# **No.1 Introduction to Pointers and Arrays**

**OBJECTIVES OF THE LAB**

‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐

In this lab, we will learn about the relation between arrays and pointers, and use them efficiently in our program.

* *Pointers and dynamic memory allocation*
* *Arrays*
* *Connection between arrays and pointers*

‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐‐

## **1.1 Pre Lab**

1.1.1 Pointers

A C pointer is a variable (memory storage) large enough to accommodate the value of a memory address. C pointer simply holds an integer (not the actual memory object it may be pointing to). For pointers to be useful there need to be some other memory objects that the pointers can point to.

**Example:**

1) int i = 7; /\* simple integer variable \*/

2) int \*ptr = NULL; /\* simply holds the value of NULL \*/

3) ptr = i; /\* bad for your health, but legal \*/

4) ptr = &i; /\* takes address of i \*/

5) \*ptr = 8; /\* dereference the pointer to access i i.e. the value referenced to by ptr is equal to 8\* means assign 8 to the address of this pointer ie at i/

**1.1.2 Initializing Pointers**

Pointers can be initialized to point to specific locations at the very moment they are defined:

|  |  |  |
| --- | --- | --- |
|  | int myvar;  int \* myptr = &myvar; |  |

The resulting state of variables after this code is the same as after:

|  |  |
| --- | --- |
| int myvar;  int \* myptr;  myptr = &myvar; |  |

When pointers are initialized, what is initialized is the address they point to (i.e., myptr), never the value being pointed (i.e., \*myptr). Therefore, the code above shall not be confused with:

|  |  |
| --- | --- |
| int myvar;  int \* myptr;  \*myptr = &myvar; |  |

Which anyway would not make much sense (and is not valid code).

The asterisk (\*) in the pointer declaration (line 2) only indicates that it is a pointer, it is not the dereference operator (as in line 3). Both things just happen to use the same sign: \*. As always, spaces are not relevant, and never change the meaning of an expression.

Pointers can be initialized either to the address of a variable (such as in the case above), or to the value of another pointer (or array):

|  |
| --- |
| int myvar;  int \*myptr = &myvar;  int \*bar = myptr; |

**Example:**

|  |  |
| --- | --- |
| // my first pointer  #include <iostream>  using namespace std;  int main ()  {  int first, second;  int \*mytr;  myptr = &first;  \*myptr = 10;  myptr = &second;  \*myptr = 20;  cout << "firstvalue is " << first << '\n';  cout << "secondvalue is " << second << '\n';  return 0;  } | firstvalue is 10  secondvalue is 20 |

**1.1.3 Dynamic Memory Allocation (malloc function/operator new, delete)**

In most programs all memory needs were determined before program execution by defining the variables needed. But there may be cases where the memory needs of a program can only be determined during runtime. For example, when the memory needed depends on user input. On these cases, programs need to dynamically allocate memory, for which the C++ language integrates the operators new and delete. The syntax is:

pointer = new type

pointer = new type [number\_of\_elements]

delete pointer;

delete[] pointer;

C++ integrates the operators new and delete for allocating dynamic memory. But these were not available in the C language; instead, it used a library solution, with the functions [malloc](http://www.cplusplus.com/malloc), [calloc](http://www.cplusplus.com/calloc), [realloc](http://www.cplusplus.com/realloc) and [free](http://www.cplusplus.com/free), defined in the header [<cstdlib>](http://www.cplusplus.com/%3Ccstdlib%3E) (known as <stdlib.h> in C). The functions are also available in C++ and can also be used to allocate and deallocate dynamic memory.  
  
Note, though, that the memory blocks allocated by these functions are not necessarily compatible with those returned by new, so they should not be mixed; each one should be handled with its own set of functions or operators.

* The size of the problem often cannot be determined at “compile time”.
* Dynamic memory allocation is to allocate memory at “run time”.
* Dynamically allocated memory must be referred to by pointers.

**Example:**

int array\_size=100;

double a[100], \*b, \*c;  
b = (double \*) malloc(array\_size \* sizeof(double)); /\* allocation in C\*/  
c = new double[array\_size]; /\* allocation in C++ \*/

**1.1.4 Pointers and arrays**

The concept of arrays is related to that of pointers. In fact, arrays work very much like pointers to their first elements, and, actually, an array can always be implicitly converted to the pointer of the proper type. For example, consider these two declarations:

int a[100], \*ptr\_a; /\*a is an array of 100 elements and ptr\_a is a pointer\*/  
ptr\_a = &(a[0]); /\* or ptr\_a = a; a pointer pointing to the first location of array\*/  
ptr\_a++; /\*or ptr\_a += 1; \*/

// now ptr\_a points to the next integer, a[1];

After that, ***ptr\_a***and ***a*** would be equivalent and would have very similar properties. The main difference being that ***ptr\_a*** can be assigned ***a*** different address, whereas a can never be assigned anything, and will always represent the same block of 100 elements of type int.

**Example:**

|  |  |
| --- | --- |
| // Pointers and Array  #include <iostream>  using namespace std;  int main ()  {  int MyArray[10];  int \* ptr;  ptr = MyArray;  \*ptr = 100;  ptr++;  \*ptr = 200;  ptr = &MyArray[2];  \*ptr = 300;  ptr = MyArray + 3;  \*ptr = 400;  ptr = MyArray; \*(ptr+4) = 500;  for (int n=0; n<5; n++)  cout << MyArray[n] << ", ";  return 0;  } | 100, 200, 300, 400, 500, |

## **1.2 Post Lab**

## **Task 01**

Write a program that reads numbers from the user in to an array of type “float”, average them and print the result.

Sample OUTPUT of the program

Enter length of the array: 5

Enter the elements of the array: 11 22 33 44 55

Average of the array: 33

Note: Add all the elements of array and divide by total number of elements. Use pointers.

## **Task 02**

Write a function that takes an int array and array’s size as argument and return maximum value of array elements.

Prototype of function:

int maxValue(int arr[], int Size)

Call to the function:

int ReturnedValue = maxValue (passArrayHere,passSizeHere)

Note: Store first element of array in a temporary variable and then traverse the array and if you find any value greater than temp update temp. After traversal return temp which contains the maximum value.

## **Task 03**

Write a function that takes an int array and the array's size as arguments. It should create a new array that is twice the size of the argument array. The function should copy the contents of the argument array to the new array, and initialize the unused elements of new array with -1. The function should return a pointer to the new array.

Note: Use malloc or new function from standard library for dynamic allocation of the array at runtime.

## **Task 04**

Write a function that takes two int arrays and the arrays' sizes as arguments. It should create a new array big enough to store both arrays. Then it should copy the contents of the first array to the new array, and then copy the contents of the second array to the new array in the remaining elements, and return a pointer to the new array.

Prototype:

int\* mergeArray (int arrA[], int sizeA, int arrB[], int SizeB)

Note: You can also use pointer to hold an array argument. So the above function can be replaced by the following function

Int \* mergeArray(int\* arrA, int sizeA, int\* arrB, int sizeB)

## **1.3 References**

1. **Introduction to Algorithms by *CLRS (3rd ed.)***
2. **Cplusplus.com**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_