## LECTURE -5

## CONSTANTS

Constants are expressions with a fixed value.

## 5.1 Literals

*Literals are used to express particular values within the source code of a program*. We have already used these previously to give concrete values to variables or to express messages we wanted our programs to print out, for example, when we wrote:

```
a = 5;
```

the *5 in this piece of code was a literal constant.*

Literal constants can be divided in Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.

### 5.2 Integer Numerals

```
1776
707
-273
```

They are numerical constants that identify integer decimal values. Notice that to express a numerical constant we do not have to write quotes (") nor any special character. There is no doubt that it is a constant: whenever we write `1776` in a program, we will be referring to the value 1776.

### 5.3 Floating Point Numbers

They express numbers with decimals and/or exponents. They can include either a decimal point, an `e` character (that expresses "by ten at the Xth height", where X is an integer value that follows the `e` character), or both a decimal point and an `e` character:

```
3.14159     // Value of Pi
6.02e23     // Avagadro Number
1.6e-19     // Electric charge of an electron
3.0         // Floating 3
```

### 5.4 Character and string literals

There also exist non-numerical constants, like:

'z' ,'p',"Hello world","How do you do?"

The first two expressions represent single character constants, and the next two represent string literals composed of several characters. Notice that to represent a single character we enclose it between single quotes (') and to express a string (which generally consists of more than one character) we enclose it between double quotes (").

When writing both single character and string literals, it is necessary to put the quotation marks surrounding them to distinguish them from possible variable identifiers or reserved keywords. Notice the difference between these two expressions:

x

'x'

x alone would refer to a variable whose identifier is x, whereas 'x' (enclosed within single quotation marks) would refer to the character constant 'x'.

**5.5 Escape codes:** Character and string literals have certain peculiarities, like the escape codes. These are special characters that are difficult or impossible to express otherwise in the source code of a program, like newline (\n) or tab (\t). All of them are preceded by a backslash (\). Here you have a list of some of such escape codes:

\n newline

\r carriage return

\t tab

\v vertical tab

\b backspace

\f form feed (page feed)

\a alert (beep)

\' single quote (')

\" double quote (")

\? question mark (?)

\\ backslash (\)

### 5.6 Boolean literals

There are only two valid Boolean values: true and false. These can be expressed in C++ as values of type bool by using the Boolean literals true and false.

## 5.7 Defined constants (#define)

You can define your own names for constants that you use very often without having to resort to memory-consuming variables, simply by using the #define preprocessor directive. Its format is:

#define identifier value

For example:

```
#define PI 3.14159265
#define NEWLINE \n'
```

This defines two new constants: PI and NEWLINE. Once they are defined, you can use them in the rest of the code as if they were any other regular constant, for example:

**Program 5.1;**

```
// defined constants: calculate circumference
#include <iostream>
using namespace std;
#define PI 3.14159
#define NEWLINE '\n';
int main ()
{
  double r=5.0;               // radius                    31.4159
  double circle;
  circle = 2 * PI * r;
  cout << circle;
  cout << NEWLINE;
  return 0;
}
```

In fact the only thing that the compiler preprocessor does when it encounters #define directives is to literally replace any occurrence of their identifier (in the

previous example, these were PI and NEWLINE) by the code to which they have been defined (3.14159265 and '\n' respectively).

The #define directive is not a C++ statement but a directive for the preprocessor; therefore it assumes the entire line as the directive and does not require a semicolon (;) at its end. If you append a semicolon character (;) at the end, it will also be appended in all occurrences within the body of the program that the preprocessor replaces.

## 5.8 Declared constants (const)

With the const prefix you can declare constants with a specific type in the same way as you would do with a variable:

```
const int pathwidth = 100;
const char tabulator = '\t';
const zipcode = 12440;
```

*In case that no type is explicitly specified (as in the last example) the compiler assumes that it is of type int.*

### 5.9 Scope Resolution Operator

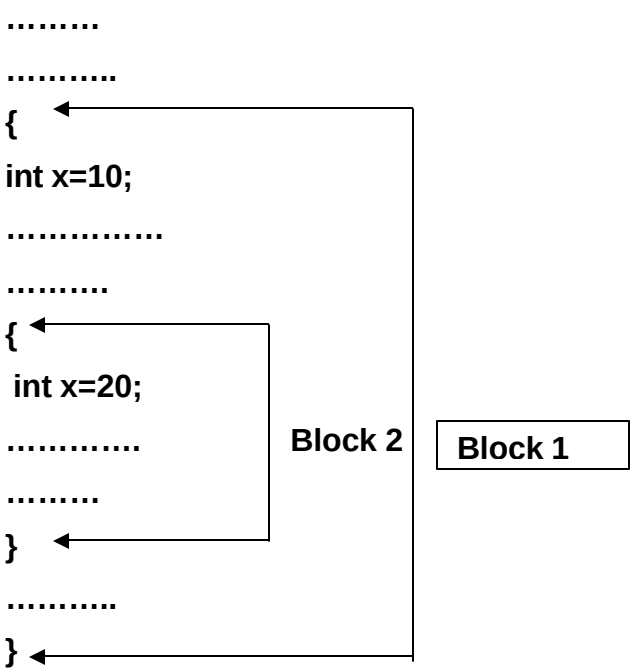
Like C, C++ is also a block structured language. Blocks and scopes can be used in constructing programs. We know that the same variable name can be used to have different meanings in different blocks. The scope of the variable extends from the point of its declaration till the end of the block containing the declaration. A variable declared inside a block is said to be local to that block. Consider the following segment of a program:

```
……
……..
{

int x=10;
……..
…….
}
……..
……
{
int x=20;
……..
```

**}**

The two declaration of x refer to two different memory locations containing different values. Blocks in C++ are often nested. For example, the following style is common:




Block 2 is contained in block 1. Note that a declaration in an inner block hides a declaration of the same variable in an outer block, and therefore, each declaration of **x** causes it to refer to different data object. Within the inner block, the variable **x** will refer to the data object declare therein.

In C++, the global version of a variable cannot be accessed from within the inner block. C++ resolves this problem by introducing a new operator **::** called *scope resolution operator*. This can be used to uncover a hidden variable. It takes the following form:

**:: variable_name**

Following program explains the scope resolution operator.

**Progrm 5.2: Program explaining scope resolution operator**

```
//Program explaining scope resolution operator


#include <iostream>
using namespace std;
int m=10; //global m
int main()
{
```

```
        int m=20; // m redeclared, local to main
        {
                int k=m;
                int m=30; //m declared again
                        // Local to inner block
                cout<<"we are in inner block\n";
                cout<<"k="<<k<<"\n";
                cout<<"m="<<m<<"\n";
                cout<<"::m="<<::m<<"\n";
        }
        cout<<"\n We are in outer block\n";
        cout<<"m="<<m<<"\n";
        cout<<"::m="<<::m<<"\n";
        return 0;
}
```

**Output of this program is :**

```
/*we are in inner block
k=20
m=30
::m=10
 We are in outer block
m=20
::m=10
Press any key to continue*/
```