|  |
| --- |
| **NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES**  **CS 201–DATA STRUCTURES LAB**  **Lab Session 2** |
| **Instructors:** Mr. Faizan Yousuf, Ms. Safia, Ms. Maham Mobin |

## Outline

* Pointers
* Pointers to array
* Double Pointers
* Dynamic Memory Management
* Task

# **Pointers:**

Pointer is a variable whose value is a memory address. Normally, a variable directly contains a specific value. A pointer contains the memory address of a variable that, in turn, contains a specific value. In this sense, a variable name directly references a value, and a pointer indirectly references a value.

# **REFERENCE OPERATOR (&):**

The address that locates a variable within memory is what we call a reference to that variable.

ted = &y;

Assume that y is placed during runtime in the memory address 1776.

y = 25;

fred = y;

ted = &y;

The values contained in each variable after the execution of this, are shown in the following diagram:



# **DE-REFERENCE OPERATOR ( \* )**

The asterisk (\*) acts as a dereference operator and that can be translated to "value pointed by".

beth = \*ted;

We could read as: "beth equal to value pointed by ted", beth would take the value 25, since ted is 1776, and the value pointed by 1776 is 25.



## POINTER TYPE DECLARATION:

SYNTAX: type \* variable;

EXAMPLE: int \*ptr, char \* character, float \* greatvalue;

EXPLANATION: The value of the pointer variable ptr is a memory address. A data item whose address is stored in this variable must be of the specified type.



# **POINTERS AND ARRAYS**

The identifier of an array is equivalent to the address of its first element, as a pointer is equivalent to the address of the first element that it points to.

EXAMPLE: Assume the following declaration:

int numbers [20];

int \* p;

The following assignment operation would be valid:

p = numbers;



# **POINTERS TO POINTERS (DOUBLE POINTER)**

C++ allows the use of pointers that point to pointers, that these, in its turn, point to data (or even to other

pointers). In order to do that, we only need to add an asterisk (\*) for each level of reference in their declarations:

EXAMPLE: char a;

char \* b;

char \*\* c;

a = 'z';

b = &a;

c = &b;

This, supposing the randomly chosen memory locations for each variable of 7230, 8092 and 10502, could be represented as:



• c has type char\*\* and a value of 8092

• \*c has type char\* and a value of 7230

• \*\*c has type char and a value of 'z'

# **VOID POINTER:**

Void pointers are pointers that can point to any data type. One limitation of void pointers is that they cannot be directly de-referenced. Example:

****

# **NULL POINTER:**

A null pointer is a regular pointer of any pointer type which has a special value that indicates that it is not pointingto any valid reference or memory address.

EXAMPLE: int \* p;

p = 0; // p has a null pointer value

# **Dynamic Memory Management:**

A good understanding of how dynamic memory really works in C++ is essential to becoming a good C++ programmer. Memory in your C++ program is divided into two parts −

* **The stack** − All variables declared inside the function will take up memory from the stack.
* **The heap** − this is unused memory of the program and can be used to allocate the memory dynamically when program runs.

Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.

You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.

If you are not in need of dynamically allocated memory anymore, you can use **delete** operator, which de-allocates memory that was previously allocated by new operator.

# **new and delete Operators**

There is following generic syntax to use **new** operator to allocate memory dynamically for any data-type.

new data-type;

Here, **data-type** could be any built-in data type including an array or any user defined data types include class or structure. Let us start with built-in data types. For example we can define a pointer to type double and then request that the memory be allocated at execution time. We can do this using the **new** operator with the following statements −

double\* pvalue = NULL; // Pointer initialized with null

pvalue = new double; // Request memory for the variable

The memory may not have been allocated successfully, if the free store had been used up. So it is good practice to check if new operator is returning NULL pointer and take appropriate action as below −

double\* pvalue = NULL;

if( !(pvalue = new double )) {

cout << "Error: out of memory." <<endl;

exit(1);

}

The **malloc()** function from C, still exists in C++, but it is recommended to avoid using malloc() function. The main advantage of new over malloc() is that new doesn't just allocate memory, it constructs objects which is prime purpose of C++.

At any point, when you feel a variable that has been dynamically allocated is not anymore required, you can free up the memory that it occupies in the free store with the ‘delete’ operator as follows −

delete pvalue; // Release memory pointed to by pvalue

Let us put above concepts and form the following example to show how ‘new’ and ‘delete’ work −

#include <iostream>

using namespace std;

int main () {

double\* pvalue = NULL; // Pointer initialized with null

pvalue = new double; // Request memory for the variable

\*pvalue = 29494.99; // Store value at allocated address

cout << "Value of pvalue : " << \*pvalue << endl;

delete pvalue; // free up the memory.

return 0;

}

If we compile and run above code, this would produce the following result −

Value of pvalue : 29495

# **Dynamic Memory Allocation for Arrays**

Consider you want to allocate memory for an array of characters, i.e., string of 20 characters. Using the same syntax what we have used above we can allocate memory dynamically as shown below.

char\* pvalue = NULL; // Pointer initialized with null

pvalue = new char[20]; // Request memory for the variable

To remove the array that we have just created the statement would look like this −

delete [] pvalue; // Delete array pointed to by pvalue

Following the similar generic syntax of new operator, you can allocate for a multi-dimensional array as follows −

double\*\* pvalue = NULL; // Pointer initialized with null

pvalue = new double [3][4]; // Allocate memory for a 3x4 array

However, the syntax to release the memory for multi-dimensional array will still remain same as above −

delete [] pvalue; // Delete array pointed to by pvalue

# **Dynamic Memory Allocation for Objects**

Objects are no different from simple data types. For example, consider the following code where we are going to use an array of objects to clarify the concept −

|  |
| --- |
| #include <iostream>  using namespace std;  class Box {  public:  Box() {  cout << "Constructor called!" <<endl;  }  ~Box() {  cout << "Destructor called!" <<endl;  }  };  int main() {  Box\* myBoxArray = new Box[4];  delete [] myBoxArray; // Delete array  return 0;  } |

If you were to allocate an array of four Box objects, the Simple constructor would be called four times and similarly while deleting these objects, destructor will also be called same number of times.

If we compile and run above code, this would produce the following result −

Constructor called!

Constructor called!

Constructor called!

Constructor called!

Destructor called!

Destructor called!

Destructor called!

Destructor called!

# **LAB TASK:**

Question 1:

Write a program that will read 10 integers from the keyboard and place them in an array.

* The program then will sort the array into ascending and descending order and print the sorted list.
* The program must not change the original array or create any other integer arrays.

Hint: It requires two pointer arrays. Make sure that your array should not go out of bound.