OBJECT ORIENTED PROGRAMMING USING C++

SYLLABUS

UNIT I

Data types, operators and statements: Identifiers and Keywords – constants C++ Operators – Type Conversion, Declaration of variables – Statements – Simple C++ programs – Manipulator functions – input and Output (I/O) Stream flags, Control Statements, Conditional expressions – Switch Statements – Loop Statements – Breaking Control Statements.

UNIT II

Functions and Program Structures: Defining a function – Types of functions – Actual and formal arguments – Local and global variables – Default arguments – Multifunction program – Storage class specifiers – Recursive function – Preprocessors – Header files – Standard functions – Arrays and Functions – Multidimensional Arrays.

Pointers: Declaration – Pointer Arithmetic – Pointers and Functions – Pointers and Arrays – Pointers and Strings – Pointers to Pointers.

UNIT III

Structures, Unions and Bit Fields – Nested Structure – Unions – Bit fields – Enumerations – Class and Objects: Declaration of class – Member functions – Defining and object of a class – Assessing a member of class – Arrays of class objects – Pointers and classes – Unions and classes – Constructors – Destructions – Inline member functions – Static class members – Friends functions – Dynamic Memory Allocations.

UNIT IV

Inheritance – Single inheritance – Types of Base Classes – Types of derivations – Ambiguity in single inheritance – Multiple inheritance – Container classes – Overloading – Function overloading – Operator overloading – Overloading of binary operators – Overloading of unary operators.

UNIT V

Polymorphism – Polymorphism with pointers – Virtual functions – Late binding abstract classes – Constructor under inheritance – Destructors under inheritance – Virtual destructors – Virtual base classes – Templates and exception handling – Function template – Class template – Exception handling – Data file Operations – Opening and closing of files – Stream state member functions – Reading/writing a character from a file – Binary file operations – Classes and file operations – Array of class objects and fill operations – Nested classes and file operations – Random Access File Processing.

UNIT I

LESSON

1

DATA TYPES, OPERATORS AND STATEMENTS

CONTENTS

Contd...

1.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Explain the data types, operators and statements
- Discuss how to identify identifiers and keywords
- Define constants
- Identify and explain the C++ operators
- Discuss the type conversion
- Describe the declaration of variables
- Explain the concept of statements
- Analyse a simple C++ programme
- Explain the manipulator functions
- Discuss the input and output (I/O) stream flags

1.1 INTRODUCTION

C++ is a language in essence. It is made up of letters, words, sentences, and constructs just like English language is. This lesson discusses these elements of the C++ language. You will now be familiar with various data types and operators used in C++. In this lesson you will learn about the variables and its declaration in a C++ programme. Manipulators are used to change the type. cout and cin are the basic I/O operators.

1.2 DATA TYPES

1.2.1 Basic Data Types

Data types in C++ can be classified under various categories as shown in Figure 1.1.

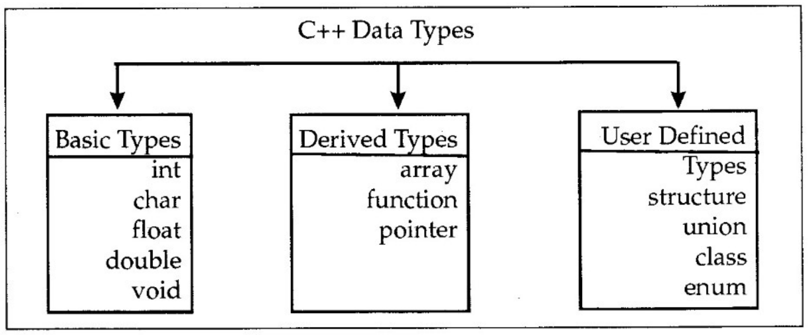

Figure 1.1: Hierarchy of C++ Data Types

Both C and C++ compilers support all the built-in (also known as basic or fundamental or standard) data types. With the exception of void, the basic data types may have several modifiers preceding them to serve the needs of various situations. The modifiers signed, unsigned, long, and short may be applied to character and integer basic data types. However, the modifier long may also be applied to double. Table 1.1 lists all combinations of the basic data types and modifiers along with their size and range.

Table 1.1: Size and Range of C++ Basic Data Types

Type	Bytes	Range
char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed int	2	32768 to 32767
short int	2	-32768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	-32768 to 32767
long int	4	-2147483648 to 2147483647
signed long int	4	2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
float	4	3.AE - 38 to 3.4E + 38
double	8	1.7E - 308 to 1.7E + 308
long double	10	3.4E - 4932 to 1.1E + 4932

The type *void* was introduced in ANSI C. Two normal uses of *void* are:

(1) to specify the return type of a function when it is not returning any value, and

(2) to indicate an empty argument list to a function. For example,

```
void funct1(void);
```

Another interesting use of *void* is in the declaration of generic pointers. For example,

```
void*gp;              //gp becomes generic pointer
```

A generic pointer can be assigned a pointer value of any basic data type. But it may not be deferenced. For example,

```
int *ip;              // pointer
gp = ip;              // assign int pointer to void pointer
```

are valid statements. But, the statement,

```
*ip = *gp;
```

is illegal. It would not make sense to dereference a pointer to a *void* value.

Assigning any pointer type to a *void* pointer without using a cast is allowed in both C++ and ANSI C. In ANSI C, we can also assign a *void* pointer to a non-pointer without using a cast to non-void pointer type. This is not allowed in C++. For example,

```
void * ptr1;
char * ptr2;
ptr2 = ptr1;
```

are all valid statements in ANSI C but not in C++. A void pointer cannot be directly assigned to other type pointers in C++.

1.2.2 User Defined Data Types

C++ also permits programmers to create their own data types using basic data types to suite their requirements. Some of such user-defined data types are discussed below.

Structures and Classes

The C struct and union data types are still legal in C++, some more features have been added to make them suitable for object-oriented programming. C++ also permits us to define another user-defined data type known as class, which can be used, just like any other basic data type, to declare variables. The class variables are known as objects, which are the central focus of object-oriented programming.

1.2.3 Enumerated Data Type

An enumerated data type is another user-defined type which provides a way for attaching names to numbers, thereby increasing comprehensibility of the code. The enum keyword (from C) automatically enumerates a list of words by assigning them values 0, 1, 2, and so on. This facility provides an alternative means for creating symbolic constants. The syntax of an enum statement is similar to that of the struct statement.

Examples:

```
enum shape {circle, square, triangle};

enum color {red, blue, green, yellow};

enum position {off, on};
```

The enumerated data types differ slightly in C++ when compared with those in ANSI C. In C++, the tag names shape, color, and position become new type names. That means we can declare new variables using these tag names. Examples:

```
shape ellipse;                //ellipse is of type shape

color background;             // background is of type color
```

ANSI C defines the types of enums to be ints. In C++, each enumerated data type retains its own separate type. This means that C++ does not permit an int value to be automatically converted to an enum value. Examples:

```
color backgrounds blue;       //allowed

color background = 7;         // Error in C++

color background = (color) 7; // OK
```

However, an enumerated value can be used in place of an int value.

```
int c " red;                  // valid, color type promoted to int
```

By default, the enumerators are assigned integer values starting with 0 for the first enumerator, 1 for the second, and so on. We can override the default by explicitly assigning integer values to the enumerators. For example,

```
enum color {red, blue = 4, green = 8};

enum color {red s 5, blue, green};
```

are valid definitions. In the first case, red is 0 by default. In the second case, blue is 6 and green is 7. Note that the subsequent initialized enumerators are larger by one than their predecessors.

C++ also permits the creation of anonymous enums (i.e., enums without tag names). Example:

```
enum {off, on};
```

Here, off is 0 and on is 1. These constants may be referenced in the same manner as regular constants. Examples:

```
int switch_1 = off;

int switch_2 = on;
```

In practice, enumeration is used to define symbolic constants for a switch statement. An example to illustrate the point is presented here. You may not understand some of the parts of the programme. You will learn more in the sections to come later.

```
enum shape
{
        circle,
        rectangle,
        triangle
);
        main( )
{
        cout << "Enter shape code:";
        int code;
        cin>> code;
        while (code >= circle && code <= triangle)
{
        switch(code)
{
        case circle:
        ....
        ....
        break;.
        case rectangle
        ....
        ....
        break;
        case triangle:
        ....
        ...
        break;
}
        cout << "Enter shape code:"
        cin>> code;
}
        cout << "BYE\n";
```

ANSI C permits an *enum* to be defined within a structure or a class, but the *enum* is globally visible. In C++, an *enum* defined within a class (or structure) is local to that class (or structure) only.

1.2.4 Derived Data Types

In addition to user-defined data types, C++ also allows programmers to derive complex data types using simple basic data types. These data types are referred to as derived data types. We will discuss some commonly used data types in brief in the following section.

Arrays

The application of arrays in C++ is similar to that in C. The only exception is the way character arrays are initialized. When initializing a character array in ANSI C, the compiler will allow us to declare the array size as the exact length of the string constant. For instance,

char string[3] = "xyz";

is valid in ANSI C. It assumes that the programmer intends to leave out the null character \0 in the definition. But in C++, the size should be one larger than the number of characters in the string.

char string[3] " "xyz"; // O.K. for C++

Functions

Functions have undergone major changes in C++. While some of these changes are simple, others require a new way of thinking when organizing our programmes. Many of these modifications and improvements were driven by the requirements of the object-oriented concept of C++. Some of these were introduced to make the C++ programme more reliable and readable.

Pointer

Pointers are declared and initialized as in C. Examples:

```
int * ip;          // Int pointer
ip = &x;           // address of x assigned to ip
*ip = 10;          //50 assigned to x through indirection
```

C++ adds the concept of constant pointer and pointer to a constant.

```
char * const ptrl = "GOOD";        // constant pointer
```

i.e., cannot modify the address that ptrl is initialized to.

```
int const * ptr2 = &m;             // pointer to a constant
```

ptr2 is declared as pointer to a constant. It can point to any variable of correct type, but the contents what it points to cannot be changed.

We can also declare both the pointer and the variable as constants in the following way:

const char * const cp = "xyz";

This statement declares cp as a constant pointer to the string, which has been declared a constant. In this case, neither the address assigned to the pointer cp nor the contents it points to can be changed.

Pointers are extensively used in C++ for memory management and achieving polymorphism.

The Const Keyword

The keyword, const (for constant), precedes the data type of a variable.

Syntax

const <data type> <name of variable> = <value>;

For example,

```
const float g = 9.8;
cout << "The acceleration due to gravity is" << g << "m/s2";
g=g+4; // ERROR!! Const variables cannot be modified.
```

The const keyword specifies that the values of the variable will not change throughout the programme. This prevents the programmer from changing the value of variable, like the one in the example given above. If the keyword, const, has been used while defining the variable, the compiler will report an error if the programmer tries to modify it.

1.3 IDENTIFIERS

Identifiers refer to the names of variables, functions, arrays, classes, etc., created by the programmer. They are the fundamental requirement of any language. Each language has its own rules for naming these identifiers. The following rules are common to both C and C++:

- Only alphabetic characters, digits and underscores are permitted.
- The name cannot start with a digit.
- Uppercase and lowercase letters are distinct.
- A declared keyword cannot be used as a variable name.

A major difference between C and C++ is the limit on the length of a name. While ANSI C recognizes only the first 32 characters in a name, C++ places no limit on its length and, therefore, all the characters in a name are significant.

Care should be exercised while naming a variable that is being shared by more than one file containing C and C++ programmes. Some operating systems impose a restriction on the length of such a variable name.

1.4 KEYWORDS

The keywords implement specific C++ language features. They are explicitly reserved identifiers and cannot be used as names for the programme variables or other user-defined programme elements. The keywords not found in ANSI C are shown boldface. These keywords have been added to the ANSI C keywords in order to enhance its features making it an object-oriented language. The C++ keywords are listed below.

asm	continue	float	new	signed	try
auto	default	for	operator	sizeof	typedef
break	delete	friend	private	static	union
case	do	goto	protected	struct	unsigned
catch	double	if	public	switch	virtual
char	else	inline	register	template	void
class	enum	integer	return	this	volatile
const	extern	long	short	throw	while

1.5 CONSTANTS

An identifier that represents a constant value throughout the life of the programme is known as constant (or literals). It allows programmers to attach meaningful names to data values and hence enhances the readability of the programmes. C++ allows several kinds of contents:

1. Integer constant
2. Character constant
3. Floating constant
4. String constant

1.5.1 Integer Constants

Integer constants are whole numbers without any fractional part. C++ allows three types of integer constants:

1. Decimal (base 10)
2. Octal (base 8)
3. Hexadecimal (base 16)

1. ***Decimal Integer Constants:*** An integer constant consisting of a sequence of digits is taken to be decimal integer constant unless it begins with 0 (digit zero). For instance, 1234, 41, +97, -17 are decimal integer constants.
2. ***Octal Integer Constants:*** A sequence of digits starting with 0 (digit zero) is taken to be on octal integer. For instance decimal integer 8 will be written as 010 as octal integer ($\because 8_{10} = 10_8$).
3. ***Hexadecimal Integer Constants:*** A sequence of digits preceded by 0x or 0x is taken to be hexadecimal integer. For example decimal 12 will be written as 0xc as hexadecimal integer.

An Integer constant must have atleast one digit and must not contain any decimal point. It may contain either + or – sign. A number with no sign is assumed to be positive. Commas cannot appear in an integer constant.

The suffix l or L and u or U attached to any constant forces it to be represented as a long and unsigned integer respectively.

1.5.2 Character Constants

A character constant is one character enclosed in single quotes, as in 'z'.

Single character constants e.g., 'c' or 'A' have type char which is a C++ data type for characters. The value of a single character constant is the numerical value of the character in the machine's character set. For instance, the value of 'c' will be 99, which is ASCII value of 'c'. Multi-character constants have type *int*. The value of a multi-character constant is implementation dependent.

C++ allows you to have certain *non graphic characters* in character constants. Non-graphic characters are those characters that cannot be typed directly from keyboard e.g., backspace, tabs, carriage return, etc. These nongraphic characters can be represented by using escape sequences. An escape sequence is represented by a back slash (\) followed by one or more characters. Table to illustrate a list of escape sequences is given below:

Table 1.2: Escape Sequences

Escape sequence	Non-graphic character
\a	Audible bell (alert)
\b	Backspace
\f	Formfeed
\n	New line or line feed
\r	Carriage Return
\t	Horizontal tab
\v	Vertical tab
\\	Back slash
\'	Single quote
\"	Double quote
\?	Question mark
\On	Octal number
\xHn	Hexadecimal number
\0	Null

An escape sequence represents a single character.

1.5.3 Floating Constants

Floating constants are also called real constants.

Real constants are numbers having fractional parts. These may be written in one of the two forms called *fractional form* or the *exponent form*.

A real constant in fraction form consists of signed or unsigned digits including a decimal point between digits.

The following are valid real constants in fractional form:

3.0, 18.4, -12.0, -0.00275

A real constant in exponent form consists of two parts: **Mantissa and exponent.** For instance 5.8 can be written as 0.58×10^1 = 0.58 E01 where 0.58 is the mantissa part and 1 is the exponent part.

1.5.4 String Constants

'Multiple character' constants are treated as string constants. A string constant (or literal) is a sequence of characters surrounded by double quotes.

Each string literal is by default (automatically) added with a special character '\o' which makes the end of a string. '\o' is a single character. Thus the size of a string is the number of characters plus one for this terminator. For instance "abc" is of size 4, "seema\'s pen" is of size 12'.

1.6 C++ OPERATORS

An expression is a combination of variables, constants and operators written according to some rules. An expression evaluates to a value that can be assigned to variables and can also be used wherever that value can be used.

Individual constant, variables, array elements function references can be joined together by various operators to form expressions. C++ includes a large number of operators, which fall into several different categories. In this lesson, we examine some of these categories in detail. Specifically, we will see how arithmetic operators, unary operators, relational and logical operators, assignment operators and the conditional operator are used to form expressions.

The data items that operators act upon are called operands. Some operators require two operands, while others act upon only one operand. Most operators allow the individual operands to be expressions. A few operators permit only single variables as operands.

Operators are used to compute and compare values, and test multiple conditions. They can be classified as:

1. Arithmetic operators
2. Assignment operators
3. Unary operators
4. Comparison operators
5. Shift operators
6. Bit-wise operators
7. Logical operators
8. Conditional operators

1.6.1 Arithmetic Operators

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

Operator	Function
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder after integer division

The % operator is sometimes referred to as the modulus operator.

There is no exponentiation operator in C++. However, there is a library function (pow) to carry out exponentiation. Alternatively you can write your own function to compute exponential value.

The operands acted upon by arithmetic operators must represent numeric values. Thus, the operands can be integer quantities, floating-point quantities or characters (remember that character constants represent integer values, as determined by the computer's character set.) The remainder operator (%) requires that both operands be integers and the second operand be non-zero. Similarly, the division operator (/) requires that the second operand be non-zero.

Division of one integer quantity by another is referred to as integer division. This operation always results in a truncated quotient (i.e., the decimal portion of the quotient will be dropped). On the other hand, if a division operation is carried out with two floating-point numbers, or with one floating-point number and one integer, the result will be a floating-pointing quotient.

Suppose that *a* and *b* are integer variables whose values are 8 and 4, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	**Value**
a + b	12
a – b	4
a * b	32
a / b	2
a % b	0

Notice the truncated quotient resulting from the division operation, since both operands represent integer quantities. Also, notice the integer remainder resulting from the use of the modulus operator in the last expression.

Now suppose that a_1 and a_2 are floating-point variables whose values are 14.5 and 2.0, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	**Value**
$a_1 + a_2$	16.5
$A_1 - a_2$	12.5
$a_1 * a_2$	29.0
a_1 / a_2	7.25

Finally, suppose that x_1 and x_2 are character-type variables that represent the character *M* and *U*, respectively. Some arithmetic expressions that make use of these variables are shown below, together with their resulting values (based upon the ASCII character set).

$$x_1 + x_2 = 162$$

$$x_1 + x_2 + '5' = 215$$

Note that M is encoded as (decimal) 77, U is encoded as 85, and 5 is encoded as 53 in the ASCII character set.

If one or both operands represent negative values, then the addition, subtraction, multiplication and division operations will result in values whose signs are determined by the usual rules of algebra. Integer division will result in truncation toward zero; i.e., the resultant will always be smaller in magnitude than the true quotient.

The interpretation of the remainder operation is unclear when one of the operands is negative. Most versions of C++ assign the sign of the first operand to the remainder. Thus, the condition

$$a = ((a/b) * b) + a \% b)$$

will always be satisfied, regardless of the signs of the values represented by *a* and *b*.

Suppose that *x* and *y* are integer variables whose values are 12 and –2, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	**Value**
x + y	10
x – y	12
x*y	-24
x/y	-6
x%y	0

If *x* had been assigned a value of –12 and *y* had been assigned 2, then the value of *x/y* would still be –6 but the value of a *x%y* would be 0. Similarly, if *x* and *y* had both been assigned negative values (-12 and –2, respectively), then the value of *x/y* would be 3 and the value of *x%y* would be –2.

$$x = ((x/y)*y) + (x\%y)$$

will be satisfied in each of the above cases. Most versions of C++ will determine the sign of the remainder in this manner, though this feature is unspecified in the formal definition of the language.

Here is an illustration of the results that are obtained with floating-point operands having different signs. Let y_1 and y_2 be floating-point variables whose assigned values are 0.70 and 3.50. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	**Value**
Y_1+y_2	2.72
$y_1 - y_2$	–4.28
y/y_2	–0.2728

Operands that differ in type may undergo type conversion before the expression takes on its final value. In general, the final result will be expressed in the highest precision possible, consistent with the data type of the operands. The following rules apply when neither operand is unsigned.

1. If both operands are floating-point types whose precisions differ (e.g., a float and a double), the lower-precision operand will be converted to the precision of the other operand, and the result will be expressed in this higher precision. Thus, an operation between a float and double will result in a double; a float and a long double will result in a long double; and a double and a long double will result in a long double. (Note: In some versions of C++, all operands of type float are automatically converted to double.)

2. If one operand is a floating-point type (e.g., float, double or long double) and the other is a char or an int (including short int or long int), the char or int will be converted to the floating-point type and the result will be expressed as such. Hence, an operation between an int and a double will result in a double.

3. If neither operand is a floating-point type but one is long int, the other will be converted to long int and the result will be long int. Thus, an operation between a long int and an int will result in a long int.

4. If neither operand is a floating-point type or a long int, then both operands will be converted to int (if necessary) and the result will be int. Thus, an operation between a short into and an int will result in an int.

1.6.2 Assignment Operators

Operator	Description	Example	Explanation
=	Assign the value of the right operand to the left	a = b	Assigns the value of b to a
+=	Adds the operands and assigns the result to the left operand	a + =b	Adds the of b to a The expression could also be written as a = a+b
-=	Subtracts the right operand from the left operand and stores the result in the left operand	a -=b	Subtracts b from a Equivalent to a = a-b
*=	Multiplies the left operand by the right operand and stores the result in the left operand	a*=b	Multiplies the values a and b and stores the result in a Equivalent to a = a*b
/=	Divides the left operand by the right operand and stores the result in the left operand	a/=b	Divides a by b and stores the result in a Equivalent to a = a/b
*=	Divides the left operand by the right operand and stores the remainder in the left operand	a%=b	Divides a by b and stores the remainder in a. Equivalent to a = x%y

Any of the operators are used as shown below:

A <operator> =y

can also be represented as

a=a <operator> b

that is, *b* is evaluated before the operation takes place.

You can also assign values to more than one variable at the same time. The assignment will take place from the right to the left. For example,

a = b = 0;

In the example given above, first *b* will be initialized and then a will be initialized.

1.6.3 Unary Operators

Operator	Description	Example	Explanation
++	Increases the value of the operand by one	a++	Equivalent a = a+1
—	Decreases the value of the operand by one	a—	Equivalent to a = a-1

The increment operator, ++, can be used in two ways - as a prefix, in which the operator precedes the variable.

```
++var;
```

In this form, the value of the variable is first incremented and then used in the expression as illustrated below:

```
var1=20;

var2 = ++var1;
```

This code is equivalent to the following set of codes:

```
var1=20;

var1 = var1+1;

var2 = var1;
```

In the end, both variables var1 and var2 store value 21.

The increment operator ++ can also be used as a postfix operator, in which the operator follows the variable.

```
var++;
```

In this case, the value of the variable is used in the expression and then incremented as illustrated below:

```
var1 = 20;

var2 = var1++;
```

The equivalent of this code is:

```
var1 = 20;

var2 = var1;

var1 = var1 + 1;   // Could also have been written as var1 += 1;
```

In this case, variable var1 has the value 21 while var2 remains set to 20.

In a similar fashion, the decrement operator can also be used in both the prefix and postfix forms.

If the operator is to the left of the expression, the value of the expression is modified before the assignment. Conversely, when the operator is to the right of the expression, the C++ statement, var2 = var1++; the original of var1 is assigned to var2. In the statement, var2 = ++var1++; the original value of var1 is assigned to var2. In the statement, var2 = ++var1; the incremented value of var1 is assigned to var2. The operators, '++' and '--', are best used in simple expressions like the ones shown above.

1.6.4 Comparison Operators

Comparison operators evaluate to true or false.

Operator	Description	Example	Explanation
==	Evaluates whether the operands are equal.	A==b	Returns true if the values are equal and false otherwise
!=	Evaluates whether the operands are not equal	a!=y	Returns true if the values are not equal and false otherwise
>	Evaluates whether the left operand is greater than the right operand	a>b	Returns true if a is greater than b and false
<	Evaluates whether the left operand is less than the right operand	a<b	Returns true if a is greater than or equal to b and false otherwise
>=	Evaluates whether the left operand is greater than or equal to the right operand	a>=b	Returns true if a is greater than or equal to b and false otherwise
<=	Evaluates whether the left operand is less than or equal to the right operand	A<=b	Returns true if a is less than or equal to b and false otherwise.

1.6.5 Shift Operators

Data is stored internally in binary format (in the form of bits). A bit can have a value of one or zero. Eight bits form a byte. The following table displays the binary representation of digits 0 to 8.

Decimal	Binary Equivalent
0	00000000
1	00000001
2	00000010
3	00000011
4	00000100
5	00000101
6	00000110
7	00000111
8	00001000

Shift operators work on individual bits in a byte. Using the shift operator involves moving the bit pattern left or right. You can use them only on integer data type and not on the char, bool, float, or double data types.

Operator	Description	Example	Explanation
>>	Shifts bits to the right, filling sign bit at the left	a=10 >> 3	The result of this is 10 divided by 2^3. An explanation follows.
<<	Shifts bits to the left, filling zeros at the right	a=10 << 3	The result of this is 10 multiplied by 2^3. An explanation follows.