Week 3: Lecture 5

Array

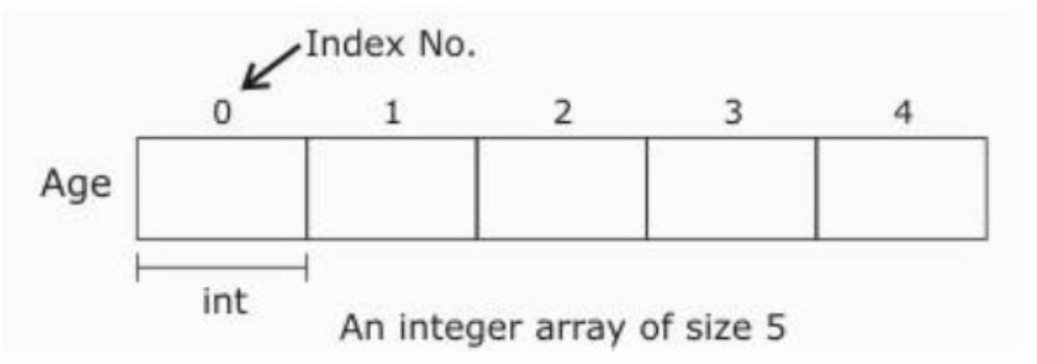
An array is a collection of data elements of same data type. It is described by a single name and each element of an array is referenced by using array name and its subscript no.

Declaration of Array

Type arrayName[numberOfElements];

For example,

```
int Age[5] ;
float cost[30];
```



Initialization of One Dimensional Array

An array can be initialized along with declaration. For array initialization it is required to place the elements separated by commas enclosed within braces.

```
int A[5] = {11,2,23,4,15};
```

It is possible to leave the array size open. The compiler will count the array size.

```
int B[] = {6,7,8,9,15,12};
```

Referring to Array Elements

In any point of a program in which an array is visible, we can access the value of any of its elements individually as if it was a normal variable, thus being able to both read and modify its value.

The format is as simple as:

name[index]

Examples:

```
cout << age[4]; //print an array element
age[4] = 55; // assign value to an array element
cin >> age[4]; //input element 4
```

Using Loop to input an Array from user

```
#include <iostream>
using namespace std;
int main ()
{

int age [10], i ;
      for (i = 0 ; i < 10; i++)
      {
      cin >> age[i];
      }

return 0;
}
```

Function to Search for an element from A by Linear Search

```
#include <iostream>
using namespace std;
int main ()
{

int A [10], i ,data;
for (i = 0 ; i < 5; i++)
      {
      cin >> A[i];
      }

cout << "Data: ";
cin>> data;

for(int i = 0; i < n; i++)
      {
      if(A[i] == data)
            {
            cout << "Data Found at : " << i;
            return;
            }
      }
cout << "Data Not Found in the array" << endl;
return 0;
}
```

Two-Dimensional Arrays

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x,y, you would write something as follows −

```
type arrayName [ x ][ y ];
```

Where **type** can be any valid C++ data type and **arrayName** will be a valid C++ identifier.

A two-dimensional array can be think as a table, which will have x number of rows and y number of columns. A 2-dimensional array **a**, which contains three rows and four columns can be shown as below −

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in array a is identified by an element name of the form **a[i][j]**, where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

Initializing Two-Dimensional Arrays

Multidimensioned arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row have 4 columns.

```
int a[3][4] = {
   {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
   {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
   {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example −

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements

An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example −

```
int val = a[2][3];
```

The above statement will take 4th element from the 3rd row of the array.

```
#include <iostream>
using namespace std;
 int main () {
  // an array with 5 rows and 2 columns.
  int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};

  // output each array element's value
  for ( int i = 0; i < 5; i++ )
    for ( int j = 0; j < 2; j++ ) {

      cout << "a[" << i << "][" << j << "]: ";
      cout << a[i][j]<< endl;
    }
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

Week 3: Lecture 6

Pointer

Pointer

Accessing address of a variable

Computer's memory is organized as a linear collection of bytes. Every byte in the computer's memory has an address. Each variable in program is stored at a unique address. We can use address operator & to get address of a variable:

```
int num = 23;
cout << &num; // prints address in hexadecimal
```

POINTER

A pointer is a variable that holds a memory address, usually the location of another variable in memory.

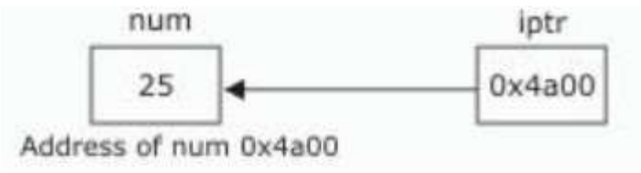
Defining a Pointer Variable

```
int *iptr;
```
iptr can hold the address of an int

Pointer Variables Assignment:

```
int num = 25;
int *iptr;
iptr = &num;
```

Memory layout



To access num using iptr and indirection operator *

```
cout << iptr; // prints 0x4a00
cout << *itptr; // prints 25
```

Similary, following declaration shows:

```
char *cptr;
float *fptr;
```

cptr is a pointer to character and fptr is a pointer to float value.

Pointer Arithmetic

Some arithmetic operators can be used with pointers:

- Increment and decrement operators ++, --
- Integers can be added to or subtracted from pointers using the operators +, -, +=, and -=

Each time a pointer is incremented by 1, it points to the memory location of the next element of its base type.

If “p” is a character pointer then “p++” will increment “p” by 1 byte.
If “p” were an integer pointer its value on “p++” would be incremented by 4 bytes.

Pointers and Arrays

Array name is base address of array

```
int vals[] = {4, 7, 11};

cout << vals; // displays 0x4a00
cout << vals[0]; // displays 4
```

Lets takes an example:

```
int arr[]={4,7,11};
int *ptr = arr;
```

What is ptr + 1?

It means (address in ptr) + (1 * size of an int)

```
cout << *(ptr+1); // displays 7
cout << *(ptr+2); // displays 11
```

Array Access

Array notation arr[i] is equivalent to the pointer notation *(arr + i)
Assume the variable definitions

```
int arr[]={4,7,11};
int *ptr = arr;
```

Examples of use of ++ and –

```
ptr++; // points at 7
ptr--; // now points at 4
```

Character Pointers and Strings

Initialize to a character string.

```
char* a = “Hello”;
```

a is pointer to the memory location where „H. is stored. Here “a” can be viewed as a character array of size 6, the only difference being that a can be reassigned another memory location.

char* a = “Hello”;

a gives address of ‘H’

*a gives ‘H’

a[0] gives ‘H’

a++ gives address of ‘e’

*a++ gives ‘e’

Pointers as Function Parameters

A pointer can be a parameter. It works like a reference parameter to allow change to argument from within function

```
#include<iostream>
using namespace std;

void swap(int *, int *);
int main()
{
     int a=10,b=20;
     swap(&a, &b);
     cout<<a<<" "<<b;
return 0;
}
void swap(int *x, int *y)
{
     int t;
     t=*x;
     *x=*y;
     *y=t;
}
```

Output:
20 10

Pointers to Constants and Constant Pointers
Pointer to a constant: cannot change the value that is pointed at
Constant pointer: address in pointer cannot change once pointer is initialized

Dynamic Memory Allocation

We can allocate storage for a variable while program is running by using new operator

To allocate memory of type integer

int *iptr=new int;

To allocate array

double *dptr = new double[25];

To allocate dynamic structure variables or objects

Student sptr = new Student; //Student is tag name of structure

Releasing Dynamic Memory

Use delete to free dynamic memory
delete iptr;

To free dynamic array memory
delete [] dptr;

To free dynamic structure
delete Student;

Week 4: Lecture 7

Function

A function is a block of code that performs a specific task.

Dividing a complex problem into smaller groups makes our program easy to understand and reusable.

Benefits of Using User-Defined Functions

- Functions make the code reusable. We can declare them once and use them multiple times.
- Functions make the program easier as each small task is divided into a function.
- Functions increase readability.

There are two types of function:

1. **Standard Library Functions:** Predefined in C++
2. **User-defined Function:** Created by users

C++ Library Functions

Library functions are the built-in functions in C++ programming. Programmers can use library functions by invoking the functions directly; they don't need to write the functions themselves. Some common library functions in C++ are `sqrt()`, `abs()`, `isdigit()`, etc.

In order to use library functions, we usually need to include the header file in which these library functions are defined. For instance, in order to use mathematical functions such as `sqrt()` and `abs()`, we need to include the header file `cmath`.

Example :

```
#include <iostream>
#include <cmath>
using namespace std;

int main() {
   double number, squareRoot;

   number = 25.0;

   // sqrt() is a library function to calculate the square root
   squareRoot = sqrt(number);

   cout << "Square root of " << number << " = " << squareRoot;

   return 0;
}
```

C++ User-defined Function

C++ allows the programmer to define their own function. A user-defined function groups code to perform a specific task and that group of code is given a name (identifier).

When the function is called from any part of the program, it all executes the codes defined in the body of the function.

User Define Function Parts:

1. **Definition**
2. **Call**
3. **Declaration of prototype**

1. Function definition

The general syntax for user-defined functions (or simply functions) is as given below:

```
return_type functionName(param1,param2,….param3)
 {
Function body;
 }
```

So as shown above, each function has:

- **Return type:** It is the value that the functions return to the calling function after performing a specific task.
- **functionName** : Identifier used to name a function.
- **Parameter List:** Denoted by param1, param2,…paramn in the above syntax. These are the arguments that are passed to the function when a function call is made. The parameter list is optional i.e. we can have functions that have no parameters.
- **Function body:** A group of statements that carry out a specific task.

Here's an example of a function definition without parameter.

```
// function definition
void greet() {
   cout << "Hello World";
}
```

Here,

- the name of the function is `greet()`
- the return type of the function is `void`
- the empty parentheses mean it doesn't have any parameters
- the function body is written inside `{}`

2. Calling a Function

Here's how we can call the above greet() function.

```
int main() {

   // calling a function
   greet();

}
```

Example :

```
#include <iostream>
using namespace std;

// declaring a function
void greet() {
   cout << "Hello there!";
}

int main() {

   // calling the function
   greet();

   return 0;
}
```

Output

```
Hello there!
```

3. Function Declaration or prototype

A function declaration tells the compiler about the return type of function, the number of parameters used by the function and its data types. Including the names of the parameters in the function, the declaration is optional. The function declaration is also called as a function prototype.

We have given some examples of the function declaration below for your reference.

```
int sum(int, int);
```

Above declaration is of a function ‘sum’ that takes two integers as parameters and returns an integer value.

```
void swap(int, int);
```

This means that the swap function takes two parameters of type int and does not return any value and hence the return type is void.

```
void display();
```

The function display does not take any parameters and also does not return any type.

In C++, the code of function declaration should be before the function call. However, if we want to define a function after the function call, we need to use the function prototype.

Example 1:

```
// function prototype
void add();

int main() {
   // calling the function before declaration.
   add();
   return 0;
}
// function definition
void add() {
            int a=10,b=20;
            cout << (a + b);
}
```

Example 2:

```
// function definition
void add() {
            int a=10,b=20;
            cout << (a + b);
}
int main() {
   // calling the function after declaration.
   add();
   return 0;
}
```

The **Example 1** program is nearly identical to **Example 2**. The only difference is that here, the function is defined **after** the function call.

That's why we have used a function prototype in this example.

Return Statement

In the above programs, we have used void in the function declaration. For example,

```
void displayNumber() {
   // code
}
```

This means the function is not returning any value.

It's also possible to return a value from a function. For this, we need to specify the `returnType` of the function during function declaration.

Then, the `return` statement can be used to return a value from a function.

For example,

```
int add () {
            int a=10,b=20;
            return (a + b);
}
```

Here, we have the data type `int` instead of `void`. This means that the function returns an `int` value.

The code `return (a + b);` returns the sum of the two parameters as the function value.

The `return` statement denotes that the function has ended. Any code after `return` inside the function is not executed.

Example 3: Add Two Numbers

```
// program to add two numbers using a function

#include <iostream>

using namespace std;

// declaring a function
int add() {
            int a=10,b=20;
   return (a + b);
}

int main() {

   int sum;

      // the returned value in sum
   sum = add();

   cout << "Sum = " << sum << endl;

   return 0;
}
```

Output

```
Sum = 30
```