Lesson 2: C++ Basics

Objectives

After reading this chapter you will understand:

- About basic terms in C++
- The different data types in C++
- The different types of operators and expressions
- The control structures in C++, Conditional branching, looping and unconditional branching

Structure of the lesson

2.1 Basic terms in C++

Token : The smallest individual units in a program are known as tokens. C++ tokens are

- Keywords
- Identifiers
- Constants
- Strings
- Operators

Keywords have a special meaning in the language. They are reserved identifiers and cannot be used as names for program variables or user-defined variables.
e.g.: int, float, throw, switch ,.. etc.

Identifiers refer to the names of variables, functions, arrays, classes, etc., created by the programmer.

Rules for naming the identifiers :
- Only alphabetic characters, digits and underscores are permitted.
- Name cannot start with a digit.
- Uppercase and lowercase are different(case-sensitive).
- Keyword cannot be declared as variable name.

Constants refer to fixed values that do not change during the execution. They include integers, character, floating point constants and strings.
e.g.:

```
123         //decimal integer
12.34       //floating integer
37          //octal integer
ox2         //hexadecimal integer
“C++”       //string constant
‘B’         //character constant
```

2.2 Data types

The data types can be categorized into 3. They are
- Basic or Fundamental data types
- Derived data types and
- User_defined data types.

2.2.1 Basic Data Types

Basic data types are again divided into numeric and non- numeric data types. There are six numeric data types and two non- numeric data types. Out of the six numeric types, three are integer types and three are floating types.

The various integer datatypes are short, int and long . The various floating point data types are float, double and long double. The sizes and the ranges of these data types are follows:

Type name	Memory used	Range	Precision
short (short int)	2	-32768 to 32767	not applicable
int	4	-2,147,483,648 to – 2,147,483,647	not applicable
long(long int)	4	-2,147,483,648 to – 2,147,483,647	not applicable
float	4	10E-38 to 10E18 (approximately)	7 digits
double	8	10E-308 to 10E308 (approximately)	15 digits
long double	10	10E-4932 to 10E4932 (approximately)	19 digits

Note: Precision refers to the number of meaningful digits, including digits in front of the decimal point.

C++ supports 2 non-numeric data types. They are **char** and **bool**. A variable of type char can hold any single character from the keyboard. The value that is stored in a variable of type char are placed within single quotes. One byte of memory is needed for a char type data to be stored in the memory. The data type can be of signed or unsigned char.

```
e.g:  char c1 = ‘A’;
```

A new data type called **bool** has been included in C++. It returns a boolean value, true or false with default values 1 and 0.

```
e.g: bool x,y;
     x = true;
```

2.2.2 User Defined Data Types

The user defined data types are structures, unions, class and enumeration.

Structures And Unions: Structures and unions are same as in C. Structures provide a method for packing together data of different types which are logically related.

Eg:

```
struct student
{ char name[20];
int rno;
float tmarks;
};
struct student s1;
```

The keyword **struct** declares student as a new data type that can hold three fields of different data types. These fields are known as **structure members** or **structure elements**. The structure name, student can be used to create variables of type student. s1 is a variable of type student that has 3 member variables.The member variables can be accessed using the dot or period operators.

```
strcpy(s1.name,”John”);
 s1.rno =  999;
```

Class: A class is a way to bind the data and its associated functions together. It allows data and functions to be hidden if necessary, from external use. Generally , the class specification has two parts.
They are

- **Class Declarations:** It describes the type and scope of its members.
- **Class Function Definitions:** It describes how the class functions are implemented.

General Form of the Class Declaration is:

```
class classname
{
          private:
            variable declarations;
            function declarations;
          public:
            variable declarations;
            function declarations;
};
```

The keyword **class** specifies that what follows is an abstract data type of type classname .The body of the class is enclosed within braces and terminated by a semicolon. The class body contains the declarations of variables or data members and functions or member functions. These functions and variables together called class members. They are grouped under sections as **private** and **public**, which denotes the visibility of the members. The private members can be accessed only within the class and the public members can be accessed from outside the class.
e.g.:

```
class item
          {
          int numb;
          float cost;
          public:
              void getdata(int a,float b);
              void  putdata(void);
          };
```

Once the class is declared, we create objects(variables) of that type by using the classname.

e.g.: item x;
Here, x is an object of the class of item.

Enumerated Data Type: It is a user-defined datatype which provides a way for attaching names to numbers. The keyword **enum** automatically enumerates a list of words by assigning them values 0,1,2, and so on.

e.g.: enum shape{ circle,rectangle,triangle};
Here, circle is assigned an int value 0,rectangle to 1 and triangle to 2.

2.2.3 Derived Data Types

The derived data types are arrays , functions and pointers. We will be discussing about arrays and functions in the next lessons. Pointers are the variables, which directly refer to the value stored in the address it points to.

```
e.g.: int *ip;      //pointer to an integer
      ip = &y;      //address of x assigned to ip
     *ip = 10;
```

2.3 Operators And Expressions

C++ has a rich set of operators. All C operators are valid in C++ also . In addition, C++ introduces some new operators . An expression is a combination of operators, constants and variables arranged as per the rules of the language. The different operators and expressions are studied in this section.

2.3.1 Operators

The operators are used to manipulate the data during the processing. The different types of operators are:

- Arithmetic operators
  There are five arithmetic operators in C++. They are

Operator	**Purpose**
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	modulus

Syntax:

operand1 operator operand2

Every arithmetic operation returns a numeric value depending on the type of the operands. For the % operator and / operator, the second operator must be non-zero.

Unary Operators: C++ includes a class of operators that act upon a single operand to produce a new value. Other than unary + and unary -, there are two other operators ++ increment,-- decrement operator . These operators can be prefixed(written before) or post fixed(written after) to the operand.

The increment operator causes its operand to be increased by one and the decrement operator causes its operand to be decreased by one. If the operator precedes the operand, then the operand will be altered before it is utilizes called as pre-increment or pre-decrement. If the operator follows the operand, then the value of the operand will be altered after it is utilized called post increment or post decrement.

Bitwise Operators : There are some bit wise operators for the manipulation of data at the bit level. These operators are used for testing or shifting the bits left or right. They may not be applied for float or double values. The bit wise operators and their respective meaning are as follows:

Operator	**Meaning**
&	bit wise AND
|	bit wise OR
^	bit wise XOR
<<	shift left
>>	shift right
~	one’s complement

Relational Operators: C++ supports various relational operators to compare one or more identifiers. The relational operators are:

Operator	**Meaning**
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to

The result of these expressions using relational operators will be boolean values, true represented by 1 or false represented by 0.

Logical Operators: The logical operators are applied between operand or relational expressions resulting in Boolean values true(1) or false(0). The logical operators in C++ are

Operator	**Meaning**
&&	logical AND
||	logical OR
!	logical NOT

The logical expression yields a value one or zero, depending on the values of the operators and operands according to the truth table given below:

Truth table for And and OR operators:

Op1	**Op1**	**Op1 && Op2**	**Op1 || Op2**
Non zero	Non zero	1	1
Non zero	0	0	1
0	Non zero	0	1
0	0	0	0

Truth table for NOT operator:

Op	**!Op**
0	1
1	0

Conditional Operators: An operation that makes use of conditional operators(?:) is known as conditional expression. These operations are also known as tertiary operators.

Syntax:

```
expr1?expr2:expr3
```

where expr1 is generally a conditional expression and is evaluated first. If it is non-zero then expr2 is evaluated else expr 3 is evaluated.

e.g.:

```
big =(a>b)?a:b;
```

Some other operators that are included in C++ are:

operators	symbols
stream output operator	<<
stream input operator	>>
Scope resolution operator	::
dynamic memory delete operator	delete
dynamic memory allocation operator	new
pointer-to-member operators	::* , ->* and .*

We have already studied about stream input and output operators in the previous chapter. We will study the new and delete operators in the next lessons. Now, we will study about the remaining operators.

Scope Resolution Operator: In C++, blocks and scope are used to construct the program. The scope of the variable extends from the point of declaration till the end of the block i.e., between {}. The same variable names can be used in different blocks to have different meanings. A variable declared inside the block is said to be local to that block. The global version of a variable cannot be accessed from within the inner block. So, a scope resolution operator :: is used , where it allows the access to the global version of a variable.

Syntax:

```
:: variable name
```

Program to demonstrate scope resolution operator

```
#include<iostream.h>
int x = 50;
void main()
{
int x = 10;
  {
        int x = 1;
        cout<<”x = “<<x<<”\n”; //prints x value  local to the scope
        cout<<”::x= “<<x<<”\n”; //prints global x value
  }     cout<<”x = “<<x<<”\n”; //prints x value  local to the scope
    cout<<”::x= “<<x<<”\n”; //prints global x value
}
```

Output:

```
x=1
x=50
x=10
x=50
```

Referencing And De-Referencing Operators: The address operator **&** or referencing operator assigns the address of the operand on the right side to the pointer variable to its left.The indirection operator * or de-referencing operator accesses the value of the operand pointed to by the pointer variable.

```
e.g.:   int x = 10,y;
        int *p;
p = &x; //p stores the address of the location where x is stored
y= *p; //*p has the value of the variable to which p points to
cout<<"The value pointed by p is " <<y;
```

Output: The value pointed by p is 10

sizeof Operator: The sizeof operator gives the amount of storage required to store an identifier.

```
        int k;
        cout<<sizeof(k);
```

Output: 2

Typecast Operator:C++ permits explicit type conversion of variables or expressions using the type cast operator.

Syntax:

```
        Typename(expression)
        Average = sum/float(i);
```

2.3.2 Precedence Of Operators

Operators in the same box have the same precedence. Operators in higher boxes have higher precedence. Unary operators and assignment operators are executed from right to left and have the same precedence.

Other operators that have the same precedence are executed from left to right. The list is given from higher precedence to the lower precedence.

Precedence of operators are as follows:

::	scope resolution operator
.	dot operator
➔	member selection
[]	array indexing
()	function call
++	postfix increment operator
--	postfix decrement operator
++	prefix increment operator
--	prefix decrement operator
!	not
-	unary minus
+	unary plus
*	dereference
&	address of
new	
delete	
delete[]	
sizeof	
*	multiply
/	divide
%;	remainder(modulo)
+	addition
-;	subtraction
<<	insertion operator(output)
>>;	extraction operator
<	less than
<=	less than or equal
>	greater than
>=;	greater than or equal
==	equal
!=;	not equal
&&;	and

|| ;	or
=	assignment
%=	modulo and assign
+=	add and assign
-=	subtract and assign
*=	multiply and assign
/=;	divide and assign

2.3.3 Expressions

Expressions combine operands, operators and constants to produce a single value. There are different types of expressions like:

Constant Expression: They can have only constants values.

```
e.g.:10
     15+6/4.0
      y'
```

Integral Expressions: These expressions produce integer result after implementing all automatic and explicit type conversions.

```
e.g.:X
     X * 'a'
     5 + int(2.0)
    where X is an integer
```

Float Expressions: These expressions produce floating point results after implementing all automatic and explicit type conversions.

Eg:

```
          X
          X * y/10
          5 + float(2)
    where x and y are floating point values.
```

Pointer Expressions: These produce address values.

```
e.g.: &x
      ptr
      ptr + 1
where x is a variable and ptr is a pointer
```

Relational Expression: These expressions results in a bool type values, true or false.

```
e.g.: x<y
      a+b>100
```

Logical Expressions: These combine two or more relational expressions using logical operators and result a bool type values.

```
e.g.:  i<j && y == 5
       p == 3 || q >5
```

Bitwise Expressions: Bitwise expressions are used to manipulate data at bit level.

```
e.g.:x<<3;//shifts 3 bits to its left
```

Special Assignment Expressions: Chained assignment:

```
A = (b = 5);
      Or
A = b= 5;
```

First 5 is assigned to b, then to a.
A chained statement cannot be used to initialize variables at the time of declaration.

Embedded Assignment:

```
x = (y = 20) + 10;
```

(y = 20) is an assignment expression known as embedded assignment. The value of 20 is assigned to y and then the result 20+10 is assigned to x.

Compound Assignment: Compound assignment operator is a combination of assignment operator with a binary arithmetic operator.

```
p = p + 10;
can be written as
p+=10;
```

2.4 Control Structures

In high-level programming languages, flow of program execution may be changed using certain control statements called control structures.

A control structure is a **control flow statement** that allows you to alter the **sequential flow**.

Control flow statements fall into three categories:

1. Conditional branching (or) Decision Making or Non-iterative
2. Looping or iterative or repetitive
3. Unconditional branching.

2.4.1 Conditional Branching

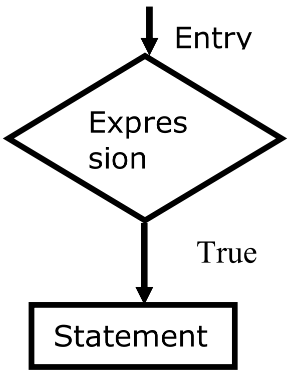
Conditional branching is the most basic control feature of any programming language. It enables a program to make decisions, to decide whether or not to execute a statement or a block of statements based on the value of an expression. The expression may result in either true or false value. Since the value of the expression may vary from one execution to another, this feature allows a program to react dynamically to different data.

C supports various types of conditional branching statements. The following categories illustrate several conditional control structures.

- **Simple if**
- **if ..else**
- **else if ladder**
- **Nested if**
- **Switch**

Simple If: The **simple if** statement is wonderful decision making statement and is used to control the flow of execution of a single or multiple instructions.

The general form of **"simple if"** follows:
If (condition/expression) Statement;
In this statement the given condition is tested first and responds accordingly. If the result of expression is true then the given statement is executed. If the result is false the statement cannot be executed.



When multiple statements are to be executed using if control structure then it may be referred as compound if.

Syntax:

```
if (expression)
{
        statement-block;
}
statement-x;
```

The statement-block may be a single statement or a group of statements. If the expression is true statement-block will be executed, other wise the statement-block will be skipped and the execution will jump to the statement-x.

Program to find biggest of two numbers.

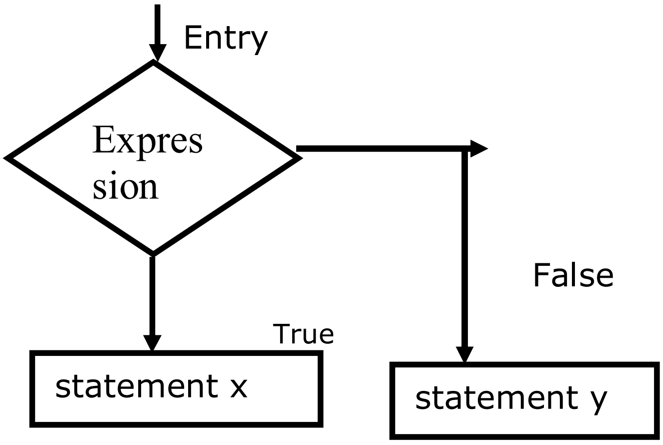
```
#include<iostream.h>
void main()
{
        int a, b;
        cout<<"\n\t Enter A value              : ";
        cin>>a;
        cout<<"\n\t Enter B value              : ";
        cin>>b;
        if (a>b)
                cout<<"\n”<<a<<” is Greater than “<<b;
        if (b>a)
                cout<<"\n”<<b<<” is Greater than “ <<a;
}
```

If_else: In **if-else** control statement there exists an extension of the simple if statement. It allows the user to perform another block of statements in case the condition result is false.

syntax :

```
if (expression)
        statement-x;
else
        statement-y ;
```

Here the **expression** is evaluated; if the result of the expression is a true then statement-x is executed otherwise statement-y will be executed.



Flow graph

Program to check whether given number is even or odd

```
#include<iostream.h>
void main()
{
        int n;

        cout<<"\n Enter a number..:";
        cin>>n;
        if (n%2==0)
                cout<<"\n Given  number is even":
        else
                cout<<"\n Given  number is odd":

}
```

else-if Ladder: In **else..if ladder** number of conditions are checked depending on the falsity of the previous condition. Literally, too many conditions are evaluated in **if**..**else** ladder.

Syntax:

```
If <condition1>
{
------
}
else if <condition2>
{
------ True block 1
}
else
{
------ False block
}
```

In this, condition1 is checked and if it is true then its corresponding condition is executed. If the condition is false then next condition is verified. If all the given conditions are false then false block is executed. Only one of all the available blocks gets executed. After the execution of any one of the blocks, control is transferred to next statement after the construct.

Program to find biggest of three numbers

```
#include<iostream.h>
void main()
{
        int a,b,c;
        clrscr();
        cout<<“enter three numbers:”;
        cin>>a>>b>>c;
        if(a>b)
                if(a>c)
                        cout<<a<<” is big”;
                else
                        cout<<C<<”is big”;
        else if(b>c)
                cout<<b<<” is big”;
        else
                cout<<c<< “ is big “;
}
```

Decision Making With Nested If: A **nested if** control structure consists of multiple **if** statements in one another. Here each **if** statement consists of subsequent branching statement. Literally a nested **if** consists of one **if** statement in another **if** statement. It is used when multiple conditions are to be evaluated.

Syntax:

```
        if(expression)
    {
        if(expression)
        {
            if(expression)
            {

                ---
                ---
```

Here evaluations of expressions or conditions are based on the first condition. If the first condition itself is false, then there is no way of evaluating other conditions. At any level of expression the program control may be altered.

Ex: Program Biggest of 3 numbers using nested if

```
    #include<iostream.h>
    void main()
    {
        int a,b,c,big;
        cout<<"\n Enter the value of a  :   ";
        cin>>a;
        cout<<"\n Enter the value of b  :   ";
        cin>>b;
        cout>>"\n Enter the value of c  :   ";
        cin>>c;
        if (a>b)
            if (a>c)
                big = a;
            else
                big = c;
        else
            if (b>c)
                big = b;
            else
                big = c;
        cout<<"\nBiggest of three numbers is:"<<big;
```

```
}
```

switch: C provides a special kind of conditional control structure that acts as an alternative to **if..else ladder**. When there are more conditions or paths in a program, **if-else** branching can become more difficult. In such situations **switch** may act better. The **switch** statement allows the user to specify an unlimited number of execution paths based on the value of a single expression. Each execution path is referred as a case.

However, all the cases should be unique. Each case must be terminated by a '**break**' statement. The '**default'** case is not mandatory.

In a **switch** statement, there are four different keywords to be used:

- **switch**
- **case**
- **break**
- **default**

Though the **switch** control structure enables the user to improve clarity of the program, it causes more errors. So, it requires more attention while implementation.

Syntax:

```
switch(expression)
        {
        case  value1:
                        statement;
                         break;
        case  value2:
                        statement;
                         break;
        :
        :
        :
        default :
                       statement;
        }
```

Among all the cases, only one case can be executed successfully because each case is terminated by a '**break'** statement.

Program to accept two integer values and perform arithmetic operation by getting the user input.

1) Addition 2) Subtraction
3) Multiplication 4) Division
5) Exit .

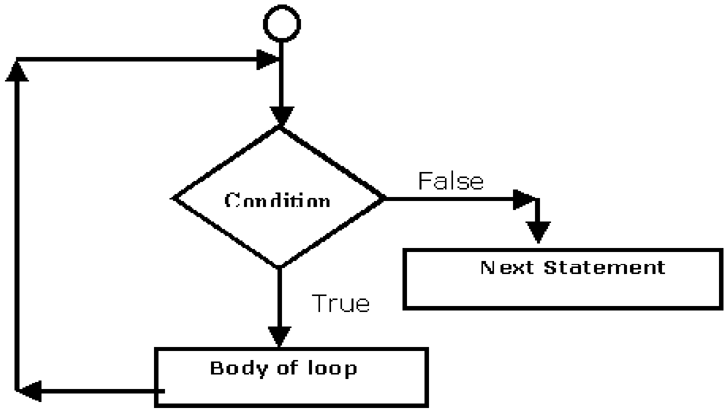
```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
        int a, b, c, ch;
        clrscr();
        cout<<"\n\t\t\t Enter two numbers      :        ";
        cin>>a>>b;
        cout<<“Enter your choice:”:l
        cout<<“1)Addition\n2)Subtraction”;
        cout<<“\n3)Multiplication”;
        cout<<”\n4) Division. \n5) Exit”.
        cin>>scanf(“%d”&ch);
        switch (ch)
        {
        case 1:
                c = a + b;
                break;
        case 2:
                c = a - b;
                break;
        case 3:
                c = a * b;
                break;
        case 4:
                c = a / b;
                break;
        default :
                cout<<"\n Invalid option  ";
                exit(0);
        }
        cout<<"\n\t\t\t Result   :”,c;
}
```

2.4.2 Looping Structures

Some times, in a program, a statement or a block of statements need to be executed repeated number of times. In such situations decision control structures may not be useful, as they do not transfer the control back. Hence the user may require another form of control structures, which perform a group of instructions for a fixed number of times. Such control structures are named as looping control structures. C language provides three different iterative or looping structures.

- **while** loop
- **do…while** loop
- **for** loop

While: The **while** control structure executes a single or multiple statements for repeated number of times based on a given condition. It executes the statements as long as the given condition or expression results in a true value. It terminates execution as and when the condition is false.



Syntax:

```
initialization statement;
while(condition)
{
    --------------
    Condition reachable Statement;
}
```

Here the condition is tested every time, it executes the block of statements. The keyword **while** verifies the trueness and falsity of the expression and responds accordingly. If the condition is false for the first time the minimum number of iterations is 0 in **while** control structure. It requires three statements in order to perform repetitive tasks.

They are

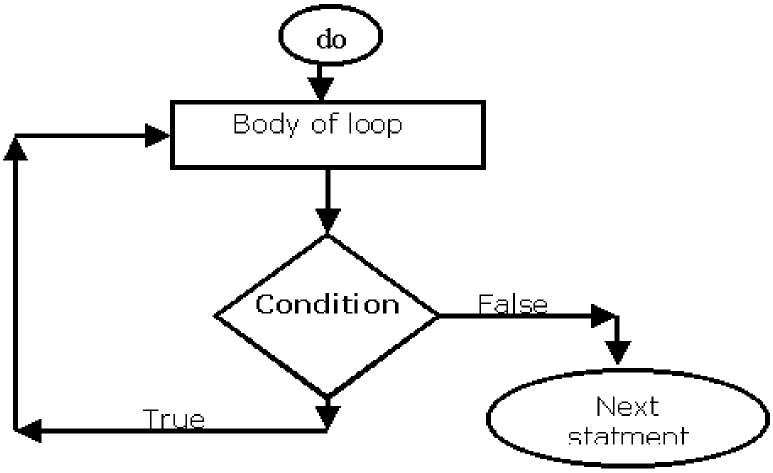
- Initialization statement
- Conditional statement
- Condition reachable statement

If any of the above statements is ignored then the **while** may not perform well.

Program to print the numbers from 1 to 10

```
#include<iostream.h>
void main()
{
  int i;
  i=1;
  while (i<=10)
    {
      cout<<”n"<< i;
      i++;
    }
}
```

Do- While: C provides another form of while control structure i.e., **do-while** control structure. In **do-while** control structure the statements in the block get executes first, later on the condition is evaluated. Hence the user can assume that the minimum number of iterations for **do-while** control structure as 1, even if the expression or condition results in false for the first time.



Syntax:

Initialization statement;
do
{

Condition reachable statement;
} ***while(condition);***

Here the statements in the loop will be executed until the given condition becomes false. The while statement should be terminated by a semicolon (;) in **do while**.

Print the numbers from 1 to 10.

```
#include <iostream.h>
void main()
{
int i;
i=1;
do
{
    cout<< i++<<"\n";
} while (i<=10);
}
```

For Loop: C provides a more flexible form of looping control structure that improves clarity of the code. It is nothing but **for** control structure. Usually the **for** control statement is used to perform fixed number of iterations.

The major difference between **for** and **other looping structures** is the number of iterations. In case of **while** and **do-while** the number of iterations are indefinite. The user may not predict the number of iterations. On the other hand **for** specifies the number of iterations in the statement itself.

Syntax:

for *(initialization; test condition; increment/decrement part)*

```
{
        Body of the loop;
}
```

The initialization may contain single or multiple assignment statements. A control variable is involved in this part of statements. The test condition verifies the validity of the control variable for each iteration. Increment or decrement part, increments or decrements the value of the control variable in order to reach the test condition.

Program to print the numbers from 1 to 10.

```
#include <iostream.h>
void main()
{
  int i;
  for (i=1 ; i<=10; i++)
    cout<< I<<"\n";
}
```

Break and **Continue** statements:

Break: This statement takes control out of the **switch** statement or loop structure. In other words, a **break** statement takes the control out of the current block in execution. The control is transferred to the statement that follows the block.

Syntax:

break;

Continue Statement : To skip a part of the body of the loop in execution on certain condition and for the loop to be continued for the next iteration **continue** statement is used.

Syntax:

continue;

2.4.3 Unconditional Branching

goto and label: C++ supports an unconditional branching statement called **goto**. This **goto** is meant for transferring control from one part of the program to another part a label is present. A label is a user-defined word to where the control is supposed to be transferred. The given label must reside in the same function and can appear before only one statement in the same function. Although it may not be preferable to use the **goto** statement in a highly structured language like C, there may be occasions where the use of **goto** is desirable.

Syntax:

```
goto label:
----------------
----------------
label:
statement;
```

```
label:
statement;
-----------------
-----------------
goto label:
```

example demonstrates the **goto** statement:

```
void main()
   {
       int x = 1;
       abc:
            cout<<x;
            x++;
            if(x <= 5 )
              goto abc;
    }
```

2.5 Summary

- The basic terms in C++ like token, keyword, identifier, constants are studied here.
- The data types in C++: basic, user defined and derived data types are discussed in detail.
- The different types of operators and expressions are also discussed in this lesson.
- The different control structures in C++ like conditional, looping and unconditional statements are studied. The conditional statements: if, if-else, nested If and switch are studied in detail. Also focus is made on three categories of loops, available in C++ language: **while**, **do–while** and **for** loop. Usage of break, continue, **goto** and exit statements, which are very useful in loops have been covered.

2.6 Technical Terms

Expression: It is a combination of operators, constants and variables arranged as per the rules of the language.

Operator: A symbol that represents an action to be performed.

Manipulator: A data object that is used with stream operators. It causes a specific operation to be performed on the stream.

Scope resolution operator: The operator that is usually used to indicate the class in which the identifier is declared.

Type casting: To convert a variable from one type to another type by explicitly.

Union: A data type that allows different data types to be assigned to the same storage location.

2.7 Model Questions

1. Explain the different data types in C++ ?
2. What are the different operators in C++ ?Explain.
3. Explain the precedence of operators.
4. How many types of expressions are there? What are they ? Explain them ?
5. Explain the different control structures in C++ with example ?

2.8 References

Object-oriented programming with C++
by E.Bala Gurusamy

Problem solving with C++
by Walter Savitch

Mastering C++
by K.R.Venugopal, Rajkumar Buyya, T.Ravi Shankar

AUTHOR:

M. NIRUPAMA BHAT, MCA., M.Phil.,
Lecturer
Dept. Of Computer Science
JKC College
GUNTUR.