Arrays, String, Pointers & Reference

INDEX

Table of Contents

[Arrays in C++ 3](#_Toc145627758)

[Array Declaration & Initialization 3](#_Toc145627759)

[Multi-Dimensional Array Declaration & Initialization 4](#_Toc145627760)

[More ways to create Multi-Dimensional Array 6](#_Toc145627761)

[Memory Layout 9](#_Toc145627762)

[Sizeof Array 10](#_Toc145627763)

[Array Traversal in C++ 11](#_Toc145627764)

[Different Type of Array 14](#_Toc145627765)

[Access elements in a 2D array 14](#_Toc145627766)

[Passing 2D arrays as arguments in C++ 15](#_Toc145627767)

[Strings in C++ 22](#_Toc145627768)

[C-style (character arrays and literals) 22](#_Toc145627769)

[String Class in C++ STL 23](#_Toc145627770)

[String manipulations 23](#_Toc145627771)

[Pointers & References in C++ 29](#_Toc145627772)

[Pointers 29](#_Toc145627773)

[Address and Dereference Operator 29](#_Toc145627774)

[Difference between arr and &arr 30](#_Toc145627775)

[Use case 31](#_Toc145627776)

[Drawbacks of Pointers 34](#_Toc145627777)

[Reference 35](#_Toc145627778)

[Use case 35](#_Toc145627779)

[Const & R-Value Reference 37](#_Toc145627780)

[Pointers vs References 40](#_Toc145627781)

[Pointers vs Array 41](#_Toc145627782)

[Function Parameters 44](#_Toc145627783)

[Call-by-Value 44](#_Toc145627784)

[Call-by-Reference with Pointer Arguments 45](#_Toc145627785)

[Call-by-Reference with Reference Arguments 46](#_Toc145627786)

[Array Name as Pointers 47](#_Toc145627787)

[Pointers and String literals 48](#_Toc145627788)

[String Literals as Pointers 48](#_Toc145627789)

[Modifiable Strings 48](#_Toc145627790)

[C++ string Class 49](#_Toc145627791)

[Pointers to pointers 50](#_Toc145627792)

[Void Pointers 51](#_Toc145627793)

[NULL Pointer 52](#_Toc145627794)

[nullptr 53](#_Toc145627795)

# Arrays in C++

Array is a collection of similar types of data elements, stored at continuous memory locations. It is a built-in data type. We cannot change the size and type of arrays after its declaration.

A colorful rectangular object with numbers and symbols

Description automatically generated

## Array Declaration & Initialization

int n = 5; // Declare 'n' as 5  
  
//Declare an integer array 'arr1' with 5 elements (uninitialized).  
int arr1[5];  
  
//Declare and initialize an integer array 'arr2' with values 1 to 5.  
int arr2[5] = {1, 2, 3, 4, 5};  
  
//Declare and initialize an integer array 'arr3' with values 1 to 5.  
//The compiler determines the array size based on the number of provided values.  
int arr3[] = {1, 2, 3, 4, 5};  
  
// Declare an integer array 'arr4' with 'n' elements.  
int arr4[n];

When you explicitly assign a value to an element of an array in C++, the other indexes that you have not explicitly initialized will be automatically set to 0.



## Multi-Dimensional Array Declaration & Initialization

* A 2D array defined as follows can be viewed as a table of two rows and three columns. There are several ways in which a 2D Array can be initialized.
* Each set of inner braces represents one row.

int arr[2][3] = { {1, 2, 3}, {4, 5, 6} };  
or  
int arr[2][3] = {  
 {1, 2, 3},  
 {4, 5, 6}  
};

* Elements stored in row major order.
* When initializing a multi-dimensional array, you can omit the size of the first dimension, and the compiler will automatically determine it based on the number of elements provided.

int arr[][3] = {  
 {1, 2, 3},  
 {4, 5, 6}  
};



int arr[2][3]; // Declare the 2D array  
  
// Input values for each element  
for (int i = 0; i < 2; i++) {  
 for (int j = 0; j < 3; j++) {  
 cout << "Enter value for arr[" << i << "][" << j << "]: ";  
 cin >> arr[i][j];  
 }  
}

**3D Array Declaration & Initialization**

A screenshot of a computer

Description automatically generated

## More ways to create Multi-Dimensional Array

#### Double Pointer

**Advantage**

We can create **Jagged array** (an array of arrays with varying sizes) using this Double Pointer approach **(int\*\*)**. This allows you to have different column sizes for each row.

**Disadvantage**

one of the disadvantages of using a jagged array, created using a double pointer approach (int\*\*), is that the **elements are not stored in contiguous memory locations**. Each row is allocated separately, which means that the memory for each row may be located at different addresses.

The lack of contiguous memory storage can have performance implications, especially in scenarios where cache-friendliness. **Accessing elements in non-contiguous memory locations may result in more cache misses and slower access times** compared to a single contiguous block of memory,

int\*\* arr = new int\*[2]; // Allocate memory for an array of pointers  
  
// Allocate memory for each row and input values  
for (int i = 0; i < 2; i++) {  
 arr[i] = new int[3]; // Each row has 3 columns  
 for (int j = 0; j < 3; j++) {   
 cout << "Enter value for arr[" << i << "][" << j << "]: ";   
 cin >> arr[i][j];  
 }  
}

#### Array of Pointers

When using the int \*arr[] approach to create a jagged array (an array of pointers to sub-arrays), the memory for the array of pointers (sub-arrays) is typically allocated on the stack. This is difference to the previous method **(int\*\*)** where memory for the individual sub-arrays was allocated on the heap.

int\* arr[2]; // Array of pointers to integers  
int n = 3; // Fixed column size for each sub-array  
  
// Initialize each element with dynamically allocated arrays  
for (int i = 0; i < 2; i++) {  
 arr[i] = new int[n]; // Allocate memory for sub-array with size3  
 for (int j = 0; j < n; j++) {  
 arr[i][j] = i \* 3 + j + 1; // Initialize values  
 }  
}

#### Array of Vectors

**Advantage**

* One of the advantages of using an array of vectors (vector<int> arr[m]) over other method is that it allows for dynamic column sizes within each row. Each vector can have a different number of elements, which can be convenient in situations where the column sizes vary from row to row.
* Easy to pass to function.

**Disadvantage**

The code provided in the example, which use an array of vectors **(vector<int> arr[m])**, does not store data in contiguous memory locations. Each vector within the array is allocated on the heap and can be located at different memory addresses, resulting in non-contiguous memory storage.

As a result, accessing elements in this structure may not be cache-friendly, and it can have implications for memory access performance, especially when dealing with large data sets.

If you need to work with large datasets and require contiguous memory storage for better cache performance, you might consider alternative data structures or memory allocation strategies. For example, you could use a dynamically allocated **2D array (int\*\*)** or a custom data structure that uses contiguous memory.

int main() {  
 int m = 3; // Number of rows  
 int n = 2; // Number of columns  
 vector<int> arr[m]; // Array of vectors to store integers  
  
 // Nested loops to fill the 2D array with 10s  
 for (int i = 0; i < m; i++) {  
 for (int j = 0; j < n; j++) {  
 arr[i].push\_back(10); // Push 10 into each vector  
 }  
 }  
 return 0;  
}

#### Vector of Vectors

**Additional Advantage**

* One of the significant advantages of using a vector of vectors (vector<vector<int>>) to create a 2D dynamic array is that the number of rows can be dynamic. You can easily add or remove rows as needed without the need for pre-declaring a fixed size.
* Other advantages are same as method 3.

int main() {  
 int m = 3; // Number of rows  
 int n = 2; // Number of columns  
 vector<vector<int>> arr; // Declare a vector of vectors  
  
 for (int i = 0; i < m; i++) {  
 vector<int> v; // Declare a temporary vector for each row  
 for (int j = 0; j < n; j++) {  
 v.push\_back(10); // Push 10 into each row's vector  
 }  
 arr.push\_back(v); // Push the row's vector into the main vector  
 }  
 return 0;  
}

**Remember**

Here's a concise note highlighting the key difference between double pointers (int\*\* arr) and an array of pointers (int\* arr [2]):

**Double Pointer (int\*\* arr):**

* Memory for both the array of pointers and the individual rows (2D array) is dynamically allocated on the heap using new.
* Requires manual deallocation of both the array of pointers and individual rows using delete[].

**Array of Pointers (int\* arr[2]):**

* Memory for the array of pointers (sub-arrays) is typically allocated on the stack or as an automatic variable.
* Requires manual deallocation of the individual sub-arrays (pointed to by the pointers) using delete[].
* Suitable for creating an array of fixed-size sub-arrays with a fixed number of rows.

## Memory Layout

A contiguous memory space is allocated for array elements and can be accessed via an array index.

A diagram of a number of numbers

Description automatically generated with medium confidence

## Sizeof Array

int main() {  
 int arr[5]; // Declare an array of 5 integers  
  
 // Get the total size of the array in bytes  
 size\_t size1 = sizeof(arr);  
  
 cout << "Size of the array: " << size1 << " bytes" << endl;  
  
 // Get the size of a single element in the array in bytes  
 size\_t size2 = sizeof(arr[0]);  
  
 cout << "Size of a single element: " << size2 << " bytes\n";  
  
 // Calculate the number of elements in the array  
 size\_t size3 = sizeof(arr) / size2;  
  
 cout << "Number of elements: " << size3 << endl;  
 return 0;  
}  
Output  
Size of the array: 20 bytes  
Size of a single element: 4 bytes  
Number of elements: 5

## Array Traversal in C++

There are three ways to traverse the elements of an array in C++:

1. **Using for loop**

int main() {  
 int arr[] = {1, 2, 3, 4, 5};  
 int size = sizeof(arr) / sizeof(arr[0]);  
 for (int i = 0; i < size; ++i) {  
 cout << arr[i] << " ";  
 }  
 return 0;  
}  
Output  
1 2 3 4 5

1. **Using for\_each loop (C++11 and later)**

#include <iostream>  
#include <algorithm>  
using namespace std;  
int main() {  
 int arr[] = {1, 2, 3, 4, 5};  
 int size = sizeof(arr) / sizeof(arr[0]);  
  
 // Using std::for\_each algorithm  
 for\_each(arr, arr + size, [](int num) {  
 cout << num << " ";  
 });  
 return 0;  
}  
Output  
1 2 3 4 5

TODO: Difference between for\_each and range-based loop because everywhere both have same example.

1. **using range-based for loop**

**int var** will hold each element of the array arr in each iteration.

int main() {  
 int arr[] = {1, 2, 3, 4, 5};  
 // Using a range-based for loop to traverse the array  
 for (int var : arr) {  
 cout << var << " "; // Output the current element  
 }  
 return 0;  
}

Works same as above. Here, the **auto** keyword is used for type inference. It allows the compiler to automatically deduce the appropriate data type for the loop variable var based on the type of the elements in the array.

int main() {  
 int arr[] = {1, 2, 3, 4, 5};  
 // Using a range-based for loop to traverse the array  
 for (auto var : arr) {  
 cout << var << " "; // Output the current element  
 }  
 return 0;  
}

**const auto &** syntax is used to capture each element by reference for efficiency and to indicate that the loop will not modify the elements.

Using const auto &var reference, var will directly access the read only memory for each element of the array in each iteration but will not modify the elements.

int main() {  
 int arr[] = {1, 2, 3, 4, 5};  
  
 // Using a range-based for loop to traverse the array  
 for (const auto &var : arr) {  
 cout << var << " "; // Output the current element  
 }  
  
 return 0;  
}

**Modifications During Array Traversal**:

int main() {  
 int arr[] = {1, 2, 3, 4, 5};  
 int n = sizeof(arr) / sizeof(arr[0]);  
  
 // Double each element in the array  
 for (int i = 0; i < n; i++) {  
 arr[i] = arr[i] \* 2;  
 }  
  
 // Print the updated array  
 for (int i = 0; i < n; i++) {  
 cout << arr[i] << " ";  
 }  
  
 return 0;  
}  
Output  
2 4 6 8 10

Using auto &var reference, var will directly access the memory for each element of the array in each iteration and do post-increments.

int main() {  
 int arr[] = {1, 2, 3, 4, 5};  
  
 // Using a range-based for loop to traverse the array  
 for (auto &var : arr) {  
 cout << var++ << " "; // Print original value  
 }  
 cout << endl;  
  
 // Using another range-based for loop to traverse the array  
 for (const auto &var : arr) {  
 cout << var << " "; // Print modified value  
 }  
 return 0;  
}  
Output  
1 2 3 4 5  
2 3 4 5 6

## Different Type of Array

**Fixed Size Arrays (Allocated in Function Call Stack):**

1. int arr [] = {1, 2, 3, 4, 5}
2. int arr [5]
3. int a[n]; (n is a known value or a constant)

**Fixed Size Arrays (Allocated in Heap - Dynamic Memory Allocation):**

int\* a = new int[n]; (Allocated in Heap)

**Dynamic Sized Arrays:**

Vector in C++ STL

## Access elements in a 2D array

**Common way to access elements in a 2D array: \*(arr + x \* m + y)**,is used to access the value at the address corresponding to the element arr[x][y] in a 2D array represented as a contiguous block of memory.

Here is a breakdown of how the formula works:

* arr is the base address of the 2D array.
* x represents the row index.
* m is the number of columns in each row.
* y represents the column index.



## Passing 2D arrays as arguments in C++

Approach 1

**Problem**: print function is not generic and is specifically designed to print a 3x2 matrix.

void print(int mat[3][2]) {  
 for (int i = 0; i < 3; i++) {  
 for (int j = 0; j < 2; j++) {  
 cout << mat[i][j] << " ";  
 }  
 }  
}  
  
int main() {  
 int mat[3][2] = {{10, 20}, {30, 40}, {50, 60}};  
 print(mat);  
 return 0;  
}  
Output  
10 20 30 40 50 60

Approach 2

**Problem**: When the number of rows (denoted as 'm') is passed as a parameter, the print function is specifically designed to print a matrix with 'm' rows and a fixed number of columns (e.g., 2).

void print(int mat[][2], int m) {  
 for (int i = 0; i < m; i++) {  
 for (int j = 0; j < 2; j++) {  
 cout << mat[i][j] << " ";  
 }  
 }  
}  
  
int main() {  
 int mat[3][2] = {{10, 20}, {30, 40}, {50, 60}};  
 print(mat, 3);  
 return 0;  
}  
Output

10 20 30 40 50 60

Unfortunately, in C++, there is no direct syntax for traditional multidimensional arrays that allows you to specify the number of rows and columns as separate parameters.

Approach 3

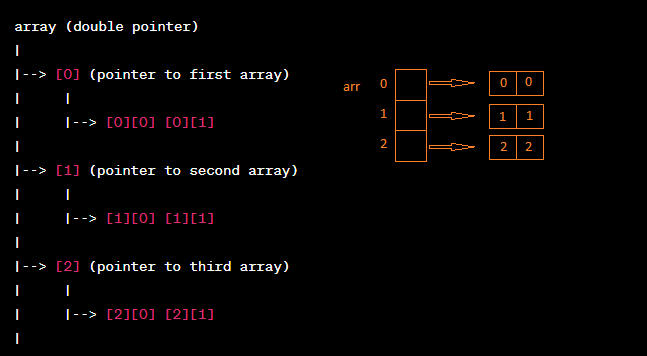
**Problem**: These arrays are not designed to be general-purpose, as the row and column sizes are provided as global constant variables and are intended to remain fixed throughout the program.

const int R = 3;  
const int C = 2;  
  
void print(int mat[R][C]) {  
 for (int i = 0; i < R; i++) {  
 for (int j = 0; j < C; j++) {  
 cout << mat[i][j] << " ";   
 }   
 }  
}  
  
int main() {  
 int mat[R][C] = {{10, 20}, {30, 40}, {50, 60}};  
 print(mat);  
 return 0;  
}  
Output  
10 20 30 40 50 60

Approach 4 - C-Style solution

Solution 1 - Using Double Pointer

#include <iostream>  
using namespace std;  
  
void print(int\*\* arr, int m, int n) {  
 for (int i = 0; i < m; i++) {  
 for (int j = 0; j < n; j++) {  
 cout << arr[i][j] << " ";  
 }  
 }  
}  
  
int main() {  
 int m = 3, n = 2;  
  
 // Dynamically allocate memory for the 2D array  
 int\*\* arr = new int\*[m];   
  
 // Initialize and print the array  
 for (int i = 0; i < m; i++) {  
 arr[i] = new int[n];  
 for (int j = 0; j < n; j++) {  
 arr[i][j] = i;   
 }  
 }  
  
 // Call the print function to display the entire array  
 print(arr, m, n);  
  
 // Deallocate memory  
 for (int i = 0; i < m; i++) {  
 delete[] arr[i];  
 }   
 delete[] arr;  
 return 0;  
}  
Output  
0 0 1 1 2 2



Solution 2 – Array of Pointers

#include <iostream>  
using namespace std;  
  
void print(int\*arr[], int m, int n) {  
 for (int i = 0; i < m; i++) {  
 for (int j = 0; j < n; j++) {  
 cout << arr[i][j] << " ";  
 }  
 }  
}  
  
int main() {  
 int m = 3, n = 2;  
  
 int \*arr[m];   
  
 for (int i = 0; i < m; i++) {  
 arr[i] = new int[n];  
 for (int j = 0; j < n; j++) {  
 arr[i][j] = i;  
 }  
 }  
  
 // Call the print function to display the entire array  
 print(arr, m, n);  
  
 // Deallocate memory  
 for (int i = 0; i < m; i++) {  
 delete[] arr[i];  
 }  
  
 return 0;  
}  
Output  
0 0 1 1 2 2

Approach 5 – C++ Style solution

Solution 1 – Array of Vectors

#include <iostream>  
#include <vector>  
using namespace std;  
  
void print(vector<int> arr[], int m) {  
 for (int i = 0; i < m; i++) {  
 for (int j = 0; j < arr[i].size(); j++) {  
 cout << arr[i][j] << " ";  
 }  
 }  
}  
  
int main() {  
 int m = 3, n = 2;  
 vector<int> arr[m];  
  
 for (int i = 0; i < m; i++) {  
 for (int j = 0; j < n; j++) {  
 arr[i].push\_back(i);  
 }  
 }  
  
 print(arr, m);  
 return 0;  
}  
Output  
0 0 1 1 2 2

Solution 2 – Vector of Vector

Extra Benefit with vector of vectors is number of rows are also dynamic in nature.

#include <iostream>  
#include <vector>  
using namespace std;  
  
void print(vector<vector<int>>& arr) {  
 for (int i = 0; i < arr.size(); i++) {  
 for (int j = 0; j < arr[i].size(); j++) {  
 cout << arr[i][j] << " ";  
 }  
 }  
}  
  
int main() {  
 int m = 3, n = 2;  
  
 vector<vector<int>> arr;  
  
 for (int i = 0; i < m; i++) {  
 vector<int> v;  
 for (int j = 0; j < n; j++) {  
 v.push\_back(i);  
 }  
 arr.push\_back(v);  
 }  
  
 print(arr);  
 return 0;  
}  
Output  
0 0 1 1 2 2

**Remember**

* We can use jagged array using this (C-style & C++ style) solution.
* Disadvantage of these solutions that rows are not stored at contiguous memory location, so they are slightly less cache friendly to 2D native array.

# Strings in C++

There are two ways of handling strings in C++.

1. **C-style** strings or character arrays.
2. string template class

## C-style (character arrays and literals)

A screenshot of a computer program

Description automatically generated

## String Class in C++ STL

A screen shot of a computer

Description automatically generated

### String manipulations

**Remember**

An **iterator** is an object (like a pointer) that points to an element inside the container. Containers are.



Each container class in the C++ Standard Library provides its own specific type of iterator. Few Examples:

|  |  |
| --- | --- |
| vector<int>::iterator | list<double>::iterator |
| set<std::string>::iterator | map<int, std::string>::iterator |

*String Manipulation Functions in the std::string Class*

**String Length and Capacity**

Size: size refers to the number of characters in the string.

Capacity:

* The capacity of a container refers to the amount of memory that has been allocated for it, which determines how many elements it can hold efficiently.
* Capacity refers to the number of characters that the **string can hold without needing to reallocate memory**. How capacity decide
* When appending characters to a string, if the size exceeds the capacity, the string might need to be reallocated to accommodate the new characters.

int main() {  
 std::string myString = "Hello, World!";  
   
 // Get the size and capacity of the string  
 cout << "Size of the string: " << myString.size() << “\n”;  
 // same as size  
 cout << "Length of the string: " << myString.length() << “\n”;   
 cout << "Capacity of the string: " << myString.capacity() << “\n”;   
  
 // Check if the string is empty  
 if (myString.empty()) {  
 cout << "The string is empty.\n";  
 } else {  
 cout << "The string is not empty.\n";  
 }  
   
 // Reserve additional capacity  
 myString.reserve(30);  
   
 // Check the new capacity  
 std::cout << "New capacity after reserving: " << myString.capacity() << “\n”;   
   
 // Clear the string  
 myString.clear();  
   
 // Check if the string is empty after clearing  
 if (myString.empty()) {  
 cout << "The string is now empty.\n";   
 } else {  
 cout << "The string is not empty.\n";  
 }  
 return 0;  
}

Output  
Size of the string: 13  
Length of the string: 13  
Capacity of the string: 15  
The string is not empty.  
New capacity after reserving: 30  
The string is now empty.

**String Iteration**

int main() {  
 string str = "Hello world";  
 cout << "Print first character: " << \*str.begin() << endl;  
 cout << "Print last character: " << \*(str.end() - 1) << endl;  
 return 0;  
}  
Output  
Print first character: H  
Print last character: d

**String Concatenation and Append**

int main() {  
 string str = "Hello, ";  
  
 // String Concatenation using + operator  
 str += "world!";  
 cout << "Concatenated string: " << str << endl;  
  
 // String Appending using append() function  
 str.append(" How are you?");  
 cout << "Appended string: " << str << endl;  
  
 // Push back characters using push\_back()  
 str.push\_back('x'); // Push a specific character ('x')  
 cout << "After push\_back(): " << str << endl;  
  
 // Pop back characters using pop\_back()  
 str.pop\_back(); // Remove the last character ('x')  
 cout << "After pop\_back(): " << str << endl;  
 return 0;  
}  
Output  
Concatenated string: Hello, world!  
Appended string: Hello, world! How are you?  
After push\_back(): Hello, world! How are you?x  
After pop\_back(): Hello, world! How are you?

**insert**: It used to insert characters or a substring into a string at a specified position.

int main() {  
 string originalString = "Hello, world!";  
 string insertedString = "beautiful ";  
  
 // Insert "beautiful " at position 7 in the original string  
 originalString.insert(7, insertedString);  
  
 cout << "Modified string: " << originalString << endl;  
  
 return 0;  
}  
Output  
Modified string: Hello, beautiful world!

**find and rfind**:

int main() {  
 string str = "Hello, how are you? how's it going?";  
 string target = "how";  
  
 // Using find() to find the first occurrence of "How"  
 size\_t pos1 = str.find(target);  
 if (pos1 != string::npos) {  
 cout << "First 'how' found at position: " << pos1 << endl;  
 } else {  
 cout << "'how' not found." << endl;  
 }  
  
 // Using rfind() to find the last occurrence of "How"  
 size\_t pos2 = str.rfind(target);  
 if (pos2 != string::npos) {  
 cout << "Last 'how' found at position: " << pos2 << endl;  
 } else {  
 cout << "'how' not found." << endl;  
 }  
  
 return 0;  
}  
Output  
First 'how' found at position: 13  
Last 'how' found at position: 27

**substr**:

int main() {  
 string originalString = "Hello, world!";  
   
 // Extract from position 7 to the end  
 string sub1 = originalString.substr(7);  
   
 // Extract first 5 characters  
 string sub2 = originalString.substr(0, 5);  
   
 cout << "Sub1: " << sub1 << '\n';  
 cout << "Sub2: " << sub2 << '\n';  
   
 return 0;  
}  
Output  
Sub1: world!  
Sub2: Hello

**find\_first\_of and find\_last\_of:**

int main() {  
 string str = "Hello, World!";  
 string characters = "lo";  
  
 size\_t first = str.find\_first\_of(characters);  
 size\_t last = str.find\_last\_of(characters);  
  
 cout << "First occurrence of 'l' or 'o' at position: " << first << endl;  
 cout << "Last occurrence of 'l' or 'o' at position: " << last << endl;  
  
 return 0;  
}  
Output  
First occurrence of 'l' or 'o' at position: 2  
Last occurrence of 'l' or 'o' at position: 10

**compare**:

int main() {  
 string sentence1 = "Hello, world!";  
 string sentence2 = "Hello, universe!";  
   
 int result = sentence1.compare(sentence2);  
  
 if (result == 0) {  
 cout << "The sentences are identical." << endl;  
 } else if (result < 0) {  
 cout << "Sentence 1 comes before Sentence 2." << endl;  
 } else {  
 cout << "Sentence 2 comes before Sentence 1." << endl;  
 }  
  
 return 0;  
}  
Output  
Sentence 2 comes before Sentence 1.

TODO: Confusion with find\_first\_of and find\_last\_of – if it searches first char in the string then why it takes string as argument. It should take one char only.

# Pointers & References in C++

## Pointers

Pointer is a special type of variable that holds the memory address of another variable. It allows you to indirectly access and manipulate data.

### Address and Dereference Operator

**Address-of Operator (&)**: If you use the address-of operator (&) before a variable, you can obtain the memory address where the variable is stored.

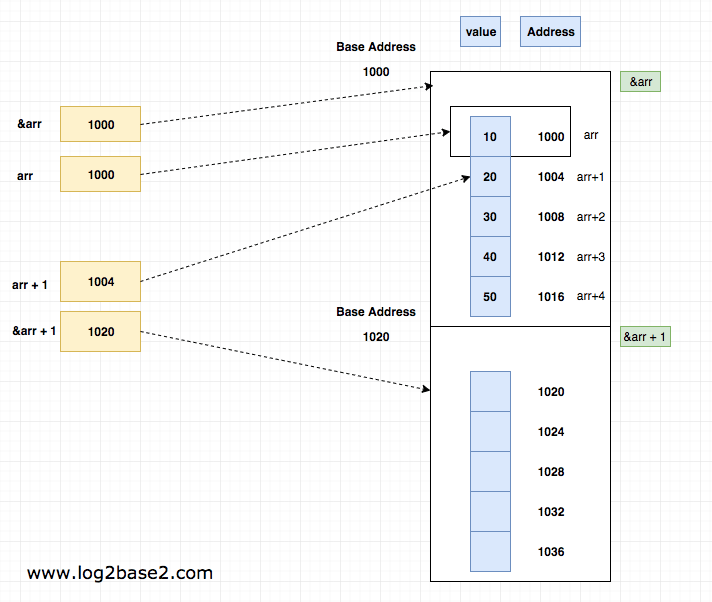
**Dereference Operator (\*)**: If you use the dereference operator (\*) before a pointer variable, you can access the value stored at the memory address pointed to by the pointer. **Aka Value at Address**

Pointers is a variable that holds the memory address of another variable.



### Difference between arr and &arr

* arr is an integer pointer (int\*) which points the first element of the array.
* &arr is an integer array pointer (int\*) [5] which points the whole array. (All five elements.)



int main() {  
 int arr[5] = {10, 20, 30, 40, 50};  
   
 // Pointer to an integer  
 int\* ptr1 = arr;  
   
 // Pointer to an array of 5 integers  
 int (\*ptr2)[5] = &arr;  
   
 cout << \*ptr1 << “ ”; // Value at the first element  
 cout << \*\*ptr2; // Value at the first element  
 return 0;  
}  
Output  
10 10

Use case:

* **Dynamic Memory Allocation**: This is useful when you need memory whose size is determined at runtime.

int\* dynamicArray = new int[10]; // Allocate an array of 10 integers   
delete[] dynamicArray; // Deallocate the memory when done

* **Passing Parameters by Reference**: It allow the function to modify the original variable.

void modifyValue(int\* numPtr) {  
 \*numPtr = 20; // Modifies the original variable  
}  
  
int main() {  
 int x = 10;  
 modifyValue(&x); // Pass by pointer  
 cout << "Modified value of x: " << x << endl;  
 return 0;  
}  
Output  
Modified value of x: 20

* **Pointers to Functions**: Pointers can be used to pass data structures or large objects efficiently to functions without making copies.

void processArray(int\* array, int size) {  
 // Process the array without copying  
 for (int i = 0; i < size; i++) {  
 // Example: Print each element  
 cout << array[i] << " ";  
 }  
 cout << endl;  
}  
  
int main() {  
 int numbers[] = {10, 20, 30};  
   
 // Call the processArray function  
 processArray(numbers, 3);  
  
 return 0;  
}  
Output  
10 20 30

* **Array Manipulation**

int main() {  
 int numbers[] = {10, 20, 30, 40, 50};  
  
 // Using pointer arithmetic to iterate through the array  
 for (int\* ptr = numbers; ptr < numbers + 5; ++ptr) {  
 cout << \*ptr << "; "; // Print the current element  
 }  
 cout << endl;  
  
 // Accessing a specific element using pointer arithmetic  
 int\* thirdElement = numbers + 2;  
 cout << "Third element: " << \*thirdElement << endl;  
  
 return 0;  
}  
Output  
10; 20; 30; 40; 50;   
Third element: 30

* **Working with Hardware**

Program interacts with hardware registers (at address 0x1000) in embedded systems, writing and printing values specific to the hardware platform.

// Simulated hardware registers  
volatile unsigned int\* hardwareRegister = reinterpret\_cast<volatile unsigned int\*>(0x1000);  
  
int main() {  
 // Write a value to the hardware register  
 \*hardwareRegister = 0xABCDEF;  
  
 // Read and print the value from the hardware register  
 unsigned int value = \*hardwareRegister;  
 cout << "Value from hardware register: " << hex << value << endl;  
  
 return 0;  
}

* **Implement a wide range of data structures.**

List of data structures implemented using pointers:

Linked Lists, Stacks, Queues, Trees (Binary Trees, Binary Search Trees, AVL Trees), Graphs, Hash Tables, Heaps (Min Heaps, Max Heaps), Hash Maps, Tries, Disjoint Set (Union-Find), Sparse Matrices, Circular Buffers, Skip Lists

* **Working with Dynamic Data Structures**: Pointers are essential when creating dynamic data structures like linked lists, trees, and graphs.

struct Node {  
 int data;  
 Node\* next;  
};  
  
int main() {  
 // Create the first node  
 Node\* firstNode = new Node;  
 firstNode->data = 50;  
 firstNode->next = nullptr;   
  
 return 0;  
}

* **Passing and Returning Multiple Values**

void getValues(int\* a, int\* b) {  
 \*a = 10;  
 \*b = 20;  
}  
  
int main() {  
 int x, y;  
 getValues(&x, &y);  
 cout << "x: " << x << endl;  
 cout << "y: " << y << endl;  
 return 0;  
}  
Output  
x: 10  
y: 20

* **Pointer Arithmetic**: Pointers allow you to perform arithmetic operations on memory addresses, which can be **useful for traversing arrays or linked data structures**.
  + Postfix or Prefix increment (++)
  + Postfix or Prefix decrement (--)
  + An integer may be added to a pointer (+ or +=)
  + An integer may be subtracted from a pointer (- or -=)
  + Difference between two pointers (p1-p2)

int main() {  
 int arr[] = {10, 20, 30, 40, 50};  
 int \*ptr = arr; // Points to the first element of the array  
  
 // Pointer arithmetic examples  
 std::cout << \*ptr << std::endl;  
 std::cout << \*(ptr + 1) << std::endl;  
 std::cout << \*(ptr + 2) << std::endl;  
  
 ptr++; // Move to the next element  
 std::cout << \*ptr << std::endl;  
  
 ptr--; // Move back to the previous element  
 std::cout << \*ptr << std::endl;  
  
 int \*ptr1 = arr; // Points to the first element  
 int \*ptr2 = arr + 2; // Points to the third element  
  
 // Calculate the number of elements between ptr1 and ptr2  
 int numElements = ptr2 - ptr1;  
 std::cout << "Number of elements between ptr1 and ptr2: " << numElements << std::endl;  
  
 return 0;  
}  
Output  
30  
10  
20  
20  
10  
Number of elements between ptr1 and ptr2: 2

### Drawbacks of Pointers

* Segmentation fault can occur due to uninitialized pointer.
* If we forgot to deallocate a memory, then it will lead to a memory leak.

Reference: References serve a different purpose than pointers. They **provide a convenient and safer way to work with values by creating an alias** for an existing variable.



Use case:

* **Function Parameter Passing**: References allow you to pass variables to functions by reference, enabling the function to modify the original variable's value.

void modifyValue(int& num) {  
 num = 20; // Modifies the original variable  
}  
  
int main() {  
 int x = 10;  
 modifyValue(x); // Pass by reference  
 std::cout << x;  
 return 0;  
}  
Output  
20

* **Avoiding Copying**: Passing large objects or structures by reference avoids copying their contents.

void processLargeObject(const LargeObject& obj) {  
 // Use obj without copying it  
 // This function can read the data but cannot modify it  
}

* **Modifying Range-Based Loops**: References are often used in range-based loops to directly access and modify elements in containers.

for (int& element : myVector) {  
 element \*= 2; // Modify elements directly  
}

* **Returning Multiple Values**: Functions can return multiple values through references, providing a convenient way to communicate results.

void findMinMax(const std::vector<int>& n, int& min, int& max) {  
 // Calculate and assign min and max values  
}

**Remember**

Always return static variables as references from functions to prevent the memory from being deallocated at the end of the function block. This approach ensures that you can maintain access to the variable beyond the function's scope, as static variables have a longer lifetime compared to local variables.

int &fun() {  
 static int x = 10;  
 return x;  
}  
  
int main() {  
 int &z = fun(); // Assign the reference returned by fun to z  
 std::cout << fun(); // Print value of x (10) returned by fun  
  
 z = 30; // Modify the value through the reference z  
  
 std::cout << fun(); // Prints the modified value of x (30)  
  
 return 0;  
}  
Output  
10 30

### Const & R-Value Reference

#### Distinction between L-values and R-values

* **L-value**: An l-value refers to an expression that **represents a memory location with a named address**. It can appear on the left side of an assignment and is typically a named variable or object that you can modify or access.
* **R-value**: An R-value refers to an expression that **represents a temporary value or a value that does not have a named memory location**. It is often used on the right side of an assignment and represents a value that can be used in an expression but not modified directly.

**What is Named memory location?** A "named memory location" refers to a place in computer memory that has been given a recognizable name through a variable or identifier.

**int x;** In this code, x is a named memory location that can hold an integer value.

#### Distinction between L-values and R-values References

* **L-value Reference (&)**: An l-value reference is a reference that binds to an l-value, which is typically a named object or variable with a memory address. It allows you to access and modify the value of the object it refers to. L-value references are commonly used when you want to pass an object to a function and allow that function to modify the original object.

int x = 10;  
int &lref = x; // lref : l-value reference to the variable x  
lref = 20; // Modifies the value of x to 20

* **R-value Reference (&&)**: An R-value reference is a reference that binds to an R-value, which includes temporary values or expressions that do not have a named memory address. R-value references are used to enable move semantics and efficient resource management, particularly when working with temporary objects.

int&& rref = 30; // rref : r-value reference to literal value 30

#### Need of Const & R-value references

If you create a normal reference, the right-hand side should be a variable. However, if the right-hand side value is a literal or the result of an expression, you can use a const reference (const L-value reference) or (&&) R-value reference.

“Hello” is a string literal and three is an integer const. So, we must use either const or R-value reference.

int main() {  
 // String cases  
 string &s1 = "Hello"; // Invalid  
 const string &s2 = "World"; // Valid  
 string &&s3 = "Merger"; // Valid  
  
 // Int cases  
 int &i1 = 3; // Invalid  
 const int &i2 = 3; // Valid  
 int &&13 = 3; // Valid  
  
 return 0;  
}  
Invalid Reason  
Attempting to bind a non-const l-value reference to a string and integer literal.

#### Scenarios for Using Const and R-value References

##### Const Reference

A **const** **reference** is useful when you want to pass data to a function, but you do not want the function to modify that data.

When you use a **const string &s** as parameter in the function, you can indeed pass string literals without any errors, but **you will not be able to perform any string manipulation** on the parameter ‘s’ inside the function. This is because ‘s’ is **treated as read-only** due to the const qualifier.

void fun(const string &s) {  
 // Attempting to modify a parameter passed by a const l-value reference is not allowed.  
 // The following line will cause a compilation error.  
 s = "Hello" + s + "!";  
  
 cout << s << endl;  
}  
  
int main() {  
 fun("user");  
 return 0;  
}  
Output  
compilation error

##### R-Value Reference

An **R-value reference** is valuable when you want to efficiently manage **temporary data or enable move semantics within a function**. It is particularly useful for scenarios where you **need to modify or transfer ownership of temporary objects**.

When you use an R-value reference string &&s as a parameter in a function, you have the flexibility to perform string manipulation on the parameter 's' inside the function. This allows you to modify the temporary data efficiently without unnecessary copies. R-value references are particularly effective when dealing with temporary data, as they enable in-place modifications and minimize resource duplication.

void fun(string &&s) {  
 s = "Hello " + s + "!";  
 cout << s << endl;  
}  
  
int main() {  
 fun("user");  
 return 0;  
}  
Output  
Hello user!

## Pointers vs References

|  |  |
| --- | --- |
| **Pointers** | **Reference** |
| Pointer is a variable that **holds the memory address** of another variable. | A reference is an alias for an existing variable. It is another **name** for the same memory location. |
| Pointers **can be reassigned** to point to different memory locations. | References must be initialized when declared, and they **cannot be reassigned** to refer to a different variable. It behaves like constant pointer. |
| They can be **declared without initialization** and may contain garbage values. | They cannot be declared without initialization. |
| Pointer arithmetic is possible, allowing you to navigate through memory based on the size of the data type. | There is no pointer arithmetic with references. |
| Pointers can be made to **point to nullptr** to indicate they are not currently pointing to any valid memory location. | References **cannot be null or uninitialized**. They must always refer to a valid variable. |
| Pointers can be used for implementing data structures like linked lists, trees, and more. | References are useful when you want to avoid copying data. |

## Pointers vs Array

**Recommended Practice for code Readability**

* When using array then do arr [2]
* When using pointer then do \*(ptr+2)

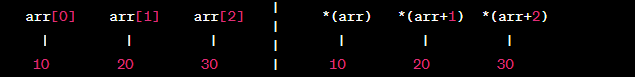
Syntax wise array and pointers looks same, but Array and Pointers are different things.

Cannot Assign to an Array  
int arr1[] = {10, 20, 30};  
int arr2[] = {40, 50, 60};  
int\* ptr = arr1; // Valid: Assigning a pointer to an array  
  
// Invalid: Trying to assign an array to a pointer (as in your original code)  
int\* arr2\_ptr = arr2; // This is not valid C++ syntax and will result in a compilation error.

**Sizeof Operator: Arrays vs. Pointers**

int main() {  
 int arr[] = {10, 20, 30};  
 int\* ptr = arr;   
 cout << sizeof(arr) << " ";  
 cout << sizeof(ptr) << " ";   
 cout << \*(arr + 2) << " ";   
 cout << ptr[2] << " ";  
  
 return 0;  
}  
Output  
20 8 30 30

Compiler job is to see arr[i] and convert to \*(arr+i)



**Remember**

**Why can't I increment an array?** Because array **treated as a constant pointer**. There is a reason for it. Array variable is supposed to point to the first element of the array or first memory instance of the block of the contiguous memory locations in which it is stored. So, if we will have the liberty to change (increment or decrement) the array pointer, it will not point to the first memory location of the block. Thus, it will lose its purpose.

We should write an expression after \* otherwise it will invoke error. This expression will invoke error: **++ptr\***, **ptr\*++** and **ptr++\* (**for both array pointer and simple pointer).

**++ and \* has same precedence so it will execute according to associativity which is Right to Left.**



**Array-to-pointer decay: An array decays into a pointer when it is passed to a function,** always pass the size of the array as a separate parameter to the function and avoid using size of on the array within the function when passing an array as a parameter. Or you can use **std::vector.**

void fun(int arr[]) { // Use this - void fun(int arr[], int size)  
 cout << "Size of arr inside fun: " << sizeof(arr) << " bytes\n";  
 cout << "Size of arr[0] inside fun: " << sizeof(arr[0]) << " bytes\n";  
 // This will not give the correct number of elements.  
 int n = sizeof(arr) / sizeof(arr[0]);   
 cout << "Number of elements in fun: " << n << “\n”;  
}  
  
int main() {  
 int a[] = {10, 20, 30, 40};  
 int n = sizeof(a) / sizeof(a[0]);  
 cout << "Size of a in main: " << sizeof(a) << " bytes\n";  
 cout << "Size of a[0] in main: " << sizeof(a[0]) << " bytes\n";  
 cout << "Number of elements in main: " << n << “\n”;  
  
 fun(a);  
  
 return 0;  
}  
Output  
Size of a in main: 16 bytes  
Size of a[0] in main: 4 bytes  
Number of elements in main: 4  
Size of arr inside fun: 8 bytes  
Size of arr[0] inside fun: 4 bytes  
Number of elements in fun: 2

**Possible increment or decrement operation on Array:**

int main() {  
 int arr[4] = {1, 0, 3, 7};  
   
 // \*(++arr) Error: Cannot increment array pointer  
 cout << \*++arr << endl;   
   
 // ++(\*arr) Works Fine: Output 2  
 cout << ++\*arr << endl;   
   
 // \*(arr++) Error: Cannot increment array pointer  
 cout << \*arr++ << endl;   
   
 return 0;  
}  
Output  
Compilation error

**Possible increment or decrement operation on Pointers:**

int main() {  
 int arr[4] = {1, 0, 3, 7};  
 int \*ptr = arr;  
  
 // \*(++ptr) => \*(ptr=ptr+1)  
 // Pre-Increment: Pointer points to the next element  
 cout << \*++ptr << “ ”;   
  
 // ++(\*ptr) => ++(\*ptr) = 1  
 // Pre-Increment: Increment the value by 1  
 cout << ++\*ptr << “ ”;   
  
 // \*(ptr++) => \*(ptr=ptr+1)  
 // Post-Increment: Pointer points to the next element after expression evaluation  
 cout << \*ptr++ << “ ”;   
  
 return 0;  
}  
Output  
0 1 1

## Function Parameters

### Call-by-Value

Problem with call by value:

1. Changes are not reflected.
2. The whole object is copied.

*// The value is modified, but 'num' remains unchanged*  
void modifyValueByValue(int value) {  
 *// Modifications are made to the copy of 'value', not the original 'num'*  
 value \*= 2;  
}  
  
int main() {  
 *// Initialize num with value 10*  
 int num = 10;  
  
 *// Call the function with the value of num to modify a copy of it*  
 modifyValueByValue(num);  
  
 *// Display the value of num*  
 std::cout << "num: " << num << std::endl; *// Output: num: 10*  
  
 return 0;  
}  
Output  
num: 10

### Call-by-Reference with Pointer Arguments

void modifyValueByPointer(int \*ptr) {  
 \*ptr \*= 2;  
}  
  
int main() {  
 *// Initialize num with value 10 at memory address 0x1000*  
 int num = 10;  
  
 *// Call the function with the address of num to modify its value*  
 modifyValueByPointer(&num); *// 'num' is modified to 20*  
  
 *// Display the modified value of num*  
 std::cout << "num: " << num << std::endl; *// Output: num: 20*  
  
 return 0;  
}  
Output  
num: 20

### Call-by-Reference with Reference Arguments

*// Function to modify a value using a reference*  
void modifyValueByReference(int &ref) {  
 ref \*= 2;  
}  
  
int main() {  
 *// Initialize num with value 10 at memory address 0x1000*  
 int num = 10;  
 int &ref = num;  
  
 modifyValueByReference(ref);   
  
 std::cout << "num: " << num << std::endl;  
 std::cout << "Address of num: " << &num << std::endl;  
 std::cout << "Address of ref: " << &ref << std::endl;  
  
 return 0;  
}  
Output  
num: 20  
Address of num: 0x1000  
Address of ref: 0x1000

TODO: Const & R Value References

If the objective is performance optimization without modifying the reference variable within the function, and for enhanced readability, it is recommended to use a const when passing. This approach ensures memory preservation and improves code clarity.

*// Function to print a string by const reference*  
void printString(const std::string& str) {  
 std::cout << str << std::endl;  
}  
  
int main() {  
 std::string message = "Hello, const reference!";  
   
 *// Passing the string by const reference to the printString function*  
 printString(message);  
  
 return 0; *// This return statement should be inside the main function*  
}  
Output  
Hello, const reference!

## Array Name as Pointers

An array name contains the address of first element of the array which acts like constant pointer. if we have an array named val then val and &val [0] can be used interchangeably.

int main() {  
 int val[3] = { 5, 10, 15 };  
 int\* ptr;

ptr = val; *// Assign the address of val[0] to ptr*  
 *// We can also use ptr = &val[0]; (both are the same)*  
  
 std::cout << "Elements of the array are: ";  
 std::cout << ptr[0] << " " << ptr[1] << " " << ptr[2];  
  
 return 0;  
}  
Output  
Elements of the array are: 5 10 15



## Pointers and String literals

### String Literals as Pointers

if you declare a pointer as **char\* str = "Hello";** without the const qualifier, you might assume that it should be writable. However, this is a **common misconception** when dealing with string literals in C++.

Even though you declare the pointer without const, the actual string literal "Hello" remains in read-only memory.

char \* str = “Hello”;

Avoid confusion and potential issues, it is a good practice to declare the pointer to a string literal as **const char\* str = "Hello";** or use a **modifiable character array** or the **std::string class** for manipulation.

const char \* str = “Hello”;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| str [0] | str [1] | str [2] | str [3] | str [4] | str [5] |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | ‘H’ | ‘e’ | ‘l’ | ‘l’ | ‘o’ | ‘\0’ | | | | | | |
| 1800 | 1801 | 1802 | 1803 | 1804 | 1805 |

### Modifiable Strings

If you want to create a modifiable string, you need to allocate memory for it explicitly and then copy the content of the string literal into that memory.

char modifiableStr[] = "Hello";  
modifiableStr[0] = 'h'; *// This is allowed because we allocated modifiable*

### C++ string Class

it is often recommended to use the std::string class for string manipulation, as it provides a more convenient and safer way to work with strings. It manages memory automatically and allows easy string manipulation without the concerns of memory management and null termination.

int main() {  
 *// Declare and initialize a string using the string class*  
 std::string cppString = "Hello";   
 cppString[0] = 'h';  
  
 *// Append more text to the string*  
 cppString += " World!";  
  
 *// Print the string and its length*  
 std::cout << "String: " << cppString << std::endl;  
  
 return 0;  
}  
Output  
String: hello World!

## Pointers to pointers

**Pointers to pointers** are a concept in C++ where you use a pointer to store the memory address of another pointer.

int main() {  
 int x = 42;  
 int\* ptrX = &x; *// Pointer to int*  
 int\*\* ptrToPtrX = &ptrX; *// Pointer to pointer to int*  
  
 *// Using pointer to pointer to access the value of x*  
 std::cout << "Value of x: " << \*\*ptrToPtrX << std::endl;  
  
 return 0;  
}  
Output  
Value of x: 42

## Void Pointers

**void\*** is a special type of pointer in C++ that **does not have a specific data type associated with it**. It can **point to memory locations of any type**. This flexibility comes with challenges, as you need to be careful when using void pointers to ensure proper type casting when dereferencing or performing pointer arithmetic. Here is an example to illustrate void pointers:

int main() {  
 int intValue = 42;  
 float floatValue = 3.14;  
 char charValue = 'A';  
 void\* ptr; *// Declare a void pointer*  
  
 ptr = &intValue; *// Point to an int*  
 cout << "Value at int pointer:"<< \*(static\_cast<int\*>(ptr));   
 cout << endl;   
  
 ptr = &floatValue; *// Point to a float*  
 cout << "Value at float pointer:"<< \*(static\_cast<float\*>(ptr));   
 cout << endl;   
  
 ptr = &charValue; *// Point to a char*  
 cout << "Value at char pointer:" << \*(static\_cast<char\*>(ptr));   
  
 return 0;  
}  
Output  
Value at int pointer: 42  
Value at float pointer: 3.14  
Value at char pointer: A

## NULL Pointer

If a pointer is not explicitly initialized, it will contain a garbage value, and attempting to dereference such a pointer can lead to undefined behaviour, including a segmentation fault or access violation. It's crucial to always initialize pointers before using them to ensure they point to valid memory locations. To avoid this, it's a recommended practice to initialize pointers before using them, like this:

A black background with white lines and colorful text

Description automatically generated

NULL is a special value of pointer. It is defined as preprocessor macro.

* #define NULL 0 (In C++)
* #define NULL ((void \*)0) (In C)

Dereferencing a NULL pointer can indeed lead to undefined behaviour, crashes, segmentation faults, or memory access violations. **It's a crucial practice to check if a pointer is NULL (or nullptr) before attempting to access the memory.**

When you create a pointer and do not immediately have a valid value to assign to it, it is a good practice to initialize it with nullptr (or in older code, you can use NULL). Initializing a pointer with nullptr or NULL explicitly sets it to a known state, indicating that it does not currently point to a valid memory location. This can help prevent accidental dereferencing of **uninitialized or wild pointers**, which can lead to undefined behaviour and crashes.

**Application of NULL**

* For pointer with no memory address
* Function use NULL to return invalid output.
* In Data Structure like Linked List, tree etc.

**Remember**

* A NULL pointer converts to bool value false, and all other value convert to bool value true.
* NULL can be used for any type. **double \*ptr = NULL, char \*ptr = NULL, int \*ptr = NULL**

### nullptr

Added in C++ 11 as replacement of NULL.

#### Issue with NULL Pointer

* int x = NULL allowed but int x = nullptr is not allowed

NULL is a preprocessor macro, Type of nullptr is nullptr\_t, so integer conversion does not happen.

int main() {  
 int x = 0; *// Compile without error on some older compilers*  
 std::cout << "x: " << x << std::endl;  
 return 0;  
}

* In C++, NULL is a macro defined to have zero value, it may mean **int as well as int\*** causing ambiguity.

*// Function overloads*  
void fun(int x) { std::cout << "Integer Call: " << x; }  
  
void fun(int\* x) { std::cout << "Pointer Call: " << x; }  
  
int main() {  
 int\* p = NULL; *// Define a pointer p and initialize with NULL*  
  
 fun(0); *// Call the fun(int x) overload with int argument 0*  
 fun(p); *// Call the fun(int\* x) overload with pointer p*  
 fun(NULL); *// This line leads to ambiguity due to NULL (0) being valid for both overloads*  
 *// To fix this ambiguity, you can explicitly cast NULL or use nullptr*  
 return 0;  
}  
Output  
main.cpp:13:8: error: call of overloaded ‘fun(NULL)’ is ambiguous  
fun(NULL); *// This line leads to ambiguity due to NULL (0) being valid for both overloads*

Fix this,



or

