Pointers and References in C++

- In C++ pointers and references both are mechanisms used to deal with memory, memory address, and data in a program.
- Pointers are used to store the memory address of another variable whereas references are used to create an alias for an already existing variable.
- Pointers are symbolic representation of addresses. Pointers store the address of variables or a memory location.
- They enable programs to simulate call-by-reference and create and manipulate dynamic data structures.

Datatype *var_name;

For eg. int *ptr // ptr points to an address that holds int data

- Note that the * sign can be confusing here, as it does two different things in our code:
- When used in declaration (string* ptr), it creates a pointer variable.
- When not used in declaration, it act as a dereference operator.
- A pointer however, is a variable that stores the memory address as its value. It is basically an integer, a number which stores a memory address.
- It also gives value stored at that address
- There are 3 ways to create pointer variables

```
string* mystring; // Preferred
string *mystring;
string * mystring;
```

- References and Pointers are important in C++, because they give you the ability to manipulate the data in the computer's memory - which can reduce the code and improve the performance.
- These two features are one of the things that make C++ stand out from other programming languages

How to use a pointer?

- Define a pointer variable
- Assigning the address of a variable to a pointer using the unary operator (&) which returns the address of that variable.
- Accessing the value stored in the address using unary operator (*) which returns the value of the variable located at the address specified by its operand.
- The reason we associate data type with a pointer is that it knows how many bytes the data is stored in. When we increment a pointer, we increase the pointer by the size of the data type to which it points.

Pointer Expressions and Pointer Arithmetic

- A limited set of arithmetic_operations can be performed on pointers which are:

- incremented (++)
- decremented (—)
- an integer may be added to a pointer (+ or +=)
- an integer may be subtracted from a pointer (– or -=)
- difference between two pointers (p1-p2)

(Note: Pointer arithmetic is meaningless unless performed on an array.)

Advanced Pointer Notation

- Consider pointer notation for the two-dimensional numeric arrays. consider the following declaration

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

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

Pointer Notation	Array Notation	Value
*(*nums)	nums[0] [0]	16
*(*nums+1)	nums[0] [1]	18
*(*nums+2)	nums[0] [2]	20
((nums + 1))	nums[1] [0]	25
((nums + 1)+1)	nums[1] [1]	26
((nums + 1)+2)	nums[1] [2]	27

Pointers to pointers

- In C++, we can create a pointer to a pointer that in turn may point to data or another pointer. The syntax simply requires the unary operator (*) for each level of indirection while declaring the pointer.

  char a;

  char *b;

  char ** c;

  a = ‘g’;

  b = &a;

  c = &b;

Here b points to a char that stores ‘g’ and c points to the pointer b.

Void Pointers

- This is a special type of pointer available in C++ which represents the absence of type.
- Void_pointers_are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereferencing properties).
- This means that void pointers have great flexibility as they can point to any data type.
- There is a payoff for this flexibility. These pointers cannot be directly dereferenced.
- They have to be first transformed into some other pointer type that points to a concrete data type before being dereferenced.
- 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 *ptr = 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 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;
```

Application of Pointers in C++

- To pass arguments by reference: Passing by reference serves two purposes
- For accessing array elements: The Compiler internally uses pointers to access array elements.
- To return multiple values: For example in returning square and the square root of numbers.
- Dynamic memory allocation: We can use pointers to dynamically allocate memory. The advantage of dynamically allocated memory is, that it is not deleted until we explicitly delete it.
- To implement data structures.
- To do system-level programming where memory addresses are useful.

```
void swap(int *x, int *y)

{

   int z = *x;
```

```
    *x = *y;
    *y = z;
}
Main(){swap(&a, &b);….}
```

Passing by Pointer	Passing By Reference

void swap(int *x, int *y)
{
int z = *x;
*x = *y;
*y = z;
}
Main(){swap(&a, &b);….}
	void swap(int& x, int& y)
	{
	int z = x;
	x = y;
	y = z;
	}
	Main(){swap(a, b);….}

We pass the address of arguments in the function call.	We pass the arguments in the function call.
The value of the arguments is accessed via the dereferencing operator *	The reference name can be used to implicitly reference a value
Passed parameters can be moved/reassigned to a different memory location.	Parameters can't be moved/reassigned to another memory address.
Pointers can contain a NULL value, so a passed argument may point to a NULL or even a garbage value.	References cannot contain a NULL value, so it is guaranteed to have some value.

Difference Between Reference Variable and Pointer Variable

- A reference is the same object, just with a different name and a reference must refer to an object. Since references can't be NULL, they are safer to use
- A pointer can be re-assigned while a reference cannot, and must be assigned at initialization only.
- The pointer can be assigned NULL directly, whereas the reference cannot.
- Pointers can iterate over an array, we can use increment/decrement operators to go to the next/previous item that a pointer is pointing to.
- A pointer is a variable that holds a memory address. A reference has the same memory address as the item it references.
- A pointer needs to be dereferenced with * to access the memory location it points to, whereas a reference can be used directly.

'this' Pointer in C++

- this is a keyword that refers to the current instance of the class.
- The 'this' pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions.

- ‘this’ pointer is not available in static member functions as static member functions can be called without any object (with class name). Even if only one member of each function exists which is used by multiple objects, the compiler supplies an implicit pointer along with the names of the functions as ‘this’.
- this->x = x;
- To understand ‘this’ pointer, it is important to know how objects look at functions and data members of a class.
- Each object gets its own copy of the data member.
- All-access the same function definition as present in the code segment. Meaning each object gets its own copy of data members and all objects share a single copy of member functions.
- Then now question is that if only one copy of each member function exists and is used by multiple objects, how are the proper data members are accessed and updated? The compiler supplies an implicit pointer along with the names of the functions as ‘this’.
- The ‘this’ pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. ‘this’ pointer is not available in static member functions as static member functions can be called without any object (with class name).
- For a class X, the type of this pointer is ‘X* ‘.
- Also, if a member function of X is declared as const, then the type of this pointer is ‘const X *’

Class pointer

- A pointer to a C++ class is done exactly the same way as a pointer to a structure and to access members of a pointer to a class you use the member access operator -> operator, just as you do with pointers to structures. Also as with all pointers, you must initialize the pointer before using it.

Structures

- The structure is a user-defined data type that is available in C++.
- Structures are used to combine different types of data types, just like an array is used to combine the same type of data types.
- A structure is declared by using the keyword “struct“. When we declare a variable of the structure we need to write the keyword “struct in C language but for C++ the keyword is not mandatory
- The are same as class only difference is that their members are by default public

```
struct
{
  // Declaration of the struct
}
```

Structure using typedef

- typedef is a keyword that is used to assign a new name to any existing data-type.
- Below is the C++ program illustrating use of struct using typedef:

```
typedef struct myStruct {
   int s1;
   char s2;
   float s3;
}str1;
int main()
{
   // Declaring a Structure
   //struct myStruct hello;
        str1 hello;
   hello.s1 = 85;
   hello.s2 = 'G';
```

- In the above code, the keyword “typedef” is used before struct and after the closing bracket of structure, “str1” is written.
- Now create structure variables without using the keyword “struct” and the name of the struct.
- A structure instance has been created named “hello” by just writing “str1” before it.

Enumeration

- Enums: Enums are user-defined types that consist of named integral constants.
- It helps to assign constants to a set of names to make the program easier to read, maintain and understand.
- An Enumeration is declared by using the keyword “enum“.

Dynamic Memory Allocation in C using malloc(), calloc(), free() and realloc()

- Dynamic Memory Allocation can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.
- C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under <stdlib.h> header file to facilitate dynamic memory allocation in C programming.
    - malloc()

    - calloc()
    - free()
    - realloc()
- The “malloc” or “memory allocation” method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn’t Initialize memory at execution time so that it has initialized each block with the default garbage value initially.
- *ptr = (int*) malloc(100 * sizeof(int));*
  Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.

```
int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    printf("Enter number of elements:");
    scanf("%d",&n);
    printf("Entered number of elements: %d\n", n);

    // Dynamically allocate memory using malloc()
    ptr = (int*)malloc(n * sizeof(int));
  // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
```

```
        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }
   free(ptr);
    return 0;
}
```

C calloc() method

1. “calloc” or “contiguous allocation” method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:
2. It initializes each block with a default value ‘0’.
3. It has two parameters or arguments as compare to malloc().

- *ptr = (float*) calloc(25, sizeof(float));*
  This statement allocates contiguous space in memory for 25 elements each with the size of the float.

C free() method

- “free” method in C is used to dynamically de-allocate the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.
- free(ptr);

C realloc() method

- “realloc” or “re-allocation” method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to dynamically re-allocate memory. re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.
- ptr = realloc(ptr, newSize);

```
// Dynamically re-allocate memory using realloc()
    ptr = (int*)realloc(ptr, n * sizeof(int));

    // Memory has been successfully allocated
    printf("Memory successfully re-allocated using realloc.\n");

    // Get the new elements of the array
    for (i = 5; i < n; ++i) {
        ptr[i] = i + 1;
    }

    // Print the elements of the array
    printf("The elements of the array are: ");
    for (i = 0; i < n; ++i) {
        printf("%d, ", ptr[i]);
    }

    free(ptr);
}

    return 0;
```

new and delete Operators in C++ For Dynamic Memory

- Dynamic memory allocation in C/C++ refers to performing memory allocation manually by a programmer.
- Dynamically allocated memory is allocated on Heap, and non-static and local variables get memory allocated on Stack

What are applications?

- One use of dynamically allocated memory is to allocate memory of variable size, which is not possible with compiler allocated memory except for variable-length arrays.
- The most important use is the flexibility provided to programmers. We are free to allocate and deallocate memory whenever we need it and whenever we don't need it anymore. There are many cases where this flexibility helps. Examples of such cases are Linked List, Tree, etc.
- For normal variables like "int a", "char str[10]", etc, memory is automatically allocated and deallocated.
- For dynamically allocated memory like "int *p = new int[10]", it is the programmer's responsibility to deallocate memory when no longer needed.
- If the programmer doesn't deallocate memory, it causes a memory leak (memory is not deallocated until the program terminates).

How is memory allocated/deallocated in C++?

- C uses the malloc() and calloc() function to allocate memory dynamically at run time and uses a free() function to free dynamically allocated memory.
- C++ supports these functions and also has two operators new and delete, that perform the task of allocating and freeing the memory in a better and easier way.

new operator

- The new operator denotes a request for memory allocation on the Free Store.
- If sufficient memory is available, a new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable
- pointer-variable = new data-type;
- Here, the pointer variable is the pointer of type data-type. Data type could be any built-in data type including array or any user-defined data type including structure and class.

```
// Pointer initialized with NULL

// Then request memory for the variable

int *p = NULL;

p = new int;

        OR

// Combine declaration of pointer

// and their assignment

int *p = new int;
```

- Initialize memory: We can also initialize the memory for built-in data types using a new operator. For custom data types, a constructor is required (with the data type as input) for initializing the value. Here's an example of the initialization of both data types :

```
pointer-variable = new data-type(value);
int* p = new int(25);
float* q = new float(75.25);

// Custom data type
struct cust
{
    int p;
    cust(int q) : p(q) {}
    cust() = default;
    //cust& operator=(const cust& that) = default;
};
int main()
{
    // Works fine, doesn't require constructor
    cust* var1 = new cust;

    //OR

    // Works fine, doesn't require constructor
    var1 = new cust();

    // Notice error if you comment this line
    cust* var = new cust(25);
    return 0;
}
```

- Allocate a block of memory: a new operator is also used to allocate a block(an array) of memory of type *data type*.
- pointer-variable = new data-type[size];
- where size(a variable) specifies the number of elements in an array.
- int *p = new int[10];

- Dynamically allocates memory for 10 integers continuously of type int and returns a pointer to the first element of the sequence, which is assigned top(a pointer). p[0] refers to the first element, p[1] refers to the second element, and so on.

Normal Array Declaration vs Using new

- There is a difference between declaring a normal array and allocating a block of memory using new. The most important difference is, that normal arrays are deallocated by the compiler
- However, dynamically allocated arrays always remain there until either they are deallocated by the programmer or the program terminates.
- What if enough memory is not available during runtime?
- If enough memory is not available in the heap to allocate, the new request indicates failure by throwing an exception of type std::bad_alloc, unless “nothrow” is used with the new operator, in which case it returns a NULL pointer
- Therefore, it may be a good idea to check for the pointer variable produced by the new before using its program.
- delete operator -Since it is the programmer’s responsibility to deallocate dynamically allocated memory, programmers are provided delete operator in C++ language.
- delete pointer-variable;
- Here, the pointer variable is the pointer that points to the data object created by *new*.

new vs malloc() and free() vs delete in C++

- We use new and delete operators in C++ to dynamically allocate memory whereas malloc() and free() functions are also used for the same purpose in C and C++.
- The functionality of the new or malloc() and delete or free() seems to be the same but they differ in various ways.
- The behavior with respect to constructors and destructors calls differ in the following ways:
  malloc() vs new():
- malloc(): It is a C library function that can also be used in C++, while the “new” operator is specific for C++ only.

- Both malloc() and new are used to allocate the memory dynamically in heap. But “new” does call the constructor of a class whereas “malloc()” does not.
- free() vs delete:
- free() is a C library function that can also be used in C++, while “delete” is a C++ keyword.
- free() frees memory but doesn’t call Destructor of a class whereas “delete” frees the memory and also calls the Destructor of the class.

What is Memory Leak?

- A memory leak occurs when programmers create a memory in a heap and forget to delete it.
- The consequence of the memory leak is that it reduces the performance of the computer by reducing the amount of available memory. Eventually, in the worst case, too much of the available memory may become allocated, all or part of the system or device stops working correctly, the application fails, or the system slows down vastly.
- Memory leaks are particularly serious issues for programs like daemons and servers which by definition never terminate.

```
void f()
{
    int* ptr = (int*)malloc(sizeof(int));

    /* Do some work */

    /* Return without freeing ptr*/
    return;
}
```

How to avoid memory leaks?

- To avoid memory leaks, memory allocated on the heap should always be freed when no longer needed.

```
/* Function without memory leak */
#include <stdlib.h>

void f()
{
    int* ptr = (int*)malloc(sizeof(int));

    /* Do some work */

    /* Memory allocated by malloc is released */
    free(ptr);
    return;
}
```