Shifting Positive Numbers

If the int data type occupies four bytes in the memory, the rightmost eight bits of the number 10 are represented in binary as

```
0    0    0    0    1    0    1    0
```

When you do a right shift by 3(10 > > 3), the result is

```
0    0    0    0    0    0    0    1
```

$10/2^3$, which is equivalent to 1.

When you do a left shift by 3 (10 < < 3), the result is

```
0    1    0    1    0    0    0    0
```

$10*2^3$, which is equivalent to 80

Shifting Negative Numbers

For negative numbers, the unused bits are initialized to 1. Therefore, -10 is represented as:

```
1    1    1    1    0    1    1    0
```

1.6.6 Bit-wise Logical Operators

Operator	Description	Example	Explanation
& (AND)	Evaluates to a binary value after a bit-wise AND on the operands	a & b	AND results in a 1 if both the bits are 1, any other combination results in a 0
! (OR)	Evaluates to binary value after a bit-wise OR on the two operands	a ! b	OR results in a 0 when both the bits are 0, any other combination results in a 1.
^ (XOR)	Evaluates to a binary value after a bit-wise XOR on the two operands	a ^ b	XOR results in a 0 if both the bits are of the same value and 1 if the bits have different values.
~ (inversion)	Converts all 1 bits to 0s and all 0 bits to 1s		Example given below.

In the example shown in the table, a and b are integers and can be replaced with expressions that give a true or false (boo1) result. For example, when both the expressions evaluate to true, the result of using the & operator is true. Otherwise, the result is false.

The ~ Operator

If you use the ~ operator, all the 1s in the bite are converted to 0s and vice versa. For example, 10011001 would become 01100110.

1.6.7 Logical Operators

Use logical operators to combine the results of Boolean expressions.

Operator	Description	Example	Explanation
&&	Evaluates to true, if both the conditions evaluate to true, false otherwise	a>6&&y<20	The result is true if condition 1 (a>6) and condition 2 (y<20) are both true. If one of them is false, the result is false.
||	Evaluate to true, if at least one of the conditions evaluates to true and false if none of the conditions evaluate to true.	a>6 || y < 20	The result is true if either condition1 (a>6) and condition2 (y<20) or both evaluate to true. If both the conditions are false, the result is false.

Short Circuit Logical Operators

These operators (&&, ||) appear to be similar to the bit-wise & and | operators, except that they are limited to Boolean expressions only. However, the difference lies in the way these operators work. In the bit-wise operators, both the expressions are evaluated. This is not always necessary since:

false & a would always result in false

true | a would always result in true

Short circuit operators do not evaluate the second expression if the result can be obtained by evaluating the first expression alone.

For example

a < 6 && y > 20

The second condition (b>20) is skipped if the first condition is false, since the entire expression will anyway, be false. Similarly with the || operator, if the first condition evaluates to true, the second condition is skipped as the result, will anyway, be true. These operators, && and ||, are therefore, called short circuit operators.

If you want both the conditions to be evaluated irrespective of the result of the first condition, then you need to use bit-wise operators.

1.6.8 Conditional Operators

Operator	Description	Example	Explanation
(condition) val1, val2	Evaluates to val1 if the condition returns true and val2 if the condition returns false	a = (b>c) ? b:c	A is assigned the value in b, if b is greater than c, else a is assigned the value of c.

This example finds the maximum of two given numbers.

```
if (num1 > num2)
{
imax = num1;
}
else
{
imax = num2;
}
```

In the above programme code, we determining whether num1 is greater than num2. The variable, imax is assigned the value, num1, if the expression, (num1 >num2), evaluates to true, and the value, num2, if the expression evaluates to false. The above programme code can be modified using the conditional operator as:

imax = (num1 > num2) ? num1 : num2;

The ?: Operator is called the ternary operator since it has three operands.

This example calculates the grade of a Ram based on his marks.

```
int marks = 0;

cout << "Please enter marks of the Ram;

cin >> marks;

char grade = (marks < 80);
```

The variable, grade is assigned the value, 'A'. If Ram's score is less than or equal to 80 grade is assigned the value 'B'.

The following code will display either "PASSED" or "FAILED", depending on the value in the variable, score.

```
Cout << {score > 50? "PASSED" or FAILED, depending on the value in the variable, score.

Cout << (score > 50)? "PASSED" : "FAILED"} << endl;
```

1.6.9 Scope Resolution Operator

C++ is also a block-structured language. We know that the same variable name can be used to have different meanings in different blocks. The scope of a 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. Blocks in C++ are often nested. For example the following style is common:

```
. . . . .
. . . . .
{
      int x=10;
      . . . . .
      . . . . .
{
         int x=1;
         . . . . .       Block 2   Block 1
         . . . . .
}
. . .
. .
}
```

The declaration in an inner block hides a declaration of the same variable in the outer block. C++ resolves this problem by introducing a new operator:: called the scope resolution operator. This can be used to uncover a hidden variable. It takes the following format:

::variable_name

This operator allows access to the global version of a variable.

Programme illustrating the use of :: is given below:

```
#include <iostream.n>
int m=10;                    //globalm
main()
{
int m=20                     //m is redeclared; local to main
        {
                int n=m;
                int m=30;    //m is declared again
                cout << "we are in innerblock \n";
                cout << "n=" <<n<< "\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";
}
```

```
The output of the programme:
We are in inner block.
n = 20
m = 30
::m = 10
We are in outer block
m = 20
::m = 10
```

1.6.10 Member Dereferencing Operators

Once a class (or a structure) is defined, its members (both data and methods/function) can be accessed (or referenced) using two operators – (.) dot operator and (->) arrow operator. While (.) operator takes class or struct type variable as operand, (->) takes a pointer or reference variable as its operand.

With class and structure type variables, (.) dot operator is used to reference the members, as shown in the following programme code:

```
struct abc
{      int a, b, c;        };
class pqr
{      int p, q, r;
       int doit(); };
abc structone;
pqr classone;
structone.a = 20;
structone.b = 120;
structone.c = 30;
classone.p = 56;
classone.q = 16;
classone.r = 51;
classone.doit();
```

Evidently, each member can be referenced by using (.) dot operator. However, (.) operator does not work on pointer or reference type variables, as is shown in the following code snippet.

```
abc *ptrAbc;
pqr *ptrPqr;
abc abcone;
pqr pqrone;
ptrAbc = abcone;   //pointer variable ptrAbc points to object abcone
ptrAbc->a;
ptrPqr->doit();
```

1.6.11 Memory Management Operators

Along with malloc(), calloc() and frec() functions. C++ also defines two unary operators new and delete that perform the task of allocating and freeing the memory. Since these operators manipulate memory on the free store, they are also known as free store operators. A data object created inside a block with *new* will remain in existence until it is explicitly destroyed by using *delete*.

The general form of the new operator

pointer_variable = new data_type;

The pointer_variable holds the address of the memory space allocated.

p = new int;

q = new float;

We can also initialize the memory using the *new* operator. This is done as follows:

pointer_variable= new data_type(value);

Value specifies the initial value as shown below:

int*p = new int(25);

When an object is no longer needed, it is destroyed to release the memory space for reuse. The general form is:

delete pointer_variable;

For example,

delete p;

delete q;

If we want to free a dynamically allocated array we must use the following form of *delete*.

delete [size]pointer_variable;

For example, statement

delete [ ]p;

will delete entire array pointed by p.

The *new* operator has several advantages over the function malloc().

1. It automatically computes the size of the data objects.
2. It automatically returns the correct pointer type so there is no need to use a type cast.
3. It is possible to initialize the object while creating the memory space.
4. They both it new and delete can be overloaded.

1.7 TYPE CONVERSION

C++ is very strict with regard to type compatibility as compared to C. For instance, C++ defines *int*, *short int*, and *long int* as three different types. They must be cast when their values are assigned to one another. Similarly, *unsigned char*, *char*, and *signed char* are considered as different types, although each of these has a size of one byte. In C++, the types of values must be the same for complete compatibility. Otherwise, a cast must be applied.

These restrictions in C++ are necessary in order to support function overloading where two functions with the same name are distinguished using the type of function arguments.

Another notable difference is the way char constants are stored. In C, they are stored as *int*. Therefore,

```
sizeof('x')
```

is equivalent to

sizeof(int)

in C. But, in C++, *char* is not promoted to the size of *int* and therefore, sizeof('x')

equals

`sizeof(char)`

When constants and variables of different types are mixed in an expression, they are converted to the same type.

"The process of converting one predefined type into another is called type conversion."

C++ facilitates the type conversion in two forms:

(a) ***Implicit type conversion:*** An implicit type conversion is a conversion performed by the compiler without programmer's intervention. It is applied generally whenever differing data types are intermixed in an expression (mixed mode expression), so as not to lose information.

(b) ***Explicit type conversion:*** An explicitly type conversion is user-defined that forces an expression to be of specific type. It is called Type Casting.

Type casting is done in C++ as shown below:

(type) expression

where *type* is a valid C++ data type to which the conversion is to be done.

1.8 DECLARATION OF VARIABLES

If you are familiar with C, you would know that, in C, all variables must be declared before they are used in executable statements. This is true with C++ as well. However, there is a significant difference between C and C++ with regard to the place of their declaration in the programme.

C requires all the variables to be defined at the beginning of a scope. When we read a C programme, we usually come across a group of variable declarations at the beginning of each scope level. Their actual use appears elsewhere in the scope, sometimes far away from the place of declaration.

Before using a variable, we should go back to the beginning of the programme to see whether it has been declared and, if so, of what type it is. C++ allows the declaration of a variable anywhere in the scope. This means that a variable can be declared right at the place of its first use. This makes programmes much easier to write and reduces the errors that may be caused by having to scan back and forth. It also makes the programme easier to understand because the variables are declared in the context of their use. The following example illustrates this point.

```
main ( )
{
      float x;                          // declaration
      float sum = 0;
      for (int i = 0; i<5; i++)         //declaration
      {
            cin >> x;
            sum = sum+x;
```

```
        }
        float average;                    //declaration
        average = sum / i;
        cout << average;
}
```

1.8.1 Dynamic Initialization of Variables

One additional feature of C++ is that it permits initialization of the variables at run time. This is referred to as dynamic initialization. Remember that, in C, a variable must be initialized using a constant expression and the C compiler would fix the initialization code at the time of compilation. However, in C++, a variable can be initialized at run time using expressions at the place of declaration. For example, the following are valid initialization statements:

```
...
...
int n = strlen(string);
...
float area = 3.14159 *rad *rad;
```

This means that both the declaration and initialization of a variable can be done simultaneously at the place where the variable is used for the first time. The two statements in the following example of the previous section:

```
float average;              // declare where it is necessary
average = sum / i;
```

can be combined into a single statement:

```
float average = sum /i;     // initialize dynamically
// at run time
```

Dynamic initialization is extensively used in object-oriented programming. We can create exactly the type of object needed using information that is known only at the run time.

1.8.2 Reference Variables

C++ introduces a new kind of variable known as the reference variable. A reference variable provides an alias (alternative name) for a previously defined variable. For example, if we make the variable *sum* a reference to the variable *total*, then *sum* and *total* can be used interchangeably to represent that variable. A reference variable is created as follows:

```
data_type & reference_name = variable_name
```

Example:

```
float total = 100;
float &sum = total;
```

Total is a float type variable that has already been declared, *sum* is the alternative name declared to represent the variable *total.* Both the variables refer to the same data object in the memory. Now, the statements

```
cout << total;
```

and

```
cout << sum;
```

both print the value 100. The statement

```
total = total + 10;
```

will change the value of both *total* and *sum* to 110. Likewise, the assignment

```
sum = 0;
```

will change the value of both the variables to zero.

A reference variable must be initialized at the time of declaration. This establishes the correspondence between the reference and the data object that it names. Note that the initialization of a reference variable is completely different from assignment.

Note that C++ assigns additional meaning to the symbol &. Here, & is not an address operator.

The notation *float &* means reference to *float*. Some more examples are presented below to illustrate this point:

```
int n[10];
int &x = n[10];          //x is alias for n[10]
char &a = '\n';          // initialize reference to a literal
```

The variable *x* is an alternative to the array element *n[10].* The variable *a* is initialized to the *new line* constant. This creates a reference to the otherwise unknown location where the new line constant *\n* is stored.

The following references are also allowed:

```
(i)    int x;
       int *p=&x;
       int &m = *p;
(ii)   int &n = 50;
```

The first set of declarations causes *m* to refer to *x* which is pointed to by the pointer *p* and the statement in (ii) creates an *int* object with value 50 and name *n*.

A major application of reference variables is when passing arguments to functions. Consider the following code snippet:

```
void f(int &x)           // uses reference
{
       x = x+10;         // x is incremented; so also m
}
main ( )
```

```
{
        int m = 10;
        f(m);                  // function call
        ....
        ...
}
```

When the function call *f(m)* is executed, the following initialization occurs:

Int &x = m;

Thus *x* becomes an alias of *m* after executing the statement *f(m);*.

Since the variable *x* and *m* are aliases, when the function increments *x*, *m* is also incremented. The value of *m* becomes 20 after the function is executed. In traditional C, we accomplish this operation using pointers and dereferencing techniques.

The call by reference mechanism is useful in object-oriented programming because it permits the manipulation of objects by reference and eliminates the copying of object parameters back and forth. Note that the references can be created not only for built-in data types but also for user-defined data types such as structures and classes. References work wonderfully well with these user-defined data types.

1.9 STATEMENTS

The statements like,

```
Var1 = 20;
Var2 =Var1 + 20;
```

assign values to two variables. The equal sign(=) cause the value on the right to be assigned to the variable on the left.

1.10 SIMPLE C++ PROGRAMME

To start with, let us look at a simple C++ programme that prints on the users' console (screen) a message - WELCOME TO THE WORLD OF C++. Here is the code:

```
#include <iostream.h>
void main()
{
        cout<<" WELCOME TO THE WORLD OF C++";
}
```

Isn't simple? A C programmer would find most of the parts of the code very familiar. Let us have a quick run through the programme in order to understand what each line of code means.

If you write this programme in a text file with extension .cpp; compile, link and run this programme, you will see the desired output on the screen. We will learn in a while how to write, compile, link and run a C++ programme in a short while.

The very first line in the code begins with #. In C++ (as also in C), a line beginning with # is a command to the compiler or a compiler directive. *#include* tells the compiler that - *while compiling this programme, substitute this line with the contents of file iostream.h.* The angle brackets (< and >) around the file name indicates that the file is located in the default directory called *include* (more on this later). You can also use (" and ") instead of (< and >) but in this case you will have to mention the location of the directory explicitly.

As a result, the content of the file *iostream.h* is placed in place of this single line of code. Files having *.h* extension are called header files. These files contain definition of functions that could be used in you programme. In particular, *iostream.h* has the definition of *cout* operator, which has been used in the programme. As we will learn shortly, *cout* simply prints whatever is handed over to it.

There are more than eight hundred lines in *iostream.h* file (you can check it by opening using a text editor). The programme, consequently, becomes as shown below:

```
#ifndef __cplusplus
#error Must use C++ for the type iostream.
#endif
#ifndef __IOSTREAM_H
#define __IOSTREAM_H
#if !defined( __DEFS_H )
#include <_defs.h>
#endif
#if !defined( __MEM_H )
#include <mem.h>
#endif
#pragma option -Vo-
#define EOF (-1)
#define zapeof(i) ((unsigned char)(i))
typedef long streampos;
typedef long streamoff;
_CLASSDEF(ios)
_CLASSDEF(streambuf)
.........;
.........;
extern istream_withassign _Cdecl cin;
extern ostream_withassign _Cdecl cout;
extern ostream_withassign _Cdecl cerr;
extern ostream_withassign _Cdecl clog;
ostream & _Cdecl endl(ostream &); // insert newline and flush
ostream & _Cdecl ends(ostream &); // insert null to terminate string
ostream & _Cdecl flush(ostream &);// flush the ostream
ios &     _Cdecl dec(ios &);      // set conversion base to decimal
ios &     _Cdecl hex(ios &);    // set conversion base to hexadecimal
ios &     _Cdecl oct(ios &);      // set conversion base to octal
istream & _Cdecl ws(istream &);   // extract whitespace characters
#pragma option -Vo.
      #endif
void main()
{
      cout<<" WELCOME TO THE WORLD OF C++";
}
```

Interestingly, the compiler will also report that it has compiled about 814 (or so) lines. This inclusion mechanism has a lot of advantages. For one, it allows the programmers to break large programmes into smaller files.

The second line reads as - void main(). This is the main ingredient of every C++ programme. Every C++ programme must have one (and only one) main function. The execution of the programme actually means executing main function. *main* function may call other functions which in turn can call other functions.

The body (action part) of a function is contained within curly braces ({ and }). In our case, the body of the main function contains a single statement. A statement is terminated by a semicolon (;). The statement - *cout<<" WELCOME TO THE WORLD OF C++";* - calls the *cout* operator which is responsible for printing the message on the screen.

1.11 MANIPULATORS

These operators are used to format the data display. The commonly used manipulators are *endl* and *setw*.

1. *endl* manipulator when used causes a linefeed to be inserted. For example,

```
cout    <<"m=" <<m <<endl
        <<"n=" <<n <<endl
        <<"p=" <<p <<endl;
```

would cause the output as:

Assuming the values of variables as 2597, 14 and 175.

```
m = 2597
n = 14
p = 175
```

2. The setw manipulator does the job as follows:

    cout << setw(5) << sum << endl;

The manipulator setw(5) specifies a field width 5 for printing. The value of the variable sum is printed as 5 digits. This value is right justified within the field.

		3	4	5

We can also write ɔur own manipulators as:

```
#include <iostream.h>
ostream & Symbol (ostream & output)
{
      return output <<"\+Rs";
}
```

The *symbol* is the new manipulator, which represents Rs. The identifier symbol can be used

Wherever we need to display the string "Rs.".

1.11.1 Manipulate Function

Manipulators are special functions that can be included in the I/O statements to alter the format parameters of a stream. The most commonly used manipulators functions are given below. To access these function, we must include the file iomanip.h in the programme.

- *Setw (int w):* As discussed earlier, this manipulator function is used to changes or set the field width for output to w.
- *Set fill (char C):* It is used to change or set the fill character to n (default is a space).
- *Set precision (int p):* It is used to change or set the floating point precision to p.
- *Set base (base n):* It is used to changes base to n, where n is 8, 10, or 16. If n is zero, output is base 10, but input uses the C convention: 10 is 10, 010 is 8 and O x c is 12.
- *Set Pos flags (print flags f):* This manipulator function changes or sets the format flag f. Setting remain in effect until next change.
- *Reset i os flags (fmt flags f):* This manipulators clears only the flags specified by f. Setting remains in effect until next change.

1.11.2 Predefined Manipulators

Manipulators comes in tow flavors: Those take an argument as discussed in previous topic and those that don't. Manipulators with no arguments are known as predefined manipulators. These are provided in iostream.h. The most commonly used predefined manipulators are given below:

- ***end l;*** This manipulator sends a newline to the stream and flushes it.
- ***skipws:*** It is used to skip white space on input.

- *no skipws:* Its purpose is not to skip whitepace on input.
- *dec:* it is used for the decimal conversion.
- *oct :* It is used for the octal conversion.
- *hex:* It is used for hexadecimal conversion.
- *left;* This manipulator is used to left align and pad on right in the field.
- *right:* This manipulator is used to right align and pad on left in the field.
- *internal:* It is used for padding between sign or base indicator and value.
- *showpos:* Shows plus sign for positive values.
- *noshowpos:* Do not show plus sign for positive value.
- *uppercase:* It displays uppercase A-F for hex values and E for Scientific values.
- *no upper case:* It do not display hex values in upper case.
- *show point:* It shows decimal point and trailing zeros for float values.
- *scientific:* It uses scientific notation for printing float value.
- *fixed:* It uses fixed notation for printing float values.
- *Ends:* It inserts null character to terminate an output string.
- *Flush:* It flushes the output stream.
- *lock:* It unlocks the file handling.

Let us consider an example to understand the usage of above said manipulators:

```
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
void main( )
{
      int I = 2b;
      float b = 123.500032;
      char str[ ] = "I am a c++ programmer";
      clrscr( );
cout<<setios flags (ios: :unitbuf | ios: :stdio | ios:: showpos);
      cout<<i<<end l;
      cout<<      set iosflogs(ios :: showbase | ios :: uppercase);
      cout << hex << i << end;
      cout < < oct << i << endl;
      cout < < set fill ('o');
      cout << set w (40) <<str << endl;
```

1.12 INPUT AND OUTPUT STREAM FLAGS

1.12.1 I/O Streams

There are data members in the ios class that control the width of the output field. The fill characters are used when the data doesn't fill the output field, and the precision for printing floating point numbers. The values of these variables can be read and written by member functions of the some name.

Width( ) is used to define the width of a field necessary for the output of an item.

Syntax cout.width (w)

Where 'w' is the field width (number of columns). Remember that the field width should be specified for each item separately. C+ + never truncates the values and therefore, if t he specified field width is smaller than the size of the value to be printed, C+ + expands the field to fit the value.

The float values are printed with six digits after the decimal point. So, precision ( ) method is used to specify the number of digits to be displayed after the decimal point syntax:

Cout.precision (d);

Where'd' is the number of digits to the right of the decimal point. Precision ( ) method retains the setting in effect until it is reset.

Fill( ) function is used to fill the unused position of the field by any desired character.

Syntax: cout fill(ch)

Where 'ch' represents the character which is used for filling the unused positions like precision( ), fill( ) stays in effect till we change it.Let us consider a small example to understand the usage of above said I/o stream function.

```
#include<iostream .h>
      Main ( )
      {
            cout.fill ('<');
            cout.Precision (3);
            for (int n = 1; n <=6; n++)
            {
                  cout.width (5);
                  cout << n;
                  cout.Width (10);
                  cout << 1.0 / float (n) << "/n";
                  if (n = = 3)
                  cout.fill('>'); }
                  cout.fill ('$'); //fillreset
                  cout.width (20);
                  cout << 12.31245 << "/n";
      }
```

1.12.2 Flags

The ios class contains data members that help formatting of stream data, the errors-status flags and the file operation mode. The following are the complete list of the formatting flags.

Flag	Meaning
Skipws	Ignore whitespaces
Showbase	Use base indicator on output
Show point	Show decimal point and trailing zeros for floats
Upper case	Use uppercase for hex values and scientific values.
Showpos	Display + before positive integer.
Unit buf	Unit buffering. Flush all streams after each insertion.
Stdio	Synchronize the stream with C standard I/O system.
Dec	Convert to decimal
Oct	Convert to octal
Hex	Convert to hexadecimal
Left	Left justify output
Right	Right justify output
Internal	Use padding between sign or base indicator and number
Scientific	Use exponential format on floating point
Boolalpha	Convert bool to "true" or "false" strings.

There are two types of flags:

(a) On/Off Flags.

(b) Flags that work in a group.

The On/Off flags are simple. They can be turned on through the set f( ) function and off though unset f() function. For example

```
cout.set f (ios:: left);
cout<<"Are you a C++ programmer?");
To remove the left justification, use
cout.unset f (ios:: left);
```

The flags that can be set/unset in this manner. Include ski pw, showbase, showpoint, upper case, showpos, unit buf and stdio.

The second type of formatting flags work in a group. You can have only one of these flags on at a time.

To set these flags you must use the second form of set f ( ) function. Example

cout.set f (ios:: hex, ios:: base field);

This calls first clears all the bits in the ios:: basefield, then sets ios:: hex:

Check Your Progress

Fill in the blanks:

1. The modifiers signed, unsigned, long, and short may be applied to .... and integer basic data types.
2. The keyword, const (for constant), precedes the .... of a variable.
3. There are data members in the .... class that control the width of the output field.
4. If you use the .... operator, all the 1s in the bite are converted to 0s and vice versa.

1.13 LET US SUM UP

C++ language is made up of letters, words, sentences and constructs just like English language. A collection of characters, much like a word in English language are called tokens. A token can be a keyword, or identifier, constant, string or an operator. Keywords are the reserve words that cannot be used as names of variable or other user-defined programme elements. Identifiers refer to the names of variables, functions, arrays, classes, etc. created by the programmer. Basic types, Derived types and user defined are the three types of data in C++. There are five basic data types: char, int, float, double and void.

An *array* represents named list of finite number of similar data elements. A function is a named part of a programme that can be invoked from the other parts of the programme. A *pointer* is a variable that holds a memory address of another variable. A *reference* is an alternative name for an object. A *constant* is a data item whose data value can never change during the programme run.

Operators are the symbols that represent specific operations. Arithmetic operators are unary +, unary-, +, -, *, / and % . '%' uses pure integer division thereby requires integer operands. Comparison operators, also called relational operators, compare the relationships among values. The order of evaluation among logical operators is NOT, AND, OR, i.e.!, &&, | |.

Along with malloc (), calloc () and frec () functions, C++ also defines two unary operators, new and delete, that perform the task of allocating and freeing the memory.

Generally, a programme in C++ includes a header file iostream.h for standard stream input /output facilities. The declarations of cin (standard stream output) and cerr (standard stream error) are defied in iostream.h. If iostream.h is not included in a programme then every reference to cin, cout or cerr will produce a type error.

A stream is a sequence of bytes and at lowest level. Files are interpreted as streams of bytes. Every C++ programme must contain a function named main ( ). The programme execution begins at main ( ).

Every executable statement in C++ must be terminated by a semi colon (;) .

Every main ( ) having a return type int must end with a statement Return 0;

The multiple use of input or output operators ("> >" or "< <") in a statement is called cascading of I/O operators.

1.14 KEYWORDS

Tokens: A collections of characters, much like a word in English language. The smallest individual unit in a programme.

Keywords: Explicitly reserved identifiers that cannot be used as names for the programme variables or other user defined programme elements.

Identifies: The names of variables, functions, arrays, classes, etc. created by the programmer.

Enumerated Data Type: An enumerated data type is user defined type that provides a way for attaching names to numbers, thereby increasing comprehensibility of the code.

Derived Data Type: C++ allows programmers to derive complex data types using simple basic data types. These data types are referred to as derived data types.

Operands: The data items that operators acted upon are called operands.

Expression: A combination of variables, constants and operators written according to some rules.

Character constant: One or more characters enclosed in single quotes.

String constant: Sequence of characters enclosed in double quotes.

Scope resolution operator: This operator enables a programme to access a global variable when a local variable of the same name is in scope.

Iosream.h: Header file included in every C++ programme to implement input/output facilities.

Manipulator: Operators used o format the data display.

1.15 QUESTIONS FOR DISCUSSION

1. What is meant by taken? Name the tokens available in C++.
2. What are keywords? Can keywords be used as identifiers?
3. What is an integer constant? Write integer-forming rule of C++.
4. How many types of integer constants are allowed in C++? How are they written?
5. What is a character constant in C++?
6. What is meant by a floating constant in C++? How many ways can a floating constant be represented into?
7. Which character is automatically added to strings in C++?
8. How are floating constants represented in C++? Give examples to support your answer.
9. How are string-literals represented and implemented in C++?
10. What are operators? What are their functions? Give examples of some unary and binary operators.
11. Describe various types of type conversion available in C++.
12. What are the differences between dynamic and reference declaration of variables?
13. Describe assignment statements.

14. Write a C++ programme to display and add five numbers
15. Describe the features of iostream.h
16. What are predefined manipulators?

Check Your Progress: Model Answer

1. Character
2. data type
3. ios
4. ~

1.16 SUGGESTED READINGS

Robert Lafore, *Object-oriented Programming in Turbo* C++, Galgotia Publications.

E Balagurusamy, *Object-Oriented Programming with* C++, Tata Mc Graw-Hill

Herbert Schildt, *The Complete Reference* C++, Tata Mc Graw Hill

LESSON 2

CONTROL EXPRESSIONS

CONTENTS

2.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Explain the concept of control statements
- Discuss how to identify the conditional expressions
- Describe the switch statement
- Identify and explain the loop statements
- Discuss the breaking control statements

2.1 INTRODUCTION

C++ programme may require that a logical test be carried out at some particular point within the programme. One of several possible actions will then be carried out, depending on the outcome of the logical test. This is known as branching. There is also a special kind of branching, called selection, in which one group of statements is selected from several available groups. In addition, the programme may require that a group of instructions be executed repeatedly, until some logical condition has been satisfied. This is known as looping. Sometimes the required number of repetitions is known in advance; and sometimes the computation continues indefinitely until the logical condition becomes true.

All of these operations can be carried out using the various control statements included in C++. Several logical expressions are given here.

```
count < = 100

sqrt( x + y + z) > 0.00

answer = = 0

balance >= cutoff

ch1 <  'A'

letter1 = 'a'
```

The first four expressions involve numerical operands. Their meaning should be readily apparent. In the fifth expression, ch1 is assumed to be a char type variable. This expression will be true if the character represented by ch1 comes before T in the character set, i.e., if the numerical value used to encode the character is less than the numerical value used to encode the letter T.

The last expression makes use of the char-type variable letter. This expression will be true if the character represented by letter is something other than x.

In addition to the relational and equality operators, C++ contains two logical connectives (also called logical operators), && (AND) | | (OR), and the unary negative operator!. The logical connectives are used to combine logical expressions, thus forming more complex expressions. The negation operator is used to reverse the meaning of a logical expression (e.g., from true to false).

2.2 CONDITIONAL EXPRESSIONS

The conditional expressions allow choosing the set-of-instructions for execution depending upon an expression, truth value. C++ provides two types of conditional statements: if and switch.

2.2.1 if Statement

An if statement tests a particular condition; if the condition evaluates to true, a course-of-action is followed i.e., a statement or set-of-statements is executed. Otherwise (if the condition evaluates to false), the course-of-action is ignored.

Syntax: if (expression) statement;

where a statement may consist of a single statement, a compound statement, or nothing (in case of empty statement). The expression must be enclosed in parentheses. If the expression evaluates to true i.e., a non zero value, the statement is executed, otherwise ignored.

Eg: int A, B, C;

```
        cin >>A>>B;
        if (A>10 && B<15)
        { C=(A-B) * (A+B);
                {cout << "The result is" <<C<< "\n";
                }
```

2.2.2 if-else Statement

The *if* conditional construct is followed by a logical expression in which data is compared and a decision is made based on the result of the comparison. The syntax is:

```
if (boolean_expr)
{
        statements;
}
else
{
        statements;
}
```

The *if-else* statement is used to carry out a logical test and then take on of two possible actions, depending on the outcome of the test (i.e., whether the outcome is true of false).

The *else* portion of the *if-else* statement is optional. Thus, in its simplest general form, the statement can be written as:

if (expression) statement;

The expression must be placed in parentheses, as shown. In this form, the statement will be executed only if the expression has a nonzero value (i.e., if expression is true). If the expression has a value of zero (i.e., if expression is false), then the statement will be ignored.

The statement can be either simple or compound. In practice, it is often a compound statement, which may include other control statements. For example, consider the following programme:

```
#include<iostream.h>
int main()
{
        char chr;
        cout<<"Please enter a character:" ;
        cin>>chr;
```

```
    if (chr == 'X')
        cout<<endl<<"The character is X";
    else
        cout<<endl<<"The character is not X";
    return 0;
}
```

Note that the operator, = =, used for comparing two data items, is different from the assignment operator, =,

In an *if-else* block of statements, the condition is evaluated first. If the condition is true (value is non-zero), the statements in the immediate block are executed, if the condition is false (value is zero) the statements in the else block are executed.

In the above example, if the input character is 'X', the message displayed is, 'the character is X' otherwise, the message, 'The character is not X' is displayed. Also, note that the condition has to be specified within parenthesis.

If, however, the operator, =, is used instead of the operator, = =, the statement is executed as an assignment. For example, if the statement, if (chr = = 'X'), was written as, if (chr = 'X'), then, chr would have been assigned the value, 'X' and the condition would be evaluated as true. Thus, the message, the character is 'X' would have been displayed, regardless of the input.

This programme could also have been written as given below:

```
#include<iostream.h>
int main()
{
    char chr;
    cout << "Please enter a character: ";
    cin >> chr;
    if (chr != 'X')
        cout << endl << "The character is no X";
    else
        cout << endl << "The character is X";
    return 0;
}
```

The operator, !=, is an equivalent of not equal to, and is written without any space between ! and =.

The general form of an *if* statement which includes the *else* clause is:

If (expression) statement1 else statement2;

If the expression has a nonzero value (i.e., if expression is true), then statement1 will be executed. Otherwise (i.e., if expression is false), statement2 will be executed.

The following, for example, programme uses a nested if...else construct to check if the input character is uppercase or lowercase.

```
#include<iostream.h>
int main()
{
        char inp;
        cout<<"Please enter a character: ";
        cin>>inp;
        if (inp>='A')
                if (inp<= 'Z')
                        cout<<endl<<"Uppercase";
                else if (inp >= 'a')
                {
                        if (inp <= 'z')
                                cout<<endl<<"Lowercase";
                        else
                                cout<<endl<<"Input character > z";
                }
                else
                {
                        cout<<endl<<"input character>z but less than a";
                }
else
{
                cout<<endl<<"Input character less than A";
}
return 0;
}
```

The following programme shows how the clarity of steps can be increased by using braces ({and}) for enclosing parts of the code that follow an *if* or an *else* statement.

```
#include<inostream.h>
int main()
{
        char inp;
        cout << "Enter a character: ";
        cin >> inp;
        if (inp >= 'A')
        {
                if (inp <= 'z')
                {
                        cout << endl << "Uppercase";
                }
                else if (inp >= 'a')
                {
                        if (inp <= 'z')
                        {
                                cout << endl << "Lowercase";
                        }
                }
        }
        return 0;
}
if ( circle) {
                scanf ( ` &f` ,  & radius);
                area = 3. 14159 * radius  * radius;
                printf ( * Area of circle = & f* , area);
}
else
{
                scanf ( `&f    & f``, & length , & width) ;
                area = length * width;
                printf  ( * Area of rectangle = &f `, area);
}
```

The above example shows how an area can be calculated for either of two different geometric figures. If circle is assigned a nonzero value, however, then the length and width of a rectangle are read into the computer, the area is calculated and then displayed. In each case, the type of geometric figure is included in the label that accompanies the value of the area. Here is an example of three nested if - else statements.

```
If (  ( time > = 0.) && ( time < 12.) ) printf ( * Good Morning*) ;
else if ( ( time > = 12.) && ( time < 18.) ) printf ( * Good Afternoon *);
else if ( ( time  > = 18.) && ( time < 24.)) printf (  * Good Evening *) ;
else printf  ( * Time is out of range *);
```

This example causes a different message to be displayed at various times of the day. Specifically, the message Good Morning will be displayed if time has a value between 0 and 12; Good Afternoon will be displayed if time has a value between 12 and 18; and Good Evening will be displayed if time has a value between 18 and 24. An error message (Time is out of range) will be displayed if the value of time is less than zero, or greater than or equal to 24.

The following programme uses a cascading *if-else* construct to determine if the input character is a vowel, else it prints an appropriate message.

```
#include<iostream.h>
int main ( )
{
        char in_chr;
        cout << " Enter a character in lowercase " < , endl;
        cin >> in_chr;
        if (in_chr == `e`)
                cout << endl << " Vowel    a" << endl;
        else if (in_chr == `e`)
                cout <, endl  << "Vowel      e" <, endl;
        else if ( in_chr == ` i )
                cout  << endl << " Vowel   i " << endl);
        else if ( in_chr == ` o  ` )
                cout <, endl << " Vowel  o " << endl;
        else if ( in_chr == `u`)
                cout <, endl <<  " Vowel u" << endl;
        else
                cout << endl << " The character is not a Vowel" << endl;
        return o;
}
```

2.2.3 switch Statement

C++ provides a multiple-branch selection statement known as switch. This selection statement successively tests the value of an expression against a list of integer or character constants. When a match is found, the statements associated with that constant are executed.

The syntax is:

```
switch (expression)
        {
        case constant1: statement sequence 1;
        break;
case constant2: statement sequence 2;
break;
case constant1: statement sequence n-1;
break;
.     .     .
.     .     .
.     .     .
[default: Statement sequence n];
}
```

The expression is evaluated and its values are matched against the values of the constants specified in the case statements. When a match is found, the statement sequence associated with that case is executed until the break statement or the end of switch statement is reached. The default statement gets executed when no match is found. The default statement is optional and, if it is missing, no action takes place if all matches fail.

A case statement cannot exit by itself, outside of a switch. The break statement, used under switch, is one of C++ jump statements. When a break statement is encountered in a switch statement, programme execution jumps to the line of code following the switch statement i.e., outside the body of switch statement.

For example, Programme to input number of week's day (1-7) and translate it to its equivalent name. (e.g. 1 to Sunday, 2 to Monday ——)

```
#include <iostream.h>
int main ( )
{ int dow;
cout << "Enter number of week's day (1-7):";
cin >> dow;
switch (dow)
{
      case 1: cout<<"\n" << "Sunday";
      break
      case 2: cout<<"\n" << "Monday";
      break;
      case 3: cout<<"\n" << "Tuesday";
      break;
      case 4: cout<<"\n" << Wednesday";
      break;
      case 5: cout<<"\n" << "Thursday";
      break;
```

```
            case 6: cout<<"\n" << "Friday";
            break;
            case 7: cout<<"\n" << "Saturday";
            break;
            default: cout<<"\n" << "wrong number of day";
        }
    return 0;
    }
```

2.3 LOOP STATEMENT

The loop statements allow a set of instructions to be repeatedly executed until a certain condition is fulfilled. The loop statements are also called iteration statements. C++ provides three kinds of loops: **for loop, while loop, and do-while loop.**

All three loops constructs of C++ repeat a set of statements as long as a specified condition remains true. This specified condition is generally referred to as a loop statement, a true condition is any non-zero value. A zero value indicates a false statement.

2.3.1 for loop

The *for* loop includes an expression that specifies an initial value for an index, another expression that determines whether or not the loop is continued, and a third expression that allows the index to be modified at the end of each pass.

The general form of the for loop is

for (expression-1; expression-2; expression-3) statement;

where expression-1 is used to initialize some parameter (called an index) that controls the looping action, expression-2 represents a condition that must be true for the loop to continue execution, and expression-3 is used to alter the value of the parameter initially assigned by expression-1. Typically, expression-1 is an assignment expression, expression-2 is a logical expression and expression-3 is a unary expression or an assignment expression.

When the *for* statement is executed, expression-2 is evaluated and tested at the beginning of each pass through the loop, and expression-3 is evaluated at the end of each pass. Thus, the for statement is equivalent to

```
Expression-1;
while (expression-2)
{
    statement
    expression-3;
}
```

The looping action will continue as long as the value of expression-2 is not zero, that is as long as the logical condition represented by expression-2 is true.

The programme will display the consecution duals using for loop.

```
#include<stdio.h>
main ( )                  /*  display the numbers 0 through 9*/
{
      int digit
      for ( digit = 0 , digit < = 9; + + digit)
            printf ( " % d\n" , digit);
}
```

The first line of the *for* statement contains three expression, enclosed in parentheses. The first expression assigns an initial value 0 to the integer variable digit; the second expression continues the looping action as long as the current value of digit does not exceed 9 at the beginning of each pass, and the third expression increases the value of digit by 1 at the end of each pass through the loop.

2.3.2 while loop

The while statement is used to carry out operations, in which a group of statements is executed repeatedly, till a condition is satisfied.

The general form of the while statement is:

while (expression) statement;

The statement will be executed repeatedly, as long as the expression is true (i.e., as long expression has a nonzero value). This statement can be simple or compound, through it is usually a compound statement. It must include some feature that eventually alters the value of the expression, thus providing a stopping condition *for* the loop.

Consecutive Integer Quantities

In this example, we try to display the consecutive digits 0,1,2, ... 9.

```
#include<iostream.h>
main ( )            // display the integers  0 through 9 //
{
      int digit =0;
      while ( digit < = 9)
      {
            cout<<digit<<endl;
            ++ digit;
      }
}
```

Initially, a digit is assigned a value of 0. The *while* loop then displays the current value digit, increase its value by 1 and then repeats the cycle, until the value of digit exceeds 9. The net effect is that the body of the loop will be repeated 10 times, resulting in 10 consecutive lines of output. Each line will contain a successive integer value, beginning and ending with 9. Thus, when the programme is executed, the following output will be generated.

```
0

1

2

3

4

5

6

7

8

9
```

This programme can also be written as:

```
#include<iostream.h>

main( )      / * display the integers 0 through 9 */

{

      int digit = 0;

      while (digit < =  9) {

            cout<<digit<<endl;

            digit + = 5; }

}
```

when executed, this programme will generate the same output as the first programme.

The following code generates the Fibonacci series between 1 and 100. In this series, each number is the sum of its two preceding numbers. The series starts with 1.

```
#include<iostream.h>

int main( )

{

      int numl = 1,  num2 = 1;

      cout << numl << endl;

      while ( num2 < 100)

      {

            cout << num2 << endl;
```