define a shorthand for a sequence of code. Whenever a macro name is encountered by the compiler, it replaces the name with the definition of the macro.

INLINE FUNCTIONS ::

<https://www.geeksforgeeks.org/inline-functions-cpp/>

VOID

<https://www.geeksforgeeks.org/return-from-void-functions-in-cpp/>

How to create a pointer object of a structure:

#include <iostream>

struct Point {

int x;

int y;

};

int main() {

Point p = {2, 3}; // create a Point object

Point\* ptr = &p; // create a pointer object to p

//Imp imp above lines , whenever u want to create a pointer object of a structure always create a normal onject and then assign this object to pointer object.

std::cout << "x: " << p.x << ", y: " << p.y << std::endl; // output: x: 2, y: 3

std::cout << "x: " << ptr->x << ", y: " << ptr->y << std::endl; // output: x: 2, y: 3

ptr->x = 4; // change the value of x through the pointer

std::cout << "x: " << p.x << ", y: " << p.y << std::endl; // output: x: 4, y: 3

std::cout << "x: " << ptr->x << ", y: " << ptr->y << std::endl; // output: x: 4, y: 3

return 0;

}