6

Key concepts

- **Concept of data types**
- **C++ data types**
- **Fundamental data types**
- **Type modifiers**
- **Variables**
- **Operators**
  - o Arithmetic
  - o Relational
  - o Logical
  - o Input/Output
  - o Assignment
  - o Arithmetic assignment
  - o Increment and decrement
  - o Conditional
  - o sizeof
  - o Precedence of operators
- **Expressions**
  - o Arithmetic
  - o Relational
  - o Logical
- **Type conversion**
- **Statements**
  - o Declaration
  - o Assignment
  - o Input /Output
- **Structure of a C++ program**
  - o Pre-processor directives
  - o Header files
  - o Concept of namespace
  - o The main() function
  - o A sample program
- **Guidelines for coding**

Data Types and Operators

In the previous chapter we familiarised ourselves with the IDE used for the development of C++ programs and also learnt the basic building blocks of C++ language. As we know, data processing is the main activity carried out in computers. All programming languages give importance to data handling. The input data is arranged and stored in computers using some structures. C++ has a predefined template for storing data. The stored data is further processed using operators. C++ also makes provisions for users to define new data types, called user-defined data types.

In this chapter, we will explore the main concepts of the C++ language like data types, operators, expressions and statements in detail.

6.1 Concept of data types

Consider the case of preparing the progress card of a student after an examination. We need data like admission number, roll number, name, address, scores in different subjects, the grades obtained in each subject, etc. Further, we need to display the percentage of marks scored by the student and the attendance in percentage. If we consider a case of scientific data processing, it may require data in the form of numbers representing the velocity of light (3×10^8 m/s), acceleration due to gravity (9.8 m/s), electric charge of an electron (-1.6×10^{-19}) etc.

From these cases, we can infer that data can be of different types like character, integer number, real number, string, etc. In the last chapter we saw that any valid character of C++ enclosed in single quotes represents character data in C++. Numbers without fractions represent integer data. Numbers with fractions represent floating point data and anything enclosed in double quotes represents a string data. Since the data to be dealt with are of many types, a programming language must provide ways and facilities to handle all types of data. C++ provides facilities to handle different types of data by providing data type names. **Data types** are the means to identify the nature of the data and the set of operations that can be performed on the data. Various data types are defined in C++ to differentiate these data characteristics.

In Chapter 4, we used variables to refer data in algorithms. Variables are also used in programs for referencing data. When we write programs in the C++ language, variables are to be declared before their use. Data types are necessary to declare these variables.

6.2 C++ data types

C++ provides a rich set of data types. Based on nature, size and associated operations, they are classified as shown in Figure 6.1. Basically, they are classified into fundamental or built-in data types, derived data types and user-defined data types.

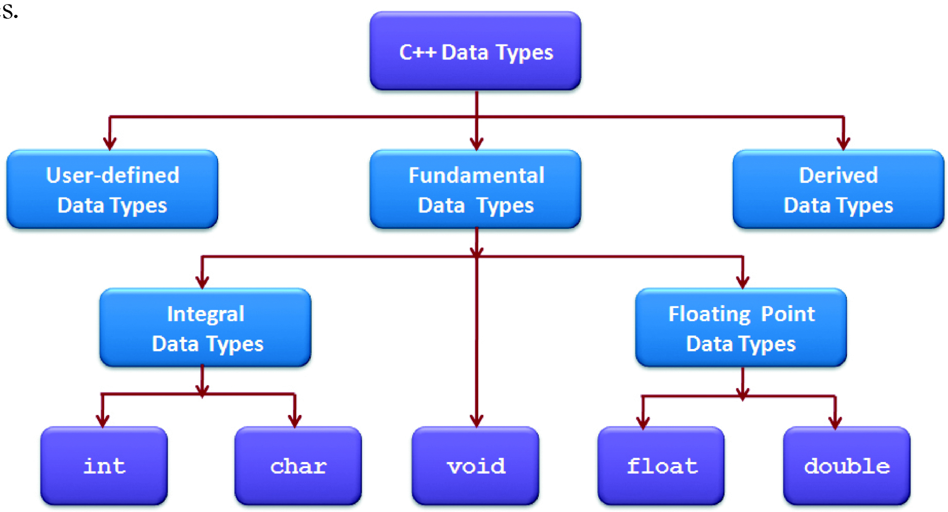

Fig 6.1 : Classification of C++ data types

Fundamental data types

Fundamental data types are defined in C++ compiler. They are also known as built-in data types. They are atomic in nature and cannot be further decomposed of. The five fundamental data types in C++ are `char`, `int`, `float`, `double` and `void`. Among these, `int` and `char` comes under integral data type as they can handle only integers. The numbers with fractions (real numbers) are generally known as floating type and are further divided into `float` and `double` based on precision and range.

User-defined data types

C++ is flexible enough to allow programmers to define their own data types. Structure (`struct`), enumeration (`enum`), union, class, etc. are examples for such data types.

Derived data types

Derived data types are constructed from fundamental data types through some grouping or alteration in the size. Arrays, pointers, functions, etc. are examples of derived data types.

6.3 Fundamental data types

Fundamental data types are basic in nature. They cannot be further broken into small units. Since these are defined in compiler, the size (memory space allocated) depends on the compiler. We use the compiler available in GCC and hence the size as well as the range of data supported by the data type are given accordingly. It may be different if you use other compilers like Turbo C++ IDE. The five fundamental data types are described below:

`int` data type (*for integer numbers*)

Integers are whole numbers without a fractional part. They can be positive, zero or negative. The keyword **`int`** represents integer numbers within a specific range. GCC allows 4 bytes of memory for integers belonging to `int` data type. So the range of values that can be represented by `int` data type is from -2147483648 to +2147483647. The data items 6900100, 0, -112, 17, -32768, +32767, etc. are examples of `int` data type. The numbers 2200000000 and -2147483649 do not belong to `int` data type as they are out of the allowed range.

`char` data type (*for character constants*)

Characters are the symbols covered by the character set of the C++ language. All letters, digits, special symbols, punctuations, etc. come under this category. When these characters are used as data they are considered as `char` type data in C++. We can say that the keyword **`char`** represents character literals of C++. Each `char` type data is allowed one byte of memory. The data items 'A', '+', '\t','0', etc. belong to `char` data type. The `char` data type is internally treated as integers, because computer recognises the character through its ASCII code. Character data is stored in the memory with the corresponding ASCII code. As ASCII codes are integers and need to be stored in one byte (8 bits), the range of `char` data type is from -128 to +127.

`float` data type (*for floating point numbers*)

Numbers with a fractional part are called floating point numbers. Internally, floating-point numbers are stored in a manner similar to scientific notation. The number 47281.97 is expressed as 0.4728197×10^5 in scientific notation. The first part of the number, 0.4728197 is called the mantissa. The power 5 of 10 is called exponent. Computers typically use exponent form (*E notation)* to represent floating-point values. In E notation, the number 47281.97 would be 0.4728197E5. The part of the number before the E is the mantissa, and the part after the E is the exponent. In C++, the keyword **`float`** is used to denote such numbers. GCC allows 4 bytes of memory for numbers belonging to `float` data type. The numbers of this data type has normally a precision of 7 digits.

`double` data type (*for double precision floating point numbers*)

In some cases, floating point numbers require more precision. Such numbers are represented by **`double`** data type. The range of numbers that can be handled by `float` type is extended by this data type, because it consumes double the size of `float` data type. In C++, it is assured that the range and precision of `double` will be at least as big as `float`. GCC reserves 8 bytes for storing a double precision value. The precision of `double` data type is generally 15 digits.

`void` data type (*for null or empty set of values*)

The data type **void** is a keyword and it indicates an empty set of data. Obviously it does not require any memory space. The use of this data type will be discussed in detail in Chapter 10.

The size of fundamental data types decreases in the order `double`, `float`, `int` and `char`.

6.4 Type modifiers

Have you ever seen travel bags that can alter its size/volume to include extra bit of luggage? Usually we don't use that extra space. But the zipper attached with the bag helps us to alter its volume either by increasing it or by decreasing. In C++ too, we need data types that can accommodate data of slightly bigger/smaller size. C++ provides **data type modifiers** which help us to alter the size, range or precision. Modifiers precede the data type name in the variable declaration. It alters the range of values permitted to a data type by altering the memory size and sign of values. Important modifiers are **`signed`**, **`unsigned`**, **`long`** and **`short`**.

The exact sizes of these data types depend on the compiler and computer you are using. It is guaranteed that:

- a `double` is at least as big as a `float`.
- a `long double` is at least as big as a `double`.

Each type and their modifiers are listed in Table 6.1 (based on GCC compiler) with their features.

Name	Description	Size	Range
`char`	Character	1 byte	`signed`: -128 to 127 `unsigned`: 0 to 255
`short int` (`short`)	Short Integer	2 bytes	`signed`: -32768 to 32767 `unsigned`: 0 to 65535
`int`	Integer	4 bytes	`signed`: -2147483648 to 2147483647 `unsigned`: 0 to 4294967295
`long int` (`long`)	Long integer	4 bytes	`signed`: -2147483648 to 2147483647 `unsigned`: 0 to 4294967295
`float`	Floating point number	4 bytes	$-3.4\times10^{+/-38}$ to $+3.4 \times 10^{+/-38}$ with approximately 7 significant digits
`double`	Double precision floating point number	8 bytes	$-1.7 \times 10^{+/-308}$ to $+1.7 \times 10^{+/-308}$ with approximately 15 significant digits
`long double`	Long double precision floating point number	12 bytes	$-3.4 \times 10^{+/-4932}$ to $+3.4 \times10^{+/-4932}$ With approximately 19 significant digits

Table 6.1: Data type and type modifiers

The values listed in Table 6.1 are only sample values to give you a general idea of how the types differ. The values for any of these entries may be different on your system.

6.5 Variables

Memory locations are to be identified to refer data. **Variables** are the names given to memory locations. These are identifiers of C++ by which memory locations are referenced to store or retrieve data. The size and nature of data stored in a variable depends on the data type used to declare it. There are three important aspects for a variable.

i. Variable name

It is a symbolic name (identifier) given to the memory location through which the content of the location is referred to.

ii. Memory address

The RAM of a computer consists of collection of cells each of which can store one byte of data. Every cell (or byte) in RAM will be assigned a unique address to refer it. All the variables are connected to one or more memory locations in RAM. The base address of a variable is the starting address of the allocated memory space. In the normal situation, the address is given implicitly by the compiler. The address is also called the L-value of a variable. In Figure 6.2 the base address of the variable **`Num`** is 1001.

iii. Content

The value stored in the location is called the content of the variable. This is also called the R-value of the variable. Type and size of the content depends on the data type of the variable.

Figure 6.2, shows the memory representation of a variable. Here the variable name is `Num` and it consumes 4 bytes of memory at memory addresses 1001, 1002, 1003 and 1004. The content of this variable is 18. That is the *L-value* of `Num` is 1001 and the *R-value* is 18.

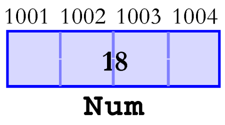

Fig. 6.2 : Memory representation of a Variable

6.6 Operators

Operators are tokens constituted by predefined symbols that trigger computer to carry out operations. The participants of an operation are called **operands**. An operand may be either a constant or a variable.

For example, `a+b` triggers an arithmetic operation in which + (addition) is the operator and `a`, `b` are operands. Operators in C++ are classified based on various criteria. Based on number of operands required for the operation, operators are classified into three. They are unary, binary and ternary.

Unary operators

A unary operator operates on a single operand. Commonly used unary operators are unary+ (positive) and unary– (negative). These are used to represent the sign of a number. If we apply unary+ operator on a signed number, the existing sign will not change. If we apply unary– operator on a signed number, the sign of the existing number will be negated. Examples of the use of unary operators are given in Table 6.2.

Variable x	Unary + +x	Unary- -x
8	8	-8
0	0	0
-9	-9	9

Table 6.2 : Unary operators

Some other examples of unary operators are increment (**++**) and decrement (**−−**) operators.

Binary operators

Binary operators operate on two operands. Arithmetic operators, relational operators, logical operators, etc. are commonly used binary operators.

Ternary operator

Ternary operator operates on three operands. The typical example is the conditional operator (? **:**).

The operations triggered by the operators mentioned above will be discussed in detail in the coming sections and some of them will be dealt with in Chapter 7.

Based on the nature of operation, operators are classified into arithmetic, relational, logical, input/output, assignment, short-hand, increment/decrement, etc.

6.6.1 Arithmetic operators

Arithmetic operators are defined to perform basic arithmetic operations such as addition, subtraction, multiplication and division. The symbols used for this are **+**, **−** ,***** and **/** respectively. C++ also provides a special operator, **%** (modulus operator) for getting remainder during division. All these operators are binary operators. Note that **+** and **−** are used as unary operators too. The operands required for these operations are numeric data. The result of these operations will also be numeric. Table 6.3 shows some examples of binary arithmetic operations.

Variable x	Variable y	Addition x + y	Subtraction x - y	Multiplication x * y	Division x / y
10	5	15	5	50	2
-11	3	-8	-14	-33	-3.66667
11	-3	8	14	-33	-3.66667
-50	-10	-60	-40	500	5

Table 6.3 : Arithmetic operators

Modulus operator (%)

The modulus operator, also called as mod operator, gives the remainder value during arithmetic division. This operator can only be applied over integer operands.

Table 6.4 shows some examples of modulus operation. Note that the sign of the result is the sign of the first operand. Here in the table the first operand is **x**.

Variable x	**Variable y**	**Modulus Operation x % y**	**Variable x**	**Variable y**	**Modulus Operation x % y**
10	5	0	100	100	0
5	10	5	32	11	10
-5	11	-5	11	-5	1
5	-11	5	-11	5	-1
-11	-5	-1	-5	-11	-5

Table 6.4 : Operations using Modulus operator

Check yourself

1. Arrange the fundamental data types in ascending order of size.
2. The name given to a storage location is known as ____________.
3. Name a ternary operator in C++.
4. Predict the output of the following operations if x = -5 and y = 3 initially:

a. –x	f. x + y
b. –y	g. x % y
c. –x + -y	h. x / y
d. –x – y	i. x * –y
e. x % –11	j. –x % -5

6.6.2 Relational operators

Relational operators are used for comparing numeric data**.** These are binary operators. The result of any relational operation will be either **True** or **False.** In C++, True is represented by **1** and False is represented by **0.** There are six relational operators in C++. They are **<** (*less than*), **>** (*greater than*), **<=** (*less than or equal to*), **>=** (*greater than or equal to*), **==** (*equal to*) and **!=** (*not equal to*). Note that equality checking requires two equal symbols (==). Some examples for the use of various relational operators and their results are shown in Table 6.5.

m	n	m<n	m>n	m<=n	m>=n	m!=n	m==n
12	5	0	1	0	1	1	0
-7	2	1	0	1	0	1	0
4	4	0	0	1	1	0	1

Table 6.5 : Operations using Relational operators

6.6.3 Logical operators

Using relational operators, we can compare values. Examples are 3<5, num!=10, etc. These comparison operations are called relational expressions in C++. In some cases, two or more comparisons may need to be combined. In Mathematics we may use expressions like a>b>c. But in C++ it is not possible. We have to separate this into two, as a>b and b>c and these are to be combined using the logical operator **&&**, i.e. (a>b)&&(b>c). The result of such logical combinations will also be either True or False (i.e. 1 or 0). The logical operators are **&&** (logical AND), **||** (logical OR) and **!** (logical NOT).

Logical AND (&&) operator

If two relational expressions El and E2 are combined using logical AND (**&&**) operator, the result will be 1 (True) only if both E1 and E2 have values 1 (True). In all other cases the result will be 0 (False). The results of evaluation of **&&** operation for different possible combination of inputs are shown in Table 6.6.

E1	E2	E1&&E2
0	0	0
0	1	0
1	0	0
1	1	1

Table 6.6 : Logical AND

Examples: 10>5 && 15<25 evaluates to 1 (True)

10>5 && 100<25 evaluates to 0 (False)

Logical OR (||) operator

If two relational expressions El and E2 are combined using logical OR (**||**) operator, the result will be 0 (False) only if both El and E2 are having value 0 (False). In all other cases the result will be 1 (True). The results of evaluation of **||** operation for different possible combination of inputs are shown in Table 6.7.

E1	E2	E1||E2
0	0	0
0	1	1
1	0	1
1	1	1

Table 6.7 : Logical OR

Examples: 10>5 || 100<25 evaluates to 1(True)

10> 15 || 100<90 evaluates to 0 (False)

Logical NOT operator (!)

E1	!E1
0	1
1	0

Table 6.8 : Logical NOT

This operator is used to negate the result of a relational expression. This is a unary operation. The results of evaluation of **!** operator for different possible inputs are shown in Table 6.8.

Example: !(100<2) evaluates to 1 (True)

!(100>2) evaluates to 0 (False)

6.6.4 Input / Output operators

Usually input operation requires user's intervention. In the process of input operation, the data given through the keyboard is stored in a memory location. C++ provides **>>** operator for this operation. This operator is known as ***get from*** or ***extraction*** operator. This symbol is constituted by two greater than symbols.

Similarly in output operation, data is transferred from RAM to an output device. Usually the monitor is the standard output device to get the results directly. The operator **<<** is used for output operation and is called ***put to*** or ***insertion*** operator. It is constituted by two less than symbols.

6.6.5 Assignment operator (=)

When we have to store a value in a memory location, assignment operator (**=**) is used. This is a binary operator and hence two operands are required. The first operand should be a variable where the value of the second operand is to be stored. Some examples are shown in table 6.9.

Item	Description
`a=b`	The value of variable **b** is stored in **a**
`a=3`	The constant **3** is stored in variable **a**

Table 6.9 : Assignment operator

We discussed the usage of the relational operator **==** in Section 6.6.2. See the difference between these two operators. The **=** symbol assigns a value to a variable, whereas **==** symbol compares two values and gives True or False as the result.

6.6.6 Arithmetic assignment operators

A simple arithmetic statement can be expressed in a more condensed form using arithmetic assignment operators. For example, `a=a+10` can be represented as **`a+=10`**. Here **+=** is an arithmetic assignment operator. This method is applicable to all arithmetic operators and they are shown in Table 6.10. The arithmetic assignment operators in C++ are **+=**, **-=**, ***=**, **/=**, **%=**. These are also known as C++ short-hands. These are all binary operators and the first operand should be a variable. The use of these operators makes the two operations (arithmetic and assignment) faster than the usual method.

Arithmetic assignment operation	Equivalent arithmetic operation
`x += 10`	`x = x + 10`
`x -= 10`	`x = x - 10`
`x *= 10`	`x = x * 10`
`x /= 10`	`x = x / 10`
`x %= 10`	`x = x % 10`

Table 6.10 : C++ short hands

6.6.7 Increment (++) and Decrement (−−) operators

Increment and decrement operators are two special operators in C++. These are unary operators and the operand should be a variable. These operators help keeping the source code compact.

Increment operator (++)

This operator is used for incrementing the content of an integer variable by one. This can be written in two ways: **++x** (pre increment) and **x++** (post increment). Both are equivalent to `x=x+1` as well as `x+=1`.

Decrement operator (−−)

As a counterpart of increment operator, there is a decrement operator which decrements the content of an integer variable by one. This operator is also used in two ways: **--x** (pre decrement) and **x--** (post decrement). These are equivalent to `x=x-1` and `x-=1`.

The two usages of these operators are called prefix form and postfix form of increment/decrement operation. Both the forms make the same effect on the operand variable, but the mode of operation will be different when these are used with other operators.

Prefix form of increment/decrement operators

In the prefix form, the operator is placed before the operand and the increment/ decrement operation is carried out first. The incremented/decremented value is used for the other operations. So, this method is often called ***change, then use*** method.

Consider the variables `a`, `b`, `c` and `d` with values `a=10`, `b=5`. If an operation is specified as `c=++a`, the value of `a` will be 11 and that of `c` will also be 11. Here the value of `a` is incremented by 1 at first and then the changed value of `a` is assigned to c. That is why both the variables get the same value. Similarly, after the execution of d=--b the value of d and b will be 4.

Postfix form of increment/decrement operators

When increment/decrement operation is performed in postfix form, the operator is placed after the operand. The current value of the variable is used for the remaining operations and after that the increment/decrement operation is carried out. So, this method is often called ***use, then change*** method.

Consider the same variables used above with the same initial values. After the operation performed with c=a++, the value of a will be 11, but that of c will be 10. Here the value of a is assigned to c at first and then a is incremented by 1. That is, before changing the value of a it is used to assign to c. Similarly, after the execution of d=b-- the value of d will be 5 but that of b will be 4.

6.6.8 Conditional operator (? :)

This is a ternary operator applied over three operands. The first operand will be a logical expression (condition) and the remaining two are values. They can be constants, variables or expressions. The condition will be checked first and if it is True, the second operand will be selected to get the value, otherwise the third operand will be selected. Its syntax is:

```
    Expression1? Expression2: Expression3
```

Let us see the operation in the following:

```
    result = score>50 ? 'p' : 'f'
```

If the value of score is greater than 50 then the value 'p' is assigned to the variable result, else value 'f' is assigned to result. More about this operator will be discussed in Chapter 7.

6.6.9 sizeof operator

The operator sizeof is a unary compile-time operator that returns the amount of memory space in bytes allocated for the operand. The operand can be a constant, a variable or a data type. The syntax followed is given below:

- `sizeof (data_type)`
- `sizeof variable_name`
- `sizeof constant`

It is to be noted that when data type is used as the operand for sizeof operator, it should be given within a pair of parentheses. For the other operands parentheses are not compulsory. Table 6.11 shows different forms of usages of sizeof operator.

Item	Description
`sizeof(int)`	Gives the value 4 (In GCC, size of `int` data type is 4 bytes)
`sizeof 3.2`	Returns 8 (A floating point constant will be taken as `double` type data)
`sizeof p;`	If `p` is `float` type variable, it gives the value 4.

Table 6.11: Various usages of sizeof operator

6.6.10 Precedence of operators

Let us consider the case where different operators are used with the required operands. We should know in which order the operations will be carried out. C++ gives priority to the operators for execution. During evaluation, pair of parentheses is given the first priority. If the expression is not parenthesised, it is evaluated according to the predefined precedence order. The order of precedence for the operators is given in Table 6.12. In an expression, if the operators of the same priority level occur, the precedence of execution will be from left to right in most of the cases.

Priority	Operations
1	`( )` parentheses
2	`++`, `--`, `!` , Unary+ , Unary −, `sizeof`
3	* (multiplication), / (division), **%** (Modulus)
4	+ (addition), − (subtraction)
5	< (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to)
6	== (equal to), != (not equal to)
7	`&&` (logical AND)
8	`||` (logical OR)
9	`? :` (Conditional expression)
10	= (Assignment operator), *=, /=, %=, +=, −= (arithmetic assignment operators)
11	, (Comma)

Table 6.12: Precedence of operators

Consider the variables with values: `a=3, b=5, c=4, d=2, x`

After the operations specified in `x = a + b * c - d`, the value in `x` will be 21. Here * (multiplication) has higher priority than + (addition) and − (subtraction). Therefore the variables `b` and `c` are multiplied, then that result is added to `a`. From that result, `d` is subtracted to get the final result.

It is important to note that the operator priority can be changed in an expression as per the need of the programmer by using parentheses `()`. For example, if `a=5`, `b=4`, `c=3`, `d=2` then the result of `a+b-c*d` will be 3. Suppose the programmer wants to perform subtraction first and then the addition and multiplication, you need to use proper parentheses as `(a+(b-c))*d`. Now the output will be 12. For changing operator priority, brackets `[]` and braces `{}` cannot be used.

The operator precedence may be different for different types of compilers. Turbo C++ gives higher precedence to prefix increment / decrement than its postfix form.

For example, if a is initially 5, the values of b and a after b=a++ + ++a are 12 and 7 respectively. This is equivalent to the set of statements a=a+1 (prefix expansion), b=a+a, and a=a+1 (postfix expansion).

6.7 Expressions

An expression is composed of operators and operands. The operands may be either constants or variables. All expressions can be evaluated to get a result. This result is known as the value returned by the expression. On the basis of the operators used, expressions are mainly classified into arithmetic expressions, relational expressions and logical expressions.

6.7.1 Arithmetic expressions

An expression in which only arithmetic operators are used is called arithmetic expression. The operands are numeric data and they may be variables or constants. The value returned by these expressions is also numeric. Arithmetic expressions are further classified into integer expressions, floating point (real) expressions and constant expressions.

Integer expressions

If an arithmetic expression contains only integer operands, it is called integer expression and it produces an integer result after performing all the operations given in the expression. For example, if x and y are integer variables, some integer expressions and their results are shown in Table 6.13. Note that all the above expressions produce integer values as the results.

x	y	x + y	x / y	-x + x * y	5 + x / y	x % y
5	2	7	2	5	7	1
6	3	9	2	12	7	0

Table 6.13: Integer expressions and their results

Floating point expressions (Real expressions)

An arithmetic expression that is composed of only floating point data is called floating point or real expression and it returns a floating point result after performing all the operations given in the expression. Table 6.14 shows some real expressions and their results, assuming that `x` and `y` are floating point variables.

x	y	x + y	x / y	-x + x * y	5 + x / y	x * x / y
5.0	2.0	7.0	2.5	5.0	7.5	12.5
6.0	3.0	9.0	2.0	12.0	7.0	12.0

Table 6.14: Floating point expressions and their results

It can be seen that all the above expressions produce floating point values as the results.

In an arithmetic expression, if all the operands are constant values, then it is called **constant expression**. The expression 20+5/2.0 is an example. The constants like `15`, `3.14`, `'A'` are also known as constant expressions.

6.7.2 Relational expressions

When relational operators are used in an expression, it is called relational expression and it produces Boolean type results like True (1) or False (0). In these expressions, the operands are numeric data. Let us see some examples of relational expressions in Table 6.15.

x	y	x > y	x == y	x+y !=y	x-2 == y+1	x*y == 6*y
5.0	2.0	1 (True)	0 (False)	1 (True)	1 (True)	0 (False)
6	13	0 (False)	0 (False)	1 (True)	0 (False)	1 (True)

Table 6.15: Relational expressions and their results

We know that arithmetic operators have higher priority than relational operators. So when arithmetic expressions are used on either side of a relational operator, arithmetic operations will be carried out first and then the results are compared. The table contains some expressions in which both arithmetic and relational operators are involved. Though they contain mixed type of operators, they are called relational expressions since the final result will be either True or False.

6.7.3 Logical expressions

Logical expressions combine two or more relational expressions with logical operators and produce either True or False as the result. A logical expression may contain constants, variables, logical operators and relational operators. Let us see some examples in Table 6.16.

x	y	x>=y && x==20	x==5||y==0	x==y && y+2==0	!(x==y)
5.0	2.0	0 (False)	1 (True)	0 (False)	1 (True)
20	13	1 (True)	0 (False)	0 (False)	1 (True)

Table 6.16: Logical expressions and their results

As seen in Table 6.16, though some expressions consist of arithmetic and relational operators in addition to logical operators, the expressions are considered as logical expressions. This is because the operation carried out at last will be the logical operation and the result will be either True or False.

Check yourself

1. Predict the output of the following operations if x=5 and y=3.

   a. x>=10 && y>=4 c. x>=1 || y>=4
   b. x>=1 && y>=3 d. x>=1 || y>=3

2. Predict the output if p=5, q=3, r=2

   a. ++p-q*r/2 c. p-q-r*2+p
   b. p*q--+r d. p+=5*q+r*r/2

6.8 Type conversion

As discussed earlier arithmetic expressions are of two types, integer expressions and real expressions. In both cases, the operands involved in the arithmetic operation are of the same data type. But there are situations where different types of numeric data may be involved. For example in C++, the integer expression 5 /2 gives 2 and the real expression 5.0/2.0 gives 2.5. But what will the result of 5/2.0 or 5.0/2 be? Conversion techniques are applied in such situations. The data type of one operand will be converted to another. It is called **type conversion** and can be done in two ways: implicitly and explicitly.

6.8.1 Implicit type conversion (Type promotion)

Implicit type conversion is performed by C++ compiler internally. In expressions where different types of data are involved, C++ converts the lower sized operands to the data type of highest sized operand. Since the conversion is always from lower type to higher, it is also known as **type promotion**. Data types in the decreasing order of size are as follows: `long double`, `double`, `float`, `unsigned long`, `long int and unsigned int / short int`. The type of the result will also be the type of the highest sized operand.

For example, the expression 5 / 2 * 3 + 2.5 gives the result 8.5. The evaluation steps are as follows:

Step 1:	5 / 2 → 2	(Integer division)
Step 2:	2 * 3 → 6	(Integer multiplication)
Step 3:	6 + 2.5 → 8.5	(Floating point addition, 6 is converted into 6.0)

6.8.2 Explicit type conversion (Type casting)

Unlike implicit type conversion, sometimes the programmer may decide the data type of the result of evaluation. This is done by the programmer by specifying the data type within parentheses to the left of the operand. Since the programmer explicitly casts a data to the desired type, it is known as explicit type conversion or **type casting**. Usually, type casting is applied on the variables in the expressions. More examples will be discussed in Section 6.9.2.

6.9 Statements

Can you recollect the learning hierarchy of a natural language? Alphabet, words, phrases, sentences, paragraphs and so on. In the learning process of C++ language we have covered character set, tokens and expressions. Now we have come to the stage where we start communication with the computer sensibly and meaningfully with the help of statements. **Statements** are the smallest executable unit of a programming language. C++ uses the symbol semicolon (;) as the delimiter of a statement. Different types of statements used in C++ are declaration statements, assignment statements, input statements, output statements, control statements etc. Each statement has its own purpose in a C++ program. All these statements except declaration statements are executable statements as they possess some operations to be done by the computer. Executable statements are the instructions to the computer. The execution of control statements will be discussed in Chapter 7. Let us discuss the other statements.

6.9.1 Declaration statements

Every user-defined word should be defined in the program before it is used. We have seen that a variable is a user-defined word and it is an identifier of a memory location. It must be declared in the program before its use. When we declare a variable, we tell the compiler about the type of data that will be stored in it. The syntax of variable declaration is:

```
data_type <variable1>[, <variable2>, <variable3>,...];
```

The `data_type` in the syntax should be any valid data type of C++. The syntax shows that when there are more than one variables in the declaration, they are separated by comma. The declaration statement ends with a semicolon. Typically, variables are declared either just before they are used or at the beginning of the program. In the syntax, everything given inside the symbols [and] are optional. The following statements are examples for variable declaration:

```
int  rollnumber;
double  wgpa,  avg_score;
```

The first statement declares the variable `rollnumber` as **`int`** type so that it will be allocated four bytes of memory (*as per GCC*) and it can hold an integer number within the range from -2147483648 to +2147483647. The second statement defines the identifiers `wgpa` and `avg_score` as variables to hold data of **`double`** type. Each of them will be allocated 8 bytes of memory. The memory is allocated to the variables during the compilation of the program.

Variable initialisation

We saw, in Section 6.5, that a variable is associated with two values: L-value (its address) and R-value (its content). When a variable is declared, a memory location with an address will be allocated for it. What will its content be? It is not blank or 0 or space! If the variable is declared with `int` data type, the content or R-value will be any integer within the allowed range. But this number cannot be predicted or will not always be the same. So we call it *garbage value*. When we store a value into the variable, the existing content will be replaced by the new one. The value can be stored in the variable either at the time of compilation or execution. Supplying value to a variable at the time of its declaration is called **variable initialisation**. This value will be stored in the respective memory location during compile-time. The assignment operator (=) is used for this. It can be done in two ways as given below:

```
data_type  variable  =  value;
```

OR

```
data_type  variable(value)
```

The statements: `int xyz =120;` and `int xyz(120);` are examples of variable initialisation statements. Both of these statements declare an integer variable `xyz` and store the value 120 in it as shown in Figure 6.3.

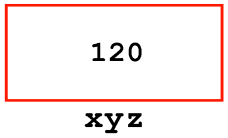

Fig. 6.3: Variable initialisation

More examples are:

```
float  val=0.12,  b=5.234;
char  k='A';
```

A variable can also be initialised during the execution of the program and is known as dynamic initialisation. This is done by assigning an expression to a variable as shown in the following statements:

```
float product =   x * y;
float  interest  = p*n*r/100.0;
```

In the first statement, the variable `product` is initialised with the product of the values stored in `x` and `y` at runtime. In the second case, the expression `p*n*r/100.0` is evaluated and the value returned by it will be stored in the variable `interest`.

Note that during dynamic initialisation, the variables included in the expression at the right of assignment operator should have valid data. Otherwise it will produce unexpected results.

`const` – The access modifier

It is a good practice to use symbolic constants rather than using numeric constants directly. For example, we can use symbolic names like Pi instead of using 22.0/7.0 or 3.14. The keyword **`const`** is used to create such symbolic constants whose value can never be changed during execution. Consider the following statement:

```
float pi=3.14;
```

The floating point variable **`pi`** is initialised with the value 3.14. The content of **`pi`** can be changed during the execution of the program. But if we modify the declaration as:

```
const float pi=3.14;
```

the value of **`pi`** remains constant (unaltered) throughout the execution of the program. The read/write accessibility of the variable is modified as read only. Thus, the **`const`** acts as an access modifier.

During software development, larger programs are developed using collaborative effort. Several people may work together on different portions of the same program. They may share the same variable. In these situations, there may be occasions where one may modify the content of the variable which will adversely affect other person's coding. In these situations we have to keep the content of variables unaffected by the activity of others. This can be done by using 'const'.

6.9.2 Assignment statements

When the assignment operator (=) is used to assign a value to a variable, it forms an assignment statement. It can take any of the following syntax:

```
variable  =  constant;
variable1  =  variable2;
variable  =  expression;
variable  =  function();
```

In the third case, the result of the expression is stored in the variable. Similarly, in the fourth case, the value returned by the function is stored. The concept of functions will be discussed in Chapter 10.

Some examples of assignment statements are given below:

```
A = 15;                        b = 5.8;
c = a + b;                     c = a * b;
d = (a + b)*(c + d);           r = sqrt(25);
```

In the last example, `sqrt()` is a function that assigns the square root of 25 to the variable `r`.

The left hand side (LHS) of an assignment statement must be a variable. During execution, the expression at the right hand side (RHS) is evaluated first. The result is then assigned (stored) to the variable at LHS.

Assignment statement can be chained for doing multiple assignments at a time. For instance, the statement `x=y=z=13;` assigns the value 13 in three variables in the order of `z`, `y` and `x`. The variables should be declared before this assignment. If we assign a value to a variable, the previous value in it, if any, will be replaced by the new value.

Type compatibility

During the execution of an assignment statement, if the data type of the RHS expression is different from that of the LHS variable, there are two possibilities.

- The size of the data type of the variable at LHS is higher than that of the variable or expression at RHS. In this case data type of the value at RHS is promoted (type promotion) to that of the variable at LHS. Consider the following code snippet:

  ```
  int a=5, b=2;
  float p, q;
  p = b;
  q = a / p;
  ```

  Here the data type of `b` is promoted to `float` and 2.0 is stored in `p`. When the expression `a/p` is evaluated, the result will be 2.5 due to the type promotion of `a`. So, `q` will be assigned with 2.5.

- The second possibility is that the size of the data type of LHS variable is smaller than the size of RHS value. In this case, the higher order bits of the result will be truncated to fit in the variable location of LHS. The following code illustrates this.

  ```
  float a=2.6;
  int p, q;
  p = a;
  q = a * 4;
  ```

  Here the value of `p` will be 2 and that of `q` will be 10. The expression `a*4` is evaluated to 10.4, but `q` being `int` type it will hold only 10.

Programmer can apply the explicit conversion technique to get the desired results when there are mismatches in the data types of operands. Consider the following code segment.

```
int p=5, q=2;
float x, y;
x = p/q;
y = (x+p)/q;
```

After executing the above code, the value of `x` will be 2.0 and that of `y` will be 3.5. The expression `p/q` being an integer expression gives 2 as the result and is stored in `x` as floating point value. In the last statement, the pair of parentheses gives priority to `x+p` and the result will be 7.0 due to the type promotion of `p`. Then the result 7.0 will be the first operand for the division operation and hence the result will be 3.5 since `q` is converted into `float`. If we have to get the floating point result from `p/q` to store in `x`, the statement should be modified as `x=(float)p/q;` or `x=p/(float)q;` by applying type casting.

6.9.3 Input statements

Input statement is a means that allows the user to store data in the memory during the execution of the program. We saw that the ***get from*** or ***extraction*** operator (**>>**) specifies the input operation. The operands required for this operator are the input device and a location in RAM where data is to be stored. Keyboard being a standard console device, the stream (sequence) of data is extracted from the keyboard and stored in memory locations identified by variables. Since C++ is an object oriented language, keyboard is considered as the standard input stream device and is identified as an object by the name **cin** (pronounced as 'see in'). The simplest form of an input statement is:

```
streamobject >> variable;
```

Since we use keyboard as the input device, the `streamobject` in the syntax will be substituted by `cin`. The operand after the **>>** operator should strictly be a variable. For example, the following statement reads data from the keyboard and stores in the variable **num**.

```
cin >> num;
```

Figure 6.4 shows how data is extracted from keyboard and stored in the variable.

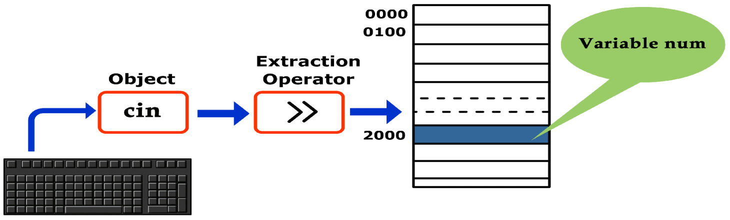

Fig 6.4 : Input procedure in C++

6.9.2 Output statements

Output statements make the results available to users through any output device. The ***put to*** or ***insertion*** operator (**<<**) is used to specify this operation. The operands in this case are the output device and the data for the output. The syntax of an output statement is:

```
streamobject << data;
```

The `streamobject` may be any output device and the data may be a constant, a variable or an expression. We use monitor as the commonly used output device and C++ identifies it as an object by the name **`cout`** (pronounced as 'see out'). The following are some examples of output statement with monitor as the output device:

```
cout << num;
cout << "hello friends";
cout << num+12;
```

The first statement displays the content of the variable **num**. The second statement displays the string constant `"hello friends"` and the last statement shows the value returned by the expression `num+12` (*assuming that* `num` *contains numeric value*). Figure 6.5 shows how data is inserted into the output stream object (monitor) from the memory location **num**.

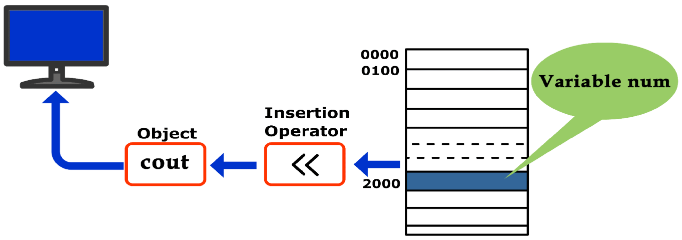

Fig. 6.5: Output procedure in C++

The tokens cin and cout are not keywords. They are predefined words that are not part of the core C++ language, and you are allowed to redefine them. They are defined in libraries required by the C++ language standard. Needless to say, using a predefined identifier for anything other than its standard meaning can be confusing and dangerous and such practice should be avoided. The safest and easiest practice is to treat all predefined identifiers as if they were keywords.

Cascading of I/O operators

Suppose you want to input three values to different variables, say x, y, and z. You may use the following statements:

```
cin>>x;
cin>>y;
cin>>z;
```

But these three statements can be combined to form a single statement as given below:

```
cin>>x>>y>>z;
```

The multiple use of input or output operators in a single statement is called **cascading of I/O operators**. In the use of cascading of input operators, the values input are assigned to the variables from left to right. In the example `cin>>x>>y>>z;` the first value is assigned to `x`, the second to `y` and the third to `z`. While entering values to the variables `x`, `y` and `z` during execution the values should be separated by space bar, tab, or carriage return.

Similarly, if you want to display the contents of different variables (say `x`, `y`, `z`), use the following statement:

```
cout<<x<<y<<z;
```

If variables, constants and expressions appear together for output operations, the above technique can be applied as in the following example:

```
cout<<"The number is "<<z;
```

While cascading output operators, the values for the output will be retrieved from right to left. Consider the code fragment given below:

```
int x=5;
cout<<x<<'\t'<<++x;
```

The output of this code will be: 6 6

It will not be: 5 6

It is to be noted that both **<<** and **>>** operators cannot be used in a single statement.

In the statement `x=y=z=5;` the **=** operator is cascaded. Here also the cascading is from right to left.

6.10 Structure of a C++ program

We are now in a position to solve simple problems by using the statements we discussed so far. But a set of statements alone does not constitute a program. A C++ program has a typical structure. It is a collection of one or more functions. A function means the set of instructions to perform a particular task referred to by a name. Since there can be many functions in a C++ program, they are usually identified by unique names. The most essential function needed for every C++ program is the **`main()`** function.

The structure of a simple C+ + program is given below:

```
#include <header file>
using namespace identifier;
int main()
{
     statements;
        :
        :
        :
     return 0;
}
```

The first line is called preprocessor directive and the second line is the namespace statement. The third line is the function header which is followed by a set of statements enclosed by a pair of braces. Let us discuss each of these parts of the program.

6.10.1 Preprocessor directives

A C++ program starts with pre-processor directives. Preprocessors are the compiler directive statements which give instruction to the compiler to process the information provided before actual compilation starts. Preprocessor directives are lines included in the code that are not program statements. These lines always start with a **#** (hash) symbol. The pre-processor directive `#include` is used to link the header files available in the C++ library by which the facilities required in the program can be obtained. No semicolon (**;**) is needed at the end of such lines. Separate `#include` statements should be used for different header files. There are some other pre-processor directives such as `#define`, `#undef`, etc.

6.10.2 Header files

Header files contain the information about functions, objects and predefined derived data types and they are available along with compiler. There are a number of such files to support C++ programs and they are kept in the standard library. Whichever program requires the support of any of these resources, the concerned header file is to be included. For example, if we want to use the predefined objects `cin` and `cout`, we have to use the following statement at the beginning of the program.

```
#include <iostream>
```

The header file `iostream` contains the information about the objects `cin` and `cout`. Eventhough header files have the extension `.h`, it should not be specified for GCC. But the extension is essential for some other compilers like Turbo C++ IDE.

6.10.3 Concept of namespace

A program cannot have the same name for more than one identifier (variables or functions) in the same scope. For example, in our home two or more persons (or even living beings) will not have the same name. If there are, it will surely make conflicts in the identity within the home. So, within the scope of our home, a name should be unique. But our neighbouring home may have a person (or any living being) with the same name as that of one of us. It will not make any confusion of identity within the respective scopes. But an outsider cannot access a particular person by simply using the name; but the house name is also to be mentioned.

The concept of namespace is similar to a house name. Different identifiers are associated to a particular namespace. It is actually a group name in which each item is unique in its name. User is allowed to create own namespaces for variables and functions. We can use an identifier to give name to a namespace. The keyword `using` technically tells the compiler about a namespace where it should search for the elements used in the program. In C++, `std` is an abbreviation of '`standard`' and it is the standard namespace in which `cout`, `cin` and a lot of other objects are defined. So, when we want to use them in a program, we need to follow the format `std::cout` and `std::cin`. This kind of explicit referencing can be avoided with the statement `using namespace std;` in the program. In such a case, the compiler searches this namespace for the elements `cin`, `cout`, `endl`, etc. So whenever the computer comes across `cin`, `cout`, `endl` or anything of that matter in the program, it will read it as `std::cout`, `std::cin` or `std::endl`.

The statement `using namespace std;` doesn't really add a function, it is the `#include <iostream>` that "loads" `cin`, `cout`, `endl` and all the like.

6.10.4 The `main()` function

Every C++ program consists of a function named `main()`. The execution starts at `main()` and ends within `main()`. If we use any other function in the program, it is called (or invoked) from `main()`. Usually a data type precedes the `main()` and in GCC, it should be `int`.

The function header `main()` is followed by its body, which is a set of one or more statements within a pair of braces `{ }`. This structure is known as the definition of the `main()` function. Each statement is delimited by a semicolon (`;`). The statements may be executable and non-executable. The executable statements represent instructions to be carried out by the computer. The non-executable statements are intended for compiler or programmer. They are informative statements. The last statement in the body of `main()` is return 0;. Even though we do not use this statement, it will not make any error. Its relavance will be discussed in Chapter 10.

C++ is a free form language in the sense that it is not necessary to write each statement in new lines. Also a single statement can take more than one line.

6.10.5 A sample program

Let us look at a complete program and familiarise ourselves with its features, in detail. This program on execution will display a text on the screen.

```
#include <iostream>
using namespace std;
int main()
{
      cout<<"Hello, Welcome to C++";
      return 0;
}
```

The program has seven lines as detailed below:

Line 1: The preprocessor directive `#include` is used to link the header file `iostream` with the program.

Line 2: The `using namespace` statement makes available the identifier `cout` in the program.

Line 3: The header of the essential function for a C++ program, i.e., `int main()`.

Line 4: An opening brace `{` that marks the beginning of the instruction set (program).

Line 5: An output statement, which will be executed when we run the program, to display the text `"Hello, Welcome to C++"` on the monitor. The header file `iostream` is included in this program to use `cout` in this statement.

Line 6: The `return` statement stops the execution of the `main()` function. This statement is optional as far as `main()` is concerned.

Line 7: A closing brace `}` that marks the end of the program.

6.11 Guidelines for coding

A source code looks good when the coding is legible, logic is communicative and errors if any are easily detectable. These features can be experienced if certain styles are followed while writing programs. Some guidelines are discussed in this section to write stylistic programs.

Use suitable naming convention for identifiers

Suppose we have to calculate the salary for an employee after deductions. We may code it as: `A = B – C;`

where `A` is the net salary, `B` the total salary and `C` total deduction. The variable names `A`, `B` and `C` do not reflect the quantities they denote. If the same instruction is expressed as follows, it would be better:

```
    Net_salary = Gross_salary – Deduction;
```

The variable names used in this case help us to remember the quantity they possess. They readily reflect their purpose. These kinds of identifiers are called ***mnemonic names***. The following points are to be remembered in the choice of names:

- Choose good mnemonic names for all variables, functions and procedures.
  e.g. `avg_hgt`, `Roll_No`, `emp_code`, `SumOfDigits`, etc.
- Use standardized prefixes and suffixes for related variables.
  e.g. `num1`, `num2`, `num3` for three numbers
- Assign names to constants in the beginning of the program.
  e.g. `float PI = 3.14;`

Use clear and simple expressions

Some people have a tendency to reduce the execution time by sacrificing simplicity. This should be avoided. Consider the following example. To find out the remainder after division of `x` by `n`, we can code as: `y = x-(x/n)*n;`

The same thing is achieved by a simpler and more elegant piece of code as shown below:

```
          y = x % n;
```

So it is better to use simpler codes in programming to make the program more simple and clear.

Use comments wherever needed

Comments play a very important role as they provide internal documentation of a program. They are lines in code that are added to describe the program. They are ignored by the compiler. There are two ways to write comments in C++:

Single line comment: The characters **//** (two slashes) is used to write single line comments. The text appearing after **//** in a line is treated as a comment by the C++ compiler.

Multiline comments: Anything written within **/*** and ***/** is treated as comment so that the comment can take any number of lines.

But care should be taken that no relevant code of the program is included accidently inside the comment. The following points are to be noted while commenting:

- Always insert prologues, the comments in the beginning of a program that summarises the purpose of the program.
- Comment each variable and constant declaration.
- Use comments to explain complex program steps.
- It is better to include comments while writing the program itself.
- Write short and clear comments.

Relevance of indentation

In computer programming, an indent style is a convention governing the indentation of blocks of code to convey the program's structure, for good visibility and better clarity. An indentation makes the statements clear and readable. It shows the levels of statements in the program.

The usage of these guidelines can be observed in the programs given in the next section.

Program gallery

Let us now write programs to solve some problems following the coding guidelines. The call-outs given are not part of the program. Program 6.1 displays a message.

Program 6.1: To display a message

```
      /* This program displays the message
         "Smoking is injurious to health"
         on the monitor */
#include <iostreamh>   // To use the cout object
using namespace std;  // To access cout
int main() //program begins here
{     //The following output statement displays a message
      cout << "Smoking is injurious to health";
      return 0;
}     //end of the program
```

Multiline comment

Single line comment

Indentation

On executing Program 6.1, the output will be as follows:

```
Smoking is injurious to health
```

More illustrations on the usage of indentation can be seen in the examples given in Chapter 7.

Program 6.2 accepts two integer numbers from the user, finds its sum and displays the result.

Program 6.2: To find the sum of two integer numbers

```
#include <iostream>
using namespace std;
int main()
{  //Program begins
/* Two variables are declared to read user inputs and the
variable sum is declared to store the result
*/
   int num1, num2, sum;
   cout<<"Enter two numbers: "; //Prompt for input
   cin>>num1>>num2;    //Cascading to get two numbers
   sum=num1+num2;   //Assignment statement to find the sum
   cout<<"Sum of the entered numbers = "<<sum;
/* The result is displayed with proper message.
   Cascading of output operator is utilized     */
   return 0;
}
```

A sample output of Program 6.2 is given below:

```
Enter two numbers: 5   7
Sum of the entered numbers = 12
```

User inputs separated by spaces

Let us consider another problem. A student is awarded with three scores obtained in three Continuous Evaluation (CE) activities. The maximum score of an activity is 20. Find the average score of the student.

Program 6.3: To find the average of three CE scores

```
#include <iostream>
using namespace std;
int main()
{
    int score_1, score_2, score_3;
    float avg;
    //Average of 3 numbers can be a floating point value
    cout << "Enter the three CE scores: ";
    cin >> score_1 >> score_2 >> score_3;
    avg = (score_1 + score_2 + score_3) / 3.0;
/* The result of addition will be an integer value. If 3
is written instead of 3.0, integer division will be
performed and will not get the correct result  */
    cout << "Average CE score is: " << avg;
    return 0;
}
```

Program 6.3 gives the following output for the CE scores 17, 19 and 20.

```
Enter the three CE scores: 17   19   20
Average CE score is: 18.666666
```

The assignment statement to find the average value uses an expression to the right of assignment operator (=). This expression has two + operators and one / operator. The precedence of / over + is changed by using parentheses for addition. The operands for the addition operators are all `int` type data and hence the result will be an integer. When this integer result is divided by 3, the output will again be an integer. If it was so, the output of Program 6.3 would have been 18, which is not accurate. Hence floating point constant 3.0 is used as the second operand for / operator. It makes the integer numerator `float` by type promotion.

Suppose the radius of a circle 'r' is given and you are requested to compute its area and the perimeter. As you know, area of a circle is calculated using the formula πr^2 and perimeter by $2\pi r$, where $\pi = 3.14$. Program 6.4 solves this problem.

Program 6.4: To find the area and perimeter of a circle for a given radius

```
#include <iostream>
using namespace std;
int main()
{
    const float PI = 22.0/7; //Use of const access modifier
    float radius, area, perimeter;
    cout<<"Enter the radius of the circle: ";
    cin>>radius;
    area = PI * radius * radius;
    perimeter = 2 * PI * radius;
    cout<<"Area of the circle = "<<area<<"\n";
    cout<<"Perimeter of the circle = "<<perimeter;
    return 0;
}
```

Escape sequence '\n' prints a new line after displaying the value of Area

A sample output of Program 6.4 is as follows:

```
    Enter the radius of the circle: 2.5
    Area of the circle = 19.642857
    Perimeter of the circle = 15.714285
```

The last two output statements of Program 6.4 displays both the results in separate lines. The escape sequence character `'\n'` brings the cursor to the new line before the last output statement gets executed.

Let us develop another program to find simple interest. As you know, principal amount, rate of interest and period are to be given as input to get the result.

Program 6.5: To find the simple interest

```
#include <iostream>
using namespace std;
int main()
{
    float p_Amount, n_Year, i_Rate, int_Amount;
    cout<<"Enter the principal amount in Rupees: ";
    cin>>p_Amount;
    cout<<"Enter the number of years for the deposit: ";
    cin>>n_Year;
    cout<<"Enter the rate of interest in percentage: ";
    cin>>i_Rate;
    int_Amount = p_Amount * n_Year * i_Rate /100;
    cout <<"Simple interest for the principal amount "
         <<p_Amount<<" Rupees for a period of "<<n_Year
         <<" years at the rate of interest "<<i_rate
         <<" is "<<int_Amount<<" Rupees";
    return 0;
}
```

A sample output of Program 6.5 is given below:

```
Enter the principal amount in Rupees: 100
Enter the number of years for the deposit: 2
Enter the rate of interest in percentage: 10
Simple interest for the principal amount 100 Rupees for a
period of 2 years at the rate of interest 10 is 20 Rupees
```

The last statement in Program 6.5 is the output statement and it spans over four lines. Note that there is no semi colon at the end of each line and so it is considered a single statement. On execution of the program the result may be displayed in multiple lines depending on the size and resolution of the monitor of your computer.

Program 6.6 solves a temperature conversion problem. The temperature in degree celsius will be given as input and the output will be its equivalent in fahrenheit.

Program 6.6: To convert temperature from Celsius to Fahrenheit

```
#include <iostream>
using namespace std;
int main()
{
    float celsius, fahrenheit;
    cout<<"Enter the Temperature in Celsius: ";
    cin>>celsius;
    fahrenheit=1.8*celsius+32;
    cout<< celsius<<" Degree Celsius = "
        << fahrenheit<<" Degree Fahrenheit";
    return 0;
}
```

Program 6.6 gives a sample output as follows:

```
Enter the Temperature in Celsius: 37
37 Degree Celsius = 98.599998 Degree Fahrenheit
```

We know that each character literal in C++ has a unique value called its ASCII code. These values are integers. Let us write a program to find the ASCII code of a given character.

Program 6.7: To find the ASCII value of a character

```
#include <iostream>
using namespace std;
int main()
{
     char ch;
     int asc;
     cout << "Enter the character: ";
     cin >> ch;
     asc = ch;
     cout << "ASCII value of "<<ch<<"  = " << asc;
     return 0;
}
```

A sampler output of Program 6.7 is given below:

```
Enter the character: A
ASCII value of A = 65
```

Let us sum up

Data types are means to identify the type of data and associated operations handling it. Each data type has a specific size and a range of data. Data types are used to declare variables. Type modifiers help handling a higher range of data and are used with data types to declare variables. Different types of operators are available in C++ for various operations. When operators are combined with operands (data), expressions are formed. There are mainly three types of expressions - arithmetic, relational and logical. Type conversion methods are used to get desired results from arithmetic expressions. Statements are the smallest executable unit of a program. Variable declaration statements define the variables in the program and they will be allocated memory space. The executable statements like assignment statements, input statements, output statements, etc. help giving instructions to the computer. Some special operators like arithmetic assignment, increment, decrement, etc. make the expressions and statements compact and the execution faster. C++ program has a typical structure and it must be followed while writing programs. Stylistic guidelines shall be followed to make the program attractive and communicative among humans.

Learning outcomes

After the completion of this chapter the learner will be able to

- identify the various data types in C++.
- list and choose appropriate data type modifiers.
- choose appropriate variables.
- experiment with various operators.
- apply the various I/O operators.
- write various expressions and statements.
- identify the structure of a simple C++ program.
- identify the need for stylistic guidelines while writing a program.
- write simple programs using C++.

Lab activity

1. Write a program that asks the user to enter the weight in grams, and then display the equivalent in Kilograms.
2. Write a program to generate the following table

2013	100%
2012	99.9%
2011	95.5%
2010	90.81%
2009	85%

   Use a single cout statement for output. (Hint: Make use of \n and \t)

4. Write a short program that asks for your height in Meter and Centimeter and converts it to Feet and inches. (1 foot = 12 inches, 1 inch = 2.54 cm).
5. Write a program to compute simple interest and compound interest.
6. Write a program to : (i) print ASCII code for a given digit, (ii) print ASCII code for backspace. (Hint : Store escape sequence for backspace in an integer variable).
7. Write a program to accept a time in seconds and convert into hrs: mins: secs format. For example, if 3700 seconds is the input, the output should be 1hr: 1 min: 40 secs.

Sample questions

Very short answer type

1. What are data types? List all predefined data types in C++.
2. What is a constant?
3. What is dynamic initialisation of variables?
4. What is type casting?
5. Write the purpose of declaration statement?
6. Name the header file to be included to use cin and cout in programs?
7. What is the input operator ">>" and output operator "<<" called ?
8. What will be the result of a = 5/3 if a is (i) `float` and (ii) `int` ?

9. What will be the value of P= P++ + ++i where **i** is 22 and P= 3 initially?

10. Find the value given by the following expression if j =5 initially.
    (i) (5*++j)%6 (ii) (5*j++)%6

11. What will be the order of evaluation for following expressions?
    (i) i+5>=j-6 (ii) s+10<p-2+2*q

12. What will be the result of the following if ans is 6 initially?
    (i) cout <<ans = 8 ; (ii) cout << ans == 8

Short answer type

1. What is a variable? List the two values associated with it.
2. In how many ways can a variable be declared in C++?
3. Explain the impact of type modifiers of C++ in variable declaration.
5. What is the role of the keyword 'const'?
6. Explain how prefix form of increment operation differs from postfix form.
8. Write down the operation performed by sizeof operator.
9. Explain the two methods of type conversions.
10. What would happen if `main()` is not present in a program?
11. Identify the errors in the following code segments:

(a)
```
int main()
{ cout << "Enter two numbers"
cin >> num >> auto
float area = Length * breadth ; }
```

(b)
```
#include <iostream>
using namespace std
void Main()
{ int a, b
cin <<a <<b
max=(a > b) a:b
cout>max
}
```

12. Find out the errors, if any, in the following + + statements:

(i) `cout<< "a=" a;`	(v) `cin >> "\n" >> y ;`
(ii) `m=5,n=12;015`	(vi) `cout >> \n "abc"`

(iii) `cout << "x" ; <<x;` (vii) `a = b + c`

(iv) `cin >> y` (viii) `break = x`

13. What is the role of relational operators? Distinguish between == and =.
14. Comments are useful to enhance readability and understandability of a program. Justify this statement with examples.

Long answer type

1. Explain the operators of C++ in detail.
2. Explain the different types of expressions in C++ and the methods of type conversions in detail.
3. Write the working of arithmetic assignment operator? Explain all arithmetic assignment operators with the help of examples.