Arrays, String, Pointers & Reference

INDEX

Table of Contents

[Arrays in C++ 3](#_Toc143965805)

[Array Declaration & Initialization 3](#_Toc143965806)

[2D Array Declaration & Initialization 4](#_Toc143965807)

[3D Array Declaration & Initialization 5](#_Toc143965808)

[Memory Layout 6](#_Toc143965809)

[Sizeof Array 6](#_Toc143965810)

[Array Traversal in C++ 7](#_Toc143965811)

[Different Type of Array 10](#_Toc143965812)

[Access elements in a 2D array 10](#_Toc143965813)

[Passing 2D arrays as arguments in C++ 11](#_Toc143965814)

[Strings in C++ 14](#_Toc143965815)

[C-style (character arrays and literals) 14](#_Toc143965816)

[String Class in C++ STL 15](#_Toc143965817)

[String manipulations 16](#_Toc143965818)

[Pointers & References in C++ 22](#_Toc143965819)

[Pointers 22](#_Toc143965820)

[Address and Dereference Operator 22](#_Toc143965821)

[Drawbacks of Pointers 22](#_Toc143965822)

[Use case 22](#_Toc143965823)

[Reference 27](#_Toc143965824)

[Use case 27](#_Toc143965825)

[Const & R-Value Reference 29](#_Toc143965826)

[Pointers vs References 32](#_Toc143965827)

[Pointers vs Array 33](#_Toc143965828)

[Function Parameters 35](#_Toc143965829)

[Call-by-Value 35](#_Toc143965830)

[Call-by-Reference with Pointer Arguments 36](#_Toc143965831)

[Call-by-Reference with Reference Arguments 37](#_Toc143965832)

[Array Name as Pointers 38](#_Toc143965833)

[Pointers and String literals 39](#_Toc143965834)

[String Literals as Pointers 39](#_Toc143965835)

[Modifiable Strings 39](#_Toc143965836)

[C++ string Class 40](#_Toc143965837)

[Pointers to pointers 41](#_Toc143965838)

[Void Pointers 42](#_Toc143965839)

[NULL Pointer 43](#_Toc143965840)

[Issue with NULL Pointer 44](#_Toc143965841)

# 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

A screenshot of a computer code

Description automatically generated

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



## 2D Array Declaration & Initialization

A 2D array defined as follows can be viewed as a table of three rows and four columns. There are various ways in which a 2D Array can be initialized.

A black screen with numbers and letters

Description automatically generated

Each set of inner braces represents one row





A black background with white text

Description automatically generated

A black screen with white text

Description automatically generated

## 3D Array Declaration & Initialization

A black screen with numbers and symbols

Description automatically generated

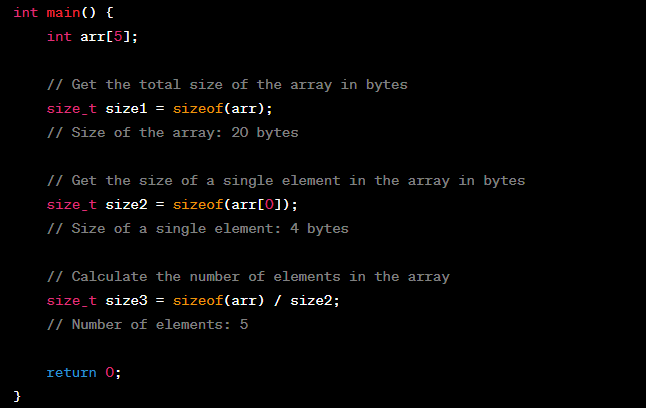
## Memory Layout

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



## Array Traversal in C++

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

1. **Using for loop**

A computer code on a black background

Description automatically generated

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



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.

A computer screen with white text

Description automatically generated

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.

A computer screen shot of a black screen

Description automatically generated

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

A computer screen shot of a black screen

Description automatically generated

**Modifications During Array Traversal**:

A computer code on a black background

Description automatically generated

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

A screenshot of a computer program

Description automatically generated

## 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.

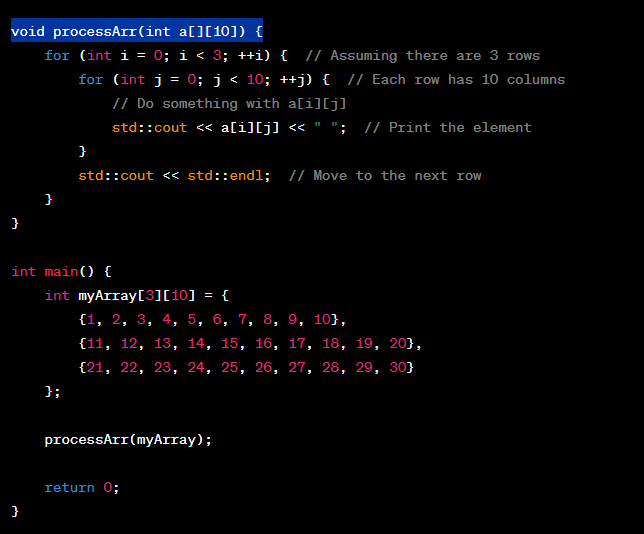
Here's a breakdown of how the formula works:

* 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++

Specify the size of columns of 2D array



OP



Pass array containing pointers

A computer screen shot of a program code

Description automatically generated

OP



Pass a pointer to a pointer

A computer screen shot of a program code

Description automatically generated

OP



# Strings in C++

There are mainly 2 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** |

Available functions in the std::string class which can be used for string manipulations.

**String Iteration**

****

OP



**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**.
* When appending characters to a string, if the size exceeds the capacity, the string might need to be reallocated to accommodate the new characters.



OP

A black background with a white and blue light

Description automatically generated with medium confidence

**String Concatenation and Append**



OP

A black screen with white text

Description automatically generated

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



OP

A black background with white text

Description automatically generated

**find and rfind**:



OP



**substr**:

**A screen shot of a computer code

Description automatically generated**

OP



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

****

OP

A black background with white text

Description automatically generated

**compare**:



OP



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

### 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.



### 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.

Use case:

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

**A black background with white text

Description automatically generated**

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

**A screen shot of a computer

Description automatically generated**

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

**A computer screen shot of a black screen

Description automatically generated**

* **Array Manipulation**

A computer code with white text

Description automatically generated

OP



* **Working with Hardware**

****

OP



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

**A black background with white text

Description automatically generated**

* **Passing and Returning Multiple Values**

**A computer screen with white text and colorful text

Description automatically generated**

OP



* **Pointer Arithmetic**: Pointers allow you to perform arithmetic operations on memory addresses, which can be useful for traversing arrays or linked data structures.
  + 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)

****

OP

A black background with white text

Description automatically generated

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.



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

A black background with white text

Description automatically generated

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

A black background with white text

Description automatically generated

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

A black background with white text

Description automatically generated

**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.

**A computer screen with text on it

Description automatically generated**

OP



### 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 doesn't have a named memory location**. It's 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.

A black background with white text

Description automatically generated

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



#### 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 3 is an integer const. So, we must use either const or R-value reference.A computer screen with text and numbers

Description automatically generated

OP: Error will be like



#### 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 don't 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 won't 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.

A black screen with white text

Description automatically generated

##### R-Value Reference

An **R-value reference** is valuable when you want to efficiently handle **temporary data or enable move semantics within a function**. It's 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.

A black screen with green lights

Description automatically generated

OP



## 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's essentially **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

**Remember**

**Why can't I increment an array?** It's because array is **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 won't 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 or simple pointer).

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



**Possible increment or decrement operation on Array**



OP

A black background with a black square

Description automatically generated with medium confidence

**Possible increment or decrement operation on Pointers**

A computer screen shot of a black background

Description automatically generated

OP

A black background with a black square

Description automatically generated with medium confidence

**Remember**

## Function Parameters

### Call-by-Value

Problem with call by value:

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

A computer screen shot of a code

Description automatically generated

### Call-by-Reference with Pointer Arguments

A computer screen shot of a program code

Description automatically generated

### Call-by-Reference with Reference Arguments

A computer screen shot of a program code

Description automatically generated

TODO: Const & R Value References

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

Description automatically generated

## 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.

A computer screen with white text

Description automatically generated



OP



## 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.



To avoid confusion and potential issues, it's 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.



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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.



### C++ string Class

it's 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.

A computer screen shot of a black background

Description automatically generated

OP



## Pointers to pointers

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



OP



## Void Pointers

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



OP

A black background with a black square

Description automatically generated with medium confidence

## NULL Pointer

A NULL pointer, often represented as **nullptr**, is a pointer that **doesn't point to any valid memory address**. Dereferencing a NULL pointer usually leads to **undefined behaviour, crashes, or memory access violations**.



OP



As a side note, **nullptr is** **convertible to bool**.

A computer screen with white text and green text

Description automatically generated

### Issue with NULL Pointer

In C++, NULL is a macro defined to have 0 value, it may mean **int as well as int\*** causing ambiguity.

OP

A black background with red and white text

Description automatically generated

To fix this,



or

