6

Introduction to Programming

Key concepts

- **Structure of C++ Program**
  - o Pre-processor directives
  - o Header files
  - o A sample C++ program
- **Stylistic guidelines for variables and comments**
- **Variable initialization**
- **Use of const access modifier**
- **Type modifiers**
- **More operators**
  - o Arithmetic assignment
  - o Increment and decrement
  - o Precedence of operators
- **Type conversions**
  - o Implicit type conversion
  - o Explicit type conversion

We have familiarized ourselves with the IDE used for C++ program. In the last chapter we discussed how operations can be expressed in C++. We also learnt different types of statements in C++ to give instructions to the computer. We are now in a position to solve simple problems by using these statements. But a set of statements alone does not constitute a program. A C++ program has a typical structure. In this chapter we discuss the structure of C++ programs and we write programs accordingly. More operators are introduced for making the programs compact and the execution faster.

6.1 Structure of a C++ program

C++ program is a collection of one or more functions. A function means a 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. Let us look at the structure of a simple C++ program.

```
#include <headerfile>
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.

6.1.1 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 preprocessor 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 preprocessor directives such as `#define`, `#undef`, etc.

6.1.2 Header files

Header files contain the information about functions, objects and predefined derived data types and 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.

6.1.3 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 a name to the 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.

6.1.4 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 Class XII.

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.

6.1.5 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.

6.2 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 program. Some guidelines are discussed below 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. This kind of identifiers are called ***mnemonic names***. The points given below 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 standardised 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 usages of these guidelines can be observed in the programs given in a later section of this chapter.

6.3 Variable initialisation

In the last chapter we discussed the significance of variables in programs. 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 and it is the L-value of the variable. 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 the 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 6.1.

More examples are:

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

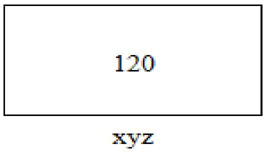


Fig 6.1: Variable initialisation

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` at runtime.

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.

6.4 `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 name 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 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`**.

6.5 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/or 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 $+3.4\times10^{+/-38}$ with approximately 7 significant digits
`double`	Double precision floating point number	8 bytes	$-1.7\times10^{+/-308}$ to $+1.7\times10^{+/-308}$ with approximately 15 significant digits
`long double`	Long double precision floating point number	10 bytes	$-3.4\times10^{+/-4932}$ to $+3.4\times10^{+/-4932}$ With approximately 19 significant digits

Table 6.1 : Data type and type modifiers

The values listed above 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.

6.6 More Operators

We discussed the operators in C++ for the commonly used operations. There are some special operators in C++ which make programs more condensed. They combine two operations together. We will discuss a few of them which combine assignment and arithmetic operations into a single operator. Arithmetic assignment, increment and decrement operators are some examples.

6.6.1 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 6.2. 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 6.2: C++ short hands

6.6.2 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 operation. 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 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 performed in postfix form, the operator is placed after the operand. The current value of the variable is used for the remaining operation 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.

6.6.3 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 execution, a pair of parentheses is given the first priority. If the expression is not parenthesised, it is evaluated according to some precedence order. The order is given in Table 6.3. In an expression, if the operations of same priority level occur, the precedence of execution will be from left to right in most of the cases.

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

After the operations specified in the following:

```
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, then 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.

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 6.3: Precedence of operators

6.7 Type conversion

We discussed in the last chapter that arithmetic operations are represented by arithmetic expressions, which may be either integer expression or 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.

6.7.1 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`, `unsigned int`, `int` and `short int`. The type of the result will also be the type of the highest sized operand.

For example, consider the expression `5 / 2 * 3 + 2.5` which 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)

6.7.2 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 also known as explicit type conversion or **type casting**. Usually, type casting is applied on the variables in the expressions. The data type compatibility is to be seriously considered in assignment statements where expressions are involved.

Type compatibility in assignment statement

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`. So 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.

Program Gallery

Let us write programs to solve some problems following the guidelines. Program 6.1 displays a message.

***Program 6.1:* To display a message**

```
/* This is a program which displays the message
   smoking is injurious to health
   on the monitor */
#include <iostream>   // To use the cout object
using namespace std; // To access the cout object
int main() //program begins here
{    //The following statement displays the 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 programs given in Chapter 7. Program 6.2 accepts two numbers from user, finds their sum and displays the result. It uses all types of statements that we discussed in Chapter 5.

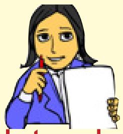
Program 6.2: To find the sum of two numbers

```
#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: ";//Used as prompt for input
     cin>>num1>>num2; //Cascading facility to get two numbers
     sum=num1+num2;  //Assignment statement to store the sum
     /* The result is displayed with proper message.
        Cascading of output operator is utilized      */
     cout<<"Sum of the entered numbers = "<<sum;
     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 do

Run Program 6.2 with some other inputs like 5.2 and 2.7 and check whether you get correct answer. If not, discuss with your friends to identify the problem and find solution.

Let's consider another problem. A student is awarded the scores in three Continuous Evaluation 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 may be a floating point
  cout << "Enter the three 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 take place and result will not be correct  */
  cout << "Average CE score is: " << avg;
  return 0;
}
```

Program 6.3 gives the following output for the scores given below:

```
Enter the three 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 is given and you have to calculate its area and 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

```
#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 a 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 shown below:

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

The last two output statements of Program 6.4 display 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's develop yet another program to find simple interest. As you know, principal amount, rate of interest and period are to be 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 of 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 of 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 an output statement and it spans over five lines. It is a single statement. Note that there is no semicolon at the end of each line. On execution the result may be displayed in two lines depending on the size and resolution of the monitor of your computer.

Program 6.6 solves a conversion problem. The temperature in degree Celsius will be the 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 codes are integers. Let us write a program to find the ASCII code of a given character.

***Program 6.7:* To find the ASCII code of a character**

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

A sample output of Program 6.7 is given below:

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

Let us sum up

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. While declaring variables, they can be initialised with user supplied values. The value thus assigned cannot be changed if we use const access modifier in the variable initialization statement. Type modifiers help handling a higher range of data and are used with data types to declare variables. There are some special operators in C++ for performing arithmetic and assignment operations together. Type conversion methods are available in C++ and are used to get desired results from arithmetic expressions.

Learning outcomes

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

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

Lab activity

1. Write a program that asks the user to enter the weight in grams, and then display the equivalent in Kilogram.
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)

3. 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).
4. Write a program to find area of a triangle.
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 is dynamic initialisation of variables?
2. What is type promotion?
3. What do you mean by type casting? What is type cast operator?

Short answer type

1. What type of variable declaration is used in the following program code ?

```
{ int area, length = 10, width = 12, perimeter;
  area = length * width;
  perimeter = 2*(length + width);
}
```

2. Modify the above program code to have dynamic initialisation for variables area and perimeter.
3. Explain type modifiers in C++.
4. Why are so many data types provided in C++?
5. What is the role of the keyword '`const`'?
6. Explain pre increment and post increment operations.
7. Explain the methods of type conversions.
8. If the file iostream.h is not included in a program, what happens?
9. What would happen if `main()` is not present in a program?

10. Identify the errors in the following code segments

    i.
```
int main ( ) { cout << "Enter two numbers"
cin >> num >> auto
float area = Length * breadth ; }
```
    ii.
```
#include <iostream>
using namespace;
int Main()
{ int a, b
  cin <<a >>b
  max = a % b
  cout > max
}
```

11. Comments are a useful and an easy way to enhance readability and understandability of a program. Justify this statement with examples.

Long answer type

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