INTRODUCTION TO C++

Part-2

educlash.com
Just another Way To Learn

When declaring a regular local variable, its value is by default undetermined. But you may want a variable to store a concrete value at the same moment that it is declared. In order to do that, you can initialize the variable. There are two ways to do this in C++:

The first one, known as c-like, is done by appending an equal sign followed by the value to which the variable will be initialized:

```
type identifier = initial_value ;
```

For example, if we want to declare an int variable called a initialized with a value of 0 at the moment in which it is declared, we could write:

```
int a = 0;
```

The other way to initialize variables, known as constructor initialization, is done by enclosing the initial value between parentheses
(()):

```
type identifier (initial_value) ;
```

For example:

```
int a (0);
```

Both ways of initializing variables are valid and equivalent in C++.

```
// initialization of variables

#include <iostream>
using namespace std;

int main ()
{
        int a=5; int              // initial value = 5
        b(2); int                 // initial value = 2
        result;                            //  initial  value
undetermined

        a = a + 3;
        result = a - b;
        cout << result;

        return 0;
}
```

3.9 REFERENCE VARIABLES

C++ allows you to create a second name for the variable that you can use to read or modify the original data stored in that variable. While this may not sound appealing at first, what this means is that when you declare a reference and assign it a variable, it will allow you to treat the reference exactly as though it were the original variable for the purpose of accessing and modifying the value of the original variable - even if the second name (the reference) is located within a different scope.

Basic Syntax:

Declaring a variable as a reference rather than a normal variable simply entails appending an ampersand to the type name.

type &identifier = identifier/constant;

When a reference is created, you must tell it which variable it will become an alias for. After you create the reference, whenever you use the variable, you can just treat it as though it were a regular variable. But when you create it, you must initialize it with another variable, whose address it will keep around behind the scenes to allow you to use it to modify that variable.

In a way, this is similar to having a pointer that always points to the same thing. One key difference is that references do not require dereferencing in the same way that pointers do; you just treat them as normal variables. A second difference is that when you create a reference to a variable, you need not do anything special to get the memory address. The compiler figures this out for you.

```
int x;
int &foo = x;

// foo is now a reference to x so this sets x to 56
foo = 56;
std::cout << x <<std::endl;
```

The most common use of references is for function parameters. Reference parameters facilitate the pass-by-reference style of arguments, as opposed to the pass-by-value style. To observe the differences, consider the three swap functions in the program below.

```
void Swap1 (int x, int y)               //    pass-by-value
(objects)
{
        int temp = x;
        x = y;
        y = temp;
}

void Swap2 (int *x, int *y)             //    pass-by-value
(pointers)
{
        int temp = *x;
        *x = *y;
        *y = temp;
}

void Swap3 (int &x, int &y)             //         pass-by-
reference
{
        int temp = x;
        x = y;
        y = temp;
}
```

Annotation:

- Although *Swap1* swaps x and y, this has no effect on the arguments passed to the function, because Swap1 receives a copy of the arguments. What happens to the copy does not affect the original.
- *Swap2* overcomes the problem of *Swap1* by using pointer parameters instead. By dereferencing the pointers, *Swap2* gets to the original values and swaps them.
- *Swap3* overcomes the problem of *Swap1* by using reference parameters instead. The parameters become aliases for the arguments passed to the function and therefore swap them as intended. *Swap3* has the added advantage that its call syntax is the same as *Swap1* and involves no addressing or dereferencing.

3.10 CONSTANTS

Constants are expressions with a fixed value. C++ has two kinds of constants: literal, and symbolic. C++ has two kinds of constants: literal, and symbolic.

3.10.1 Literal constants:

Literal constants are literal numbers used to express particular values within the source code of a program. They are constants because you can't change their values. We have already used these previously to give concrete values to variables or to express messages we wanted our programs to print out, for example, when we wrote:

```
a = 5;
```

the 5 in this piece of code was a literal constant.

Literal constants can be divided in Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.

Integer Numerals: They are numerical constants that identify *integer decimal values*. Notice that to express a numerical constant we do not have to write quotes (") nor any special character. In addition to decimal numbers (those that all of us are used to use every day) C++ allows the use as literal constants of octal numbers (base 8) and hexadecimal numbers (base 16). If we want to express an octal number we have to precede it with a 0 (zero character). And in order to express a hexadecimal number we have to precede it with the characters 0x (zero, x). For example, the following literal constants are all equivalent to each other:

```
75      // decimal

0113    // octal

0x4b    // hexadecimal
```

Floating-Point Numerals: They express numbers with *decimals* and/or *exponents*. They can include either a decimal point, an e character (that expresses "by ten at the Xth height", where X is an integer value that follows the e character), or both a decimal point and an e character. For example, the following are floating-point literals:

```
3.14159    // 3.14159
6.02e23    // 6.02 x 10^23
1.6e.19    // 1.6 x 10^.19
3.0        // 3.0
```

Character and String Literals: There also exist non-numerical constants, like:

```
'z'
'p'
"Hello world"
"How do you do?"
```

The first two expressions represent single character constants, and the following two represent string literals composed of several characters. Notice that to represent a single character we enclose it between *single quotes (')* and to express a string (which generally consists of more than one character) we enclose it between *double quotes (")*.

Boolean Literals: There are only two valid Boolean values: *true* and *false*. These can be expressed in C++ as values of type *bool* by using the Boolean literals true and false.

3.10.2 Symbolic constants:

Symbolic constants can be declared in two different ways:
using the **#define** preprocessor directive, and
through use of the **const** keyword.

You can define your own names for constants that you use very often without having to resort to memory consuming variables, simply by using the #define preprocessor directive. Its format is:

```
#define identifier value
```

For example:

```
#define PI 3.14159
#define NEWLINE '\n'
```

This defines two new constants: PI and NEWLINE. Once they are defined, you can use them in the rest of the code as if they were any other regular constant, for example:

```
// defined constants: calculate circumference

#include <iostream>
using namespace std;

#define PI 3.14159
#define NEWLINE '\n'

int main ()
{
        double r=5.0;                // radius
        double circumf;

        circumf = 2 * PI * r;
        cout << circumf;
        cout << NEWLINE;

        return 0;
}
```

OUTPUT: 31.4159

In fact the only thing that the compiler preprocessor does when it encounters *#define* directives is to literally replace any occurrence of their identifier (in the previous example, these were PI and NEWLINE) by the code to which they have been defined (3.14159 and '\n' respectively).

There are two major problems with symbolic constants declared using #define. First, because they are resolved by the preprocessor, which replaces the symbolic name with the defined value, #defined symbolic constants do not show up in the debugger. Second, #defined values always have global scope. This means value #defined in one piece of code may have a naming conflict with a value #defined with the same name in another piece of code.

A better way to do symbolic constants is through use of the **const** keyword. Const variables must be assigned a value when declared, and then that value cannot be changed.

```
const double pi = 3.14159
const char newline = ‘\n’
```

Here, *pi* and *newline* are two typed constants. Although a constant variable might seem like an oxymoron, they can be very useful in helping to document your code. Const variables act exactly like normal variables in every case except that they cannot be assigned to.

3.11 ASSIGNMENT STATEMENTS

The main statement in C++ for carrying out computation and assigning values to variables is the **assignment statement**. For example the following assignment statement:

```
average = (a + b)/2;
```

assigns half the sum of `a` and `b` to the variable `average`. The general form of an assignment statement is:

identifier = *expression* ;

The *expression* is evaluated and then the value is assigned to the *identifier*. It is important to note that the value assigned to *identifier* must be of the same type as *identifier*.

The *expression* can be a single variable, a single constant or involve variables and constants combined by the arithmetic operators. Rounded brackets `()` may also be used in matched pairs in expressions to indicate the order of evaluation.

For example, let us have a look at the following code - I have included the evolution of the content stored in the variables as comments:

```
// assignment operator

#include <iostream>
using namespace std;

int main ()
{
  int a, b;          // a:?,   b:?
  a = 10;            // a:10, b:?
  b = 4;             // a:10, b:4
  a = b;             // a:4,   b:4
  b = 7;             // a:4,   b:7

  cout << "a:";
  cout << a;
  cout << " b:";
  cout << b;

  return 0;
}
```

OUTPUT:

```
a:4 b:7
```

This code will give us as result that the value contained in *a* is 4 and the one contained in *b* is 7. Notice how *a* was not affected by the final modification of *b*, even though we declared *a = b* earlier (that is because of the right-to-left rule).

The following expression is also valid in C++:

a = b = c = 5;

It assigns 5 to the all the three variables: a, b and c.

3.12 SUMMARY

- A variable is a symbolic name for a memory location in which data can be stored and subsequently recalled.
- A variable has two important attributes - a type and a value
- A variable declaration has the form:
    - type identifier-list;
- Identifiers are names given to variables which distinguish them from all other variables.
- Variables names (identifiers) can only include letters of the alphabet, digits and the underscore character. They must commence with a letter.
- Reserved words are valid identifiers that have special significance to C++.
- C++ is case-sensitive. That is lower-case letters are treated as distinct from upper-case letters.
- Variables can either have global scope or local scope
- All variables and constants that are used in a C++ program must be declared before use. Declaration associates a type and an identifier with a variable.
- C++ allows us to declare a variable anywhere in the program, as also its initialization at run time, using the expressions at the place of declaration.
- Reference variable is the second name for the variable that can be used to read or modify the original data stored in that variable
- Variables can be initialized in two ways:
    - type identifier = initial_value ;
    - type identifier (initial_value) ;
- Literal constants are literal numbers used to express particular values within the source code of a program.
- Literal constants can be divided in Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.
- Symbolic constants can be declared in two different ways:

- using the #define preprocessor directive, and
- through use of the const keyword.

➢ The assignment statement in C++ is used for carrying out computation and assigning values to variables.

➢ In an assignment statement, the expression on the right hand side of the assignment is evaluated and, if necessary, converted to the type of the variable on the left hand side before the assignment takes place.

13. UNIT-END EXERCISES

1. Questions:

These questions are intended as a self-test for readers.

1) Define variable. State the important attributes of a variable.
2) Define Identifier. State the rules for valid identifiers.
3) Define global and local scope.
4) Explain reference variables with suitable examples.
5) State the different types of constants
6) Which of the following represent valid variable definitions?
    a. int n = -100;
    b. unsigned int i = -100;
    c. signed int = 2.9;
    d. long m = 2, p = 4;
    e. int 2k;
    f. double x = 2 * m;
    g. float y = y * 2;
    h. unsigned double z = 0.0;
    i. double d = 0.67F;
    j. float f = 0.52L;
    k. signed char = -1786;
    l. char c = '$' + 2;
    m. sign char h = '\111';
    n. char *name = "Peter Pan";
    o. unsigned char *num = "276811";
7) Which of the following represent valid identifiers?
    a. identifier
    b. seven_11
    c. _unique_
    d. gross-income
    e. gross$income

f. 2by2
g. default
h. average_weight_of_a_large_pizza
i. variable
j. object.oriented

2. Programming Projects:

Writing programs that solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied.

1) Given the following definition of a Swap function void Swap (int x, int y)

```
{
        int temp = x;
        x = y;
        y = temp;
}
```

what will be the value of x and y after the following call:

```
        x = 10;
        y = 20;
        Swap(x, y);
```

2) Assuming that n is 20, what will the following code fragment output when executed?

```
if (n >= 0)
        if (n < 10)
                cout << "n is small\n";
else
        cout << "n is negative\n";
```

3.14 FURTHER READING

Bjarne Stroustrup, “The C++ Programming Language: Second Edition”.

Herbert Schildt, “The C++ Complete Reference”

E Balagurusamy, “Object Oriented Programming C++”, Third Edition.

4

Data Types and Expressions

Unit Structure:

1. OBJECTIVES

After completing this chapter you will be able to:

- ➢ Understand and Identify different data types used in C++
- ➢ Understand various operators
- ➢ Understand expressions and utilise them in programs

2. INTRODUCTION

The memory in our computers is organized in bytes. A byte is the minimum amount of memory that we can manage in C++. A byte can store a relatively small amount of data: one single character or a small integer (generally an integer between 0 and 255). In addition, the computer can manipulate more complex data types that come from grouping several bytes, such as long numbers or non-integer numbers.

When programming, we store the variables in our computer's memory, but the computer has to know what kind of data we want to store in them, since it is not going to occupy the same amount of memory to store a simple number than to store a single letter or a large number, and they are not going to be interpreted the same way.

This chapter simply provides the most basic elements from which C++ programs are constructed. You must know these elements, plus the terminology and simple syntax that goes with them, in order to complete a real project in C++ and especially to read code written by others.

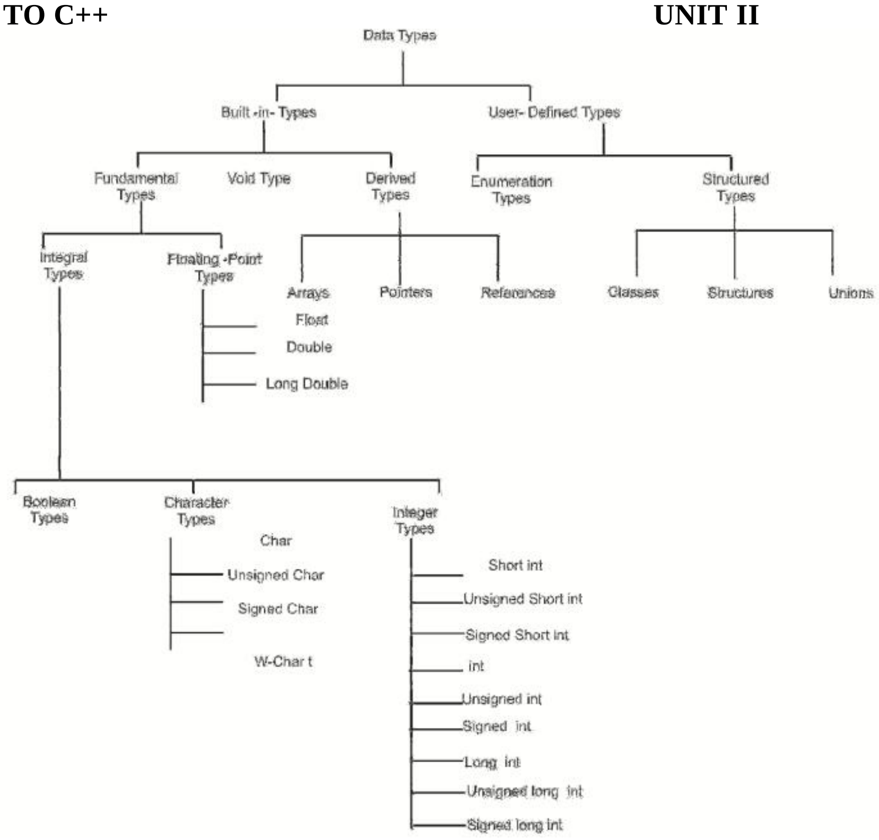
3. DATA TYPES

Every name (identifier) in a C++ program has a type associated with it. This type determines what operations can be applied to the name (that is, to the entity referred to by the name) and how such operations are interpreted. For example, the declarations

```
float x;         //x is a floating-point variable
int y = 7;       //y is an integer variable with the initial value 7
float f(int);    //f is a function taking an argument of type int and returning a floating-point number
```

would make the example meaningful. Because y is declared to be an int, it can be assigned to, used in arithmetic expressions, etc. On the other hand, f is declared to be a function that takes an int as its argument, so it can be called given a suitable argument.

Data types in Standard C++ are classified as shown in the diagram below.



The *Boolean*, *character*, and *integer* types are collectively called *integral* types. The *integral* and *floating-point* types are collectively called *arithmetic* types. *Enumeration* and *structured* types are called *user-defined* types because they must be defined by users rather than being available for use without previous declaration, the way fundamental types are. In contrast, other types are called *built-in* types.

The *integral* and *floating-point* types are provided in a variety of sizes to give the programmer a choice of the amount of storage consumed, the precision, and the range available for computations. The assumption is that a computer provides bytes for holding characters and words, for holding and computing - integer values, some entity most suitable for floating-point computation, and addresses for referring to those entities. The C++ fundamental types together with *pointers* and *arrays* present these machine-level notions to the programmer in a reasonably implementation independent manner.

For most applications, one could simply use *bool* for logical values, *char* for characters, *int* for integer values, and *double* for floating-point values. The remaining fundamental types are variations for optimizations and special needs that are best ignored until such needs arise. They must be known, however, to read old C and C++ code.

4.3.1 Booleans:

A Boolean, *bool*, can have one of the two values *true* or *false*. A Boolean is used to express the results of logical operations. For example:

```
void f(int a, int b)
{
      bool b1 = a == b; /   / = is assignment, == is
equality
```

```
            /  / ...
    }
```

If a and b have the same value, b1 becomes true; otherwise, b1 becomes false.
A common use of bool is as the type of the result of a function that tests some condition (a
predicate). For example:

```
    bool is_open (File*);
    bool greater(int a, int b)  { return a>b;  }
```

By definition, true has the value 1 when converted to an integer and false has the value 0. Conversely, integers can be implicitly converted to bool values: nonzero integers convert to true and 0 converts to false. For example:

```
    bool b = 7;        // bool(7) is true, so b
    becomes true
    int i = true;     // int(true) is 1, so i becomes 1
```

In arithmetic and logical expressions, bools are converted to ints; integer arithmetic and logical operations are performed on the converted values. If the result is converted back to bool, a 0 is converted to false and a nonzero value is converted to true.

```
    void g()
    {
        bool a = true;
        bool b = true;
        bool x = a + b;         // a+b is 2, so x
        becomes true
        bool y = a – b;        // a-b is 0, so y
        becomes false
    }
```

4.3.2 Character Types:

A variable of type char can hold a character of the implementation’s character set. For example:

```
    char ch = ‘a’;
```

Almost universally, a char has 8 bits so that it can hold one of 256 different values. Typically, the character set is a variant of ISO-646, for example ASCII, thus providing the characters appearing on your keyboard. Each character constant has an integer value. For example, the value of ‘b’ is 98 in the ASCII character set. Here is a small program that will tell you the integer value of any character you care to input:

```
#include <iostream>
int main()
{
    char c;
    std::cin >> c;
    std::cout << “The value of ‘” << c << “’ is” <<
int(c) << ‘\n’;
}
```

The notation ***int***(c) gives the integer value for a character *c*. The possibility of converting a char to an integer raises the question: is a char signed or unsigned? The 256 values represented by an 8-bit byte can be interpreted as the values 0 to 255 or as the values -127 to 127. Unfortunately, which choice is made for a plain char is implementation-defined. C++ provides two types for which the answer is definite; signed char, which can hold at least the values -127 to 127, and unsigned char, which can hold at least the values 0 to 255.

Fortunately, the difference matters only for values outside the 0 to 127 range, and the most common characters are within that range.

A type *wchar_t* is provided to hold characters of a larger character set such as Unicode. It is a distinct type. The size of *wchar_t* is implementation-defined and large enough to hold the largest character set supported by the implementation's locale. The strange name is a leftover from C. In C, *wchar_t* is a *typedef* rather than a built-in type. The suffix '*_t*' was added to distinguish standard *typedefs*.

Note that the character types are integral types so that arithmetic and logical operations apply.

3. Integer Types:

Like char, each integer type comes in three forms: *"plain" int, signed int, and unsigned int*. In addition, integers come in three sizes: *short int, "plain" int, and long int*. A *long int* can be referred to as plain long. Similarly, short is a synonym for *short int*, unsigned for *unsigned int*, and signed for *signed int*. The unsigned integer types are ideal for uses that treat storage as a bit array. Using an unsigned instead of an int to gain one more bit to represent positive integers is almost never a good idea. Attempts to ensure that some values are positive by declaring variables unsigned will typically be defeated by the implicit conversion rules. Unlike plain chars, plain ints are always signed. The signed int types are simply more explicit synonyms for their plain int counterparts.

4. Floating-Point Types:

The floating-point types represent floating-point numbers. Like integers, floating-point types come in three sizes: *float* (single-precision), *double* (double-precision), and *long double* (extended-precision).

The exact meaning of single-, double-, and extended-precision is implementation-defined. Choosing the right precision for a problem where the choice matters requires significant understanding of floating-point computation. If you don't have that understanding, get advice, take the time to learn, or use *double* and hope for the best.

5. Sizes:

Some of the aspects of C++'s fundamental types, such as the size of an ***int***, are implementation - defined. I point out these dependencies and often recommend avoiding them or taking steps to minimize their impact. Why should you bother? People who program on a variety of systems or use a variety of compilers care a lot because if they don't, they are forced to waste time finding and fixing obscure bugs. If your program is a success, it is likely to be ported, so someone will have to find and fix problems related to implementation-dependent features. In addition, programs often need to be compiled with other compilers for the same system, and even a future release of your favourite compiler may do some things differently from the current one. It is far easier to know and limit the impact of implementation dependencies when a program is written than to try to untangle the mess afterwards.

It is relatively easy to limit the impact of implementation-dependent language features. Limiting the impact of systemdependent library facilities is far harder. Using standard library facilities wherever feasible is one approach.

Sizes of C++ objects are expressed in terms of multiples of the size of a char, so by definition the size of a char is 1. The size of an object or type can be obtained using the ***sizeof*** operator. Following is a complete C++ program showing the number of bytes that the fundamental types occupy in the memory.

```
#include <iostream>
using namespace std;
int main()
{
      cout << “size of char          =”          <<
sizeof(char)      << endl;
      cout << “size of short         =”          <<
sizeof(short)     << endl;
      cout << “size of int           =”          <<
sizeof(int)       << endl;
      cout << “size of long          =”          <<
sizeof(long)      << endl;
      cout << “size of float         =”          <<
sizeof(float)     << endl;
      cout << “size of double        =”          <<
sizeof(char)      << endl;
}
```

The ***char*** type is supposed to be chosen by the implementation to be the most suitable type for holding and manipulating characters on a given computer; it is typically an 8-bit byte. Similarly, the ***int*** type is supposed to be chosen to be the most suitable for holding and manipulating integers on a given computer; it is typically a 4-byte (32-bit) word. It is unwise to assume more. For example, there are machines with 32 bit chars.

Next you have a summary of the basic fundamental data types in C++, as well as the range of values that can be represented with each one:

Name	Size	Range
Char	1 byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	2 bytes	signed: -32768 to 32767 unsigned: 0 to 65535
Int	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
Bool	1 byte	true or false
Float	4 bytes	+/- 3.4e +/- 38 (~7 digits)
Double	8 bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8 bytes	+/- 1.7e +/- 308 (~15 digits
wchar_t	2 or 4 bytes	1 wide character

4.3.6 Void:

The type *void* is syntactically a fundamental type. It can, however, be used only as part of a more complicated type; there are no objects of type void. It is used either to specify that a function does not return a value or as the base type for pointers to objects of unknown type. For example:

```
void x;     // error: there are no void objects
void f()   ; // function f does not return a value
void *pv;    // pointer to object of unknown type
```

When declaring a function, you must specify the type of the value returned. Logically, you would expect to be able to indicate that a function didn't return a value by omitting the return type. However, that would make the grammar less regular and clash with C usage. Consequently, *void* is used as a ''pseudo return type'' to indicate that a function doesn't return a value.

7. Enumerations:

An ***enumeration*** is a type that can hold a set of values specified by the user. Once defined, an *enumeration* is used very much like an integer type. Named integer constants can be defined as members of an enumeration. For example,

```
enum { ASM, AUTO, BREAK };
```

defines three integer constants, called enumerators, and assigns values to them. By default, enumerator values are assigned increasing from 0, so ASM = 0, AUTO = 1, and BREAK = 2. An enumeration can be named. For example:

```
enum keyword { ASM, AUTO, BREAK };
```

Each *enumeration* is a distinct type. The type of an enumerator is its enumeration. For example,
AUTO is of type keyword.

Declaring a variable *keyword* instead of *plain int* can give both the user and the compiler a hint as to the intended use. For example:

```
void f(keyword key)
{
      switch (key)
      {
            case ASM:
                  // do something
                  break;
            case BREAK:
                  // do something
                  break;
      }
}
```

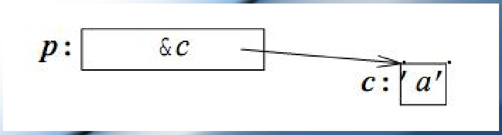
A compiler can issue a warning because only two out of three keyword values are handled. By default, enumerations are converted to integers for arithmetic operations. An *enumeration* is a *user-defined type*, so users can define their own operations, such as ++ and << for an enumeration.

8. Pointers:

For a type T, T* is the type "***pointer*** to T". That is, a variable of type T* can hold the address of an object of type T. For example:

```
char c =  'a';
char *p = &c;           // p holds the address of c
```

or graphically:



Unfortunately, pointers to arrays and pointers to functions need a more complicated notation:

```
int *pi;            // pointer to int
char **ppc;         // pointer to pointer to char
int *ap[15];        // array of 15 pointers to ints
int (*fp)(char*); // pointer to function taking a
char* argument;
                  returns an int
int *f(char*);    // function taking a char*
argument; returns a
                  pointer to int
```

The fundamental operation on a ***pointer*** is dereferencing, that is, referring to the object pointed to by the pointer. This operation is also called indirection. The dereferencing operator is (prefix) unary *. For example:

```
char c = 'a';
char *p = &c;           // p holds the address of c
char c2 = *p;           // c2 == 'a'
```

The variable pointed to by p is c, and the value stored in c is 'a', so the value of *p assigned to c2 is 'a'.

It is possible to perform some arithmetic operations on *pointers* to array elements. *Pointers* to functions can be extremely useful. The implementation of pointers is intended to map directly to the addressing mechanisms of the machine on which the program runs.

4.3.9 Arrays:

For a type T, T[size] is the type "***array*** of size elements of type T". The elements are indexed
from 0 to size - 1. For example:

```
float v[3];       // an array of three floats:
v[0], v[1], v[2]
char *a[32];            // an array of 32 pointers
to char: a[0] .. a[31]
```

The number of elements of the array, the array bound, must be a constant expression. If you need variable bounds, use a ***vector***. For example:

```
void f(int i)
{
      int v1[i];  // error: array size not a
      constant expression
      vector <int> v2 (i);    // ok
}
```

Multidimensional arrays are represented as arrays of arrays. For example:

```
      int d2[10][20];         // d2 is an array of 10
arrays of 20 integers
```

4.3.10 References:

A ***reference*** is an alternative name for an object. The main use of references is for specifying arguments and return values for functions in general and for overloaded operators in particular. The notation X& means reference to X. To ensure that a reference is a name for something (that is, bound to an object), we must initialize the reference. For example:

```
int i = 1;
int &r1 = i;            // ok: r1 initialized
int &r2;          // error: initializer missing
extern int &r3;         // ok: r3 initialized
elsewhere
```

References to variables and references to constants are distinguished because the introduction of a temporary in the case of the variable is highly error-prone; an assignment to the variable would become an assignment to the – soon to disappear – temporary. No such problem exists for references to constants, and references to constants are often important as function arguments. A reference can be used to specify a function argument so that the function can change the value of an object passed to it.

4.3.11 Structures:

An array is an aggregate of elements of the same type. A ***struct*** is an aggregate of elements of (nearly) arbitrary types. For example:

```
struct address
{
        char *name;
        long int number;
        char * street;
        char * town;
        char state[2];
        long zip;
 };
```

This defines a new type called address consisting of the items you need in order to send mail to someone. Note the semicolon at the end. This is one of very few places in C++ where it is necessary to have a semicolon after a curly brace, so people are prone to forget it. Variables of type address can be declared exactly as other variables, and the individual members can be accessed using the **.** ***(dot)*** operator. For example:

```
void f()
{
        address jd;
        jd.name = “Pratap”;
        jd.number = 11;
}
```

Structure objects are often accessed through pointers using the `->` (structure pointer dereference) operator. For example:

```
void print_addr(address * p)
{
        cout  << p->name << ‘\n’
              <<  p->number  <<  ‘  ’  <<  p->street  <<
        ‘\n’
              << p->town << ‘\n’
              << p->state[0] << p->state[1] << ‘’ <<
        p->zip << ‘\n’;
}
```

When p is a pointer, p `->` m is equivalent to (*p).m.

Objects of structure types can be assigned, passed as function arguments, and returned as the result from a function. For example:

```
address current;
address set_current (address next)
{
        address prev = current;
        current = next;
        return prev;
}
```

Other plausible operations, such as comparison (== and !=), are not defined. However, the user can define such operators.

A ***struct*** is a simple form of a ***class***.

4.4 OPERATORS

This section introduces the built-in C++ operators for composing expressions. C++ provides operators for composing arithmetic, relational, logical, bitwise, and conditional expressions. It also provides operators which produce useful side-effects, such as assignment, increment, and decrement. We will look at each category of operators in turn and also discuss the precedence rules which govern the order of operator evaluation in a multi - operator expression.

4.4.1 Arithmetic Operators:

C++ provides five basic *arithmetic operators*. These are summarized in the table below.

Operator	Name
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo (Remainder)

Except for modulo (%) all other arithmetic operators can accept a mix of integer and real operands. Generally, if both operands are integers then the result will be an integer. However, if one or both of the operands are reals then the result will be a real (or double to be exact).

Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators. The only one that you might not be so used to see is modulo; whose operator is the percentage sign (%). Modulo is the operation that gives the remainder of a division of two values. The remainder operator (%) expects integers for both of its operands. It returns the remainder of integer-dividing the operands. For example 13%3 is calculated by integer dividing 13 by 3 to give an outcome of 4 and a remainder of 1; the result is therefore 1.

4.4.2 Relational Operators:

C++ provides six *relational operators* for comparing numeric quantities. These are summarized in the table below. Relational operators evaluate to 1 (representing the true outcome) or 0 (representing the false outcome).

Operator	Name	Example
==	Equality	5 == 5 //gives 1
!=	Inequality	5 != 5 //gives 0
<	Less Than	5 < 5.5 //gives 1
<=	Less Than or Equal	5 <= 5 //gives 1
>	Greater Than	5 > 5.5 //gives 0
>=	Greater Than or Equal	5 >= 6 //gives 0

Note that the <= and >= operators are only supported in the form shown. In particular, =< and => are both invalid and do not mean anything.

The relational operators should not be used for comparing strings, because this will result in the string addresses being compared, not the string contents. For example, the expression
"HELLO" < "BYE"

causes the address of "HELLO" to be compared to the address of "BYE". As these addresses are determined by the compiler (in a machine-dependent manner), the outcome may be 0 or may be 1, and is therefore undefined. C++ provides library functions (e.g., ***strcmp***) for the lexicographic comparison of string. These will be described later in the book.

4.4.3 Logical Operators:

C++ provides three *logical operators* for combining logical expression. These are summarized in the table below. Like the relational operators, logical operators evaluate to 1 or 0.

Operator	Name
!	Logical Negation
&&	Logical And
||	Logical Or

Logical ***negation*** is a unary operator, which negates the logical value of its single operand. If its operand is nonzero it produces 0, and if it is 0 it produces 1. Logical ***and*** produces 0 if one or both of its operands evaluate to 0. Otherwise, it produces 1. Logical ***or*** produces 0 if both of its operands evaluate to 0. Otherwise, it produces 1.

The following panels show the result of Logical And and Or operators evaluating two expressions 'a' and 'b':

a	b	a && b
True	True	True
True	False	False
False	True	False
False	False	False

a	b	a || b
True	True	True
True	False	True
False	True	True
False	False	False

4.4.4 Bitwise Operators:

C++ provides six *bitwise operators* for manipulating the individual bits in an integer quantity. These are summarized in the table below.

Operator	Name
~	Bitwise Negation
&	Bitwise And
|	Bitwise Or
^	Bitwise Exclusive Or
<<	Bitwise Left Shift
>>	Bitwise Right Shift

Bitwise operators expect their operands to be integer quantities and treat them as bit sequences. Bitwise **negation** is a unary operator which reverses the bits in its operands. Bitwise **and** compares the corresponding bits of its operands and produces a 1 when both bits are 1, and 0 otherwise. Bitwise **or** compares the corresponding bits of its operands and produces a 0 when both bits are 0, and 1otherwise. Bitwise **exclusive or** compares the corresponding bits of its operands and produces a 0 when both bits are 1 or both bits are 0, and 1 otherwise.

Bitwise **left shift** operator and bitwise **right shift** operator both take a bit sequence as their left operand and a positive integer quantity 'n' as their right operand. The former produces a bit sequence equal to the left operand but which has been shifted 'n' bit positions to the left. The latter produces a bit sequence equal to the left operand but which has been shifted 'n' bit positions to the right. Vacated bits at either end are set to 0.

4.4.5 Increment/Decrement Operators:

The *auto increment (++)* and *auto decrement (--)* operators provide a convenient way of, respectively, adding and subtracting 1 from a numeric variable. These are summarized in the table below. The examples assume the following variable definition:

```
int k = 5;
```

Operator	Name	Example
++	Auto Increment (prefix)	++k + 10 //gives 16
++	Auto Increment (postfix)	k++ + 10 //gives 15
--	Auto Decrement (prefix)	--k + 10 //gives 14
--	Auto Decrement (postfix)	k-- + 10 //gives 15

Both operators can be used in prefix and postfix form. The difference is significant. When used in prefix form, the operator is first applied and the outcome is then used in the expression. When used in the postfix form, the expression is evaluated first and then the operator applied.

4.4.6 Assignment Operator:

The *assignment operator* is used for storing a value at some memory location (typically denoted by a variable). Its left operand should be an lvalue, and its right operand may be an arbitrary expression. The latter is evaluated and the outcome is stored in the location denoted by the lvalue. An lvalue (standing for left value) is anything that denotes a memory location in which a value may be stored. The kind of lvalue we have seen so far is a variable, pointers and references.

The assignment operator has a number of variants, obtained by combining it with the arithmetic and bitwise operators. These are summarized in the table below. The examples assume that ***n*** is an integer variable.

Operator	Example	Equivalent to
=	n = 25	
+=	n += 25	n = n+25
-=	n -= 25	n = n – 25
*=	n *= 25	n = n * 25
/=	n /= 25	n = n / 25
%=	n %= 25	n = n % 25
&=	n &= 0xF2F2	n = n & 0xF2F2
|=	n |= 0xF2F2	n = n | 0xF2F2
^=	n ^= 0xF2F2	n = n ^ 0xF2F2
<<=	n <<= 0xF2F2	n = n << 0xF2F2

`>>=`	`n >>= 0xF2F2`	`n = n >> 0xF2F2`

7. Conditional Operator:

The *conditional operator* takes three operands. It has the general form:

```
        operand1 ? operand2 : operand3
```

First operand1 is evaluated, which is treated as a logical condition. If the result is nonzero then operand2 is evaluated and its value is the final result. Otherwise, operand3 is evaluated and its value is the final result. For example:

```
int m = 1, n = 2;
int min = (m < n ? m : n); // min receives 1
```

Note that of the second and the third operands of the conditional operator only one is evaluated.

8. Comma Operator:

Multiple expressions can be combined into one expression using the *comma operator*. The comma operator takes two operands. It first evaluates the left operand and then the right operand, and returns the value of the latter as the final outcome. For example:

```
int m, n, min;
int mCount = 0, nCount = 0;
//...
min = (m < n ? mCount++, m : nCount++, n);
```

Here when m is less than n, mCount++ is evaluated and the value of m is stored in min. Otherwise, nCount++ is evaluated and the value of n is stored in min.

9. Scope Resolution Operator:

C++, like C, is a block – structured language. Blocks and scopes can be used in constructing programs. We know that the same variable can be used to have different meanings in different blocks. In C, the global version of a variable cannot be accessed from within the inner block. C++ resolves this problem by introducing a new operator **::** called the *scope resolution operator*. This can be used to uncover a hidden variable. It takes the following form:

```
:: variable-name
```

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

```
int main()
{
        int m = 20;        //m declared, local to main
        {
                int m = 10; //m  declared  again,  local
to inner block
                cout << “m = ” << m << “\n”;
                cout << “:: m = ” << ::m << “\n”;
        }
}
```

The output of the above program will be

```
m = 10
m = 20
```

A major application of the scope resolution operator is in the classes to identify the class to which a member function belongs. This will be dealt in detail later when classes are introduced.

4.4.10 Member Dereferencing Operators:

C++ permits us to define a class containing various types of data and functions as members. C++ also permits us to access the class members through pointers. In order to achieve this, C++ provides a set of three pointer – to – member operators. The following table shows these operators and their functions.

Operator	Function
`::*`	To declare a pointer to a member of a class
`*`	To access a member using object name and a pointer to that member
`->*`	To access a member using a pointer to the object and a pointer to that member

Further details on these operators will be meaningful only after we discuss classes, and therefore we defer the use of member dereferencing operators until then.

11. Memory Management Operators:

C uses **malloc**() and **calloc**() functions to allocate memory dynamically at run time. Similarly, it uses the function **free**() to free dynamically allocated memory. Although C++ supports these functions, it also defines two unary operators **new** and **delete** that perform the task of allocating and freeing the memory in a better and easier way.

An object can be created by using **new**, and destroyed by using **delete**, as and when required. A data object created inside a block with new, will remain in existence until it is explicitly destroyed by using delete. Thus, lifetime of an object is directly under our control and is unrelated to the block structure of the program.

The **new** operator can be used to create objects of any type. It takes the following general form:

```
        pointer – variable = new data – type;
```

Examples:

```
        p = new int;
        q = new float;
```

where p is a pointer of type int and q is a pointer of type float.
When a data object is no longer needed, it is destroyed to release the memory space for reuse. The general form of its use is:

```
        delete pointer – variable;
```

For example:

```
        delete p;
```

12. Type Cast Operator:

C++ permits explicit type conversion of variables or expressions using the *type cast operator*. The following two versions are equivalent:

```
        (type – name) expression        // C notation
        type – name (expression)        // C++ notation
```

Examples:

```
        average = sum / (float) i;      //C notation
        average = sum / float (i);      //C++ notation
```

ANSI C++ adds the following new cast operators:

- ***const_cast***
- ***static_cast***
- ***dynamic_cast***

- ➢ ***reinterpret_cast***

Application of these operators is discussed later.

4.5 OPERATOR PRECEDENCE

When writing complex expressions with several operands, we may have some doubts about which operand is evaluated first and which later. There is an established order with the priority of each operator, and not only the arithmetic ones (those whose preference come from mathematics) but for all the operators which can appear in C++. From ***greatest to lowest priority***, the priority order is as follows:

Level	Operator	Description	Grouping
1	::	scope	Left-to-right
2	() [] . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid	postfix	Left-to-right
3	++ -- ~ ! sizeof new delete	unary (prefix)	Right-to-left
	* &	indirection and reference (pointers)	
	+ -	unary sign operator	
4	(type)	type casting	Right-to-left
5	.* ->*	pointer-to-member	Left-to-right
6	* / %	multiplicative	Left-to-right
7	+ -	additive	Left-to-right
8	<< >>	shift	Left-to-right
9	< > <= >=	relational	Left-to-right
10	== !=	equality	Left-to-right
11	&	bitwise AND	Left-to-right
12	^	bitwise XOR	Left-to-right
13	|	bitwise OR	Left-to-right
14	&&	logical AND	Left-to-right
15	||	logical OR	Left-to-right
16	? :	conditional	Right-to-left
17	= *= /= %= += -= >>= <<= &= ^= |=	assignment	Right-to-left
18	,	comma	Left-to-right

Grouping defines the precedence order in which operators are evaluated in the case that there are several operators of the same level in an expression. All these precedence levels for operators can be manipulated or become more legible by removing possible ambiguities using ***parentheses signs (())***.

6. Expressions

An expression is any computation which yields a value. When discussing expressions, we often use the term evaluation. For example, we say that an expression evaluates to a certain value. Usually the final value is the only reason for evaluating the expression. However, in some cases, the expression may also produce side-effects. These are permanent changes in the program state. In this sense, C++ expressions are different from mathematical expressions. Expressions may be of the following seven types:

- ➢ Constant expressions
- ➢ Integral expressions
- ➢ Float expressions
- ➢ Pointer expressions
- ➢ Relational expressions
- ➢ Logical expressions
- ➢ Bitwise expressions

An expression may also use combinations of the above expressions. Such expressions are known as compound expressions

1. Constant Expressions:

Constant expressions consist of only constant values. Examples:

```
15
20 + 5 / 2.0
```

2. Integral Expressions:

Integral expressions are those which produce integer results after implementing all the automatic and explicit type conversions. Examples:

```
m + n
m * n – 5
5 + int(2.3)
```

where m and n are integer variables.

3. Float Expressions:

Float expressions are those which, after all conversions, produce floating – point results. Examples:

```
x * y / 10
5 + float(10)
```

where x and y are floating – point variables.

4. Pointer Expressions:

Pointer expressions produce address values. Examples:

```
&m
ptr + 1
```

where m is a variable and ptr is a pointer.

4.6.5 Relational Expressions:

Relational expressions yield results of type *bool* which takes a value *true* or *false*. Examples:

```
x <= y
a+b == c+d
```

When arithmetic expressions are used on either side of a relational operator, they will be evaluated first and then the results compared. Relational expressions are also known as ***Boolean expressions.***

6. Logical Expressions:

Logical expressions combine two or more relational expressions and produce *bool* type results. Examples:

```
a>b && x==10
x==10 || y==5
```

7. Bitwise Expressions:

Bitwise expressions are used to manipulate data at a bit level. They are basically used for testing or shifting bits. Examples:

```
x << 3            // Shift three bit position to
left
y >> 1            // Shift one bit position to
right
```

Shift operators are often used for multiplication and division by powers of two.

7. SUMMARY

- Standard C++ provides two different data types – **built-in** types and **user-defined** types.
- The type **`int`** is used for whole numbers which are represented exactly within the computer.
- The type **`float`** is used for real (decimal) numbers. They are held to a limited accuracy within the computer.
- The type **`char`** is used to represent single characters. A **`char`** constant is enclosed in single quotation marks.
- Literal strings can be used in output statements and are represented by enclosing the characters of the string in double quotation marks `"`.
- C++ provides an additional use of **void**, for declaration of generic pointers.
- The **enumerated** data types differ slightly in C++. The tag names of the enumerated data types become new type names. That is, we can declare new variables using these tag names.
- **Pointers** are widely used in C++ for memory management and to achieve polymorphism.
- C++ is very strict regarding type checking of variables. It does not allow to equate variables of two different data types. The only way to break this rule is **type casting**.
- A major application of the **scope resolution (::) operator** is in the classes to identify the class to which a member function belongs.

- C++ provides two new unary operators, in addition to **malloc**(), **calloc**() and **free**() functions, **new** and **delete** to perform the task of allocating and freeing the memory in a better and easier way.
- The order of evaluation of an expression is determined by the precedence of the operators.
- C++ supports seven types of **expressions**. When data types are mixed in an expression, C++ performs the conversion automatically using certain rules.
- When **`float`** expressions are assigned to **`int`** variables there may be loss of accuracy.
- C++ also permits **explicit type conversion** of variables and expressions using the type **cast operators**.

8. Unit End Exercises

1. Questions

These questions are intended as a self-test for readers.

1) An unsigned int can be twice as large as the signed int. Explain how?
2) What are the applications of void data type in C++?
3) Describe the implementation of enum data type in C++?
4) In C++, a variable can be declared anywhere in the scope. What is the significance of this feature?
5) What is an operator?
6) What is the application of the scope resolution operator in C++?
7) What data types would you use to represent the following items?
    a. the number of students in a class
    b. the grade (a letter) attained by a student in the class
    c. the average mark in a class
    d. the distance between two points
    e. the population of a city
    f. the weight of a postage stamp
    g. the registration letter of a car

2. Programming Projects:

Writing programs that solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied.

1) What is wrong with the following code fragment?

```
a. enum Season { SPRING, SUMMER, FALL, WINTER };

b. enum Semester { FALL, SPRING, SUMMER };
```

2) Write declaration statements to declare integer variables ***i*** and ***j*** and float variables ***x*** and ***y***. Extend your declaration statements so that ***i*** and ***j*** are both initialised to 1 and ***y*** is initialised to 10.0.

3) Find errors, if any, in the following C++ statements.

   a. long float x;

   b. char *cp = vp; //vp is a void pointer

   c. int code = three; //three is an enumerator

   d. int *p = new; //allocate memory with new

   e. enum {green, yellow, red};

   f. int const *p = total;

   g. const int array_size;

   h. int &number = 100;

   i. float *p = new int[10];

   j. char name[3] = "USA";

4) To what do the following expressions evaluate?

   a. 17/3
   b. 17%3
   c. 1/2
   d. 1/2*(x+y)

5) Given the declarations:

```
float x;
int k, i = 5, j = 2;
```

To what would the variables `x` and `k` be set as a result of the assignments

```
a. k = i/j;
b. x = i/j;
c. k = i%j;
```

d. `x = 5.0/j;`

4.9 FURTHER READING

Bjarne Stroustrup, "The C++ Programming Language: Second Edition".

Herbert Schildt, "The C++ Complete Reference"

E Balagurusamy, "Object Oriented Programming C++", Third Edition.
John Hubbard "Fundamentals of Computing with C++"

5

Input and Output

Unit Structure

1. OBJECTIVES

After completing this chapter you will be able to:

- ➢ Understand the Standard Input and Output stream
- ➢ Identify and use escape characters
- ➢ Understand Preprocessor directives
- ➢ Understand and use namespaces.

2. INTRODUCTION

Until now, the example programs of previous sections provided very little interaction with the user, if any at all. Using the standard input and output library, we will be able to interact with the user by printing messages on the screen and getting the user's input from the keyboard.

C++ uses a convenient abstraction called ***streams*** to perform input and output operations in sequential media such as the screen or the keyboard. A ***stream*** is an object where a program can either insert or extract characters to/from it. We do not really need to care about many specifications about the physical media associated with the stream - we only need to know it will accept or provide characters sequentially.

The most common way in which a program communicates with the outside world is through simple, character-oriented input/output (IO) operations. C++ provides two useful operators for this purpose: >> for input and << for output. The standard C++ library includes the header file ***iostream***, where the standard input and output stream objects are declared.

3. STANDARD OUTPUT (COUT)

By default, the standard output of a program is the screen, and the C++ stream object defined to access it is ***cout***. ***cout*** is used in conjunction with the ***insertion operator***, which is written as << (two "less than" signs).

```
cout << "Output sentence";     //   prints   Output
sentence on screen
cout << 120;                   // prints number 120
on screen
cout << x;                     // prints the content
of x on screen
```

The << operator inserts the data that follows it into the stream preceding it. In the examples above it inserted the constant string Output sentence, the numerical constant 120 and variable x into the standard output stream cout. Notice that the sentence in the first instruction is enclosed between double quotes (") because it is a constant string of characters. Whenever we want to use constant strings of characters we must enclose them between double quotes (") so that they can be clearly distinguished from variable names. For example, these two sentences have very different results:

```
cout << "Hello";        // prints Hello
cout << Hello;          //  prints  the  content  of
Hello variable
```

The insertion operator (<<) may be used more than once in a single statement. The utility of repeating the insertion operator (<<) is demonstrated when we want to print out a combination of variables and constants or more than one variable:

```
cout << "Hello, I am " << age << " years old and my
postal code is " << pincode;
```

If we assume the age variable to contain the value 24 and the pincode variable to contain 400054 the output of the previous statement would be:

```
    Hello, I am 24 years old and my postal code is
400054
```

It is important to notice that ***cout*** does not add a line break after its output unless we explicitly indicate it, therefore, the following statements:

```
    cout << "This is a sentence.";
    cout << "This is another sentence.";
```

will be shown on the screen one following the other without any line break between them:

```
    This is a sentence.This is another sentence.
```

even though we had written them in two different insertions into cout. In order to perform a line break on the output we must explicitly insert a new-line character into cout. In C++ a new-line character can be specified as ***\n (backslash, n)***:

```
    cout << "First sentence.\n ";
    cout << "Second sentence.\nThird sentence.";
```

This produces the following output:

```
    First sentence.
    Second sentence.
    Third sentence.
```

Additionally, to add a new-line, you may also use the ***endl*** manipulator. For example:

```
    cout << "First sentence." << endl;
    cout << "Second sentence." << endl;
```

would print out:

```
    First sentence.
    Second sentence.
```

The ***endl*** manipulator produces a newline character, exactly as the insertion of '***\n***' does, but it also has an additional behaviour when it is used with buffered streams: the buffer is flushed. Anyway, cout will be an unbuffered stream in most cases, so you can generally use both the ***\n*** escape character and the ***endl*** manipulator in order to specify a new line without any difference in its behaviour.

~~5.4 STANDARD INPUT (CIN)~~

The standard input device is usually the keyboard. Handling the standard input in C++ is done by applying the overloaded operator of extraction (>>) on the ***cin*** stream. The operator must be followed by the variable that will store the data that is going to be extracted from the stream. For example:

```
    int age;
    cin >> age;
```

The first statement declares a variable of type int called age, and the second one waits for an input from cin (the keyboard) in order to store it in this integer variable.

cin can only process the input from the keyboard once the ***RETURN*** key has been pressed. Therefore, even if you request a single character, the extraction from cin will not process the input until the user presses RETURN after the character has been introduced.

You must always consider the type of the variable that you are using as a container with cin extractions. If you request an integer you will get an integer, if you request a character you will get a character and if you request a string of characters you will get a string of characters.

```
// i/o example
#include <iostream>
using namespace std;
 int main ()
{
      int i;
      cout << "Please enter an integer value: ";
      cin >> i;
      cout << "The value you entered is " << i; cout
      << " and its double is " << i*2 <<
".\n";
      return 0;
}
Output:
Please enter an integer value: 702
The value you entered is 702 and its double is
1404.
```

The user of a program may be one of the factors that generate errors even in the simplest programs that use ***cin*** (like the one we have just seen). Since if you request an integer value and the user introduces a name (which generally is a string of characters), the result may cause your program to misoperate since it is not what we were expecting from the user. So when you use the data input provided by cin extractions you will have to trust that the user of your program will be cooperative and that he/she will not introduce his/her name or something similar when an integer value is requested. You can also use ***cin*** to request more than one datum input from the user:

```
cin >> a >> b;
```

is equivalent to:

```
cin >> a;
cin >> b;
```

In both cases the user must give two data, one for variable **a** and another one for variable **b** that may be separated by any valid blank separator: a space, a tab character or a newline.

5.4.1 cin and Strings:

We can use cin to get strings with the extraction operator (>>) as we do with fundamental data type variables:

```
cin >> mystring;
```

However, as it has been said, cin extraction stops reading as soon as if finds any blank space character, so in this case we will be able to get just one word for each extraction. This behaviour may or may not be what we want; for example if we want to get a sentence from the user, this extraction operation would not be useful. In order to get entire lines, we can use the function ***getline***, which is the more recommendable way to get user input with cin.

```
// cin with strings
```

```
#include <iostream>
#include <string>
using namespace std;
 int main ()
{
      string mystr;
      cout << "What's your name? ";
      getline (cin, mystr);
      cout << "Hello " << mystr << ".\n";
      return 0;
}
Output:
What's your name? Prakash Pratap Singh
Hello Prakash Pratap Singh.
```

Notice how we used the ***getline*** function with ***cin*** to get a complete line using the string identifier (***mystr***).

5.5 ESCAPE CHARACTERS

Escape characters are special characters that are difficult or impossible to express otherwise in the source code of a program, like ***newline*** (**\n**) or ***tab*** (**\t**). All of them are preceded by a ***backslash*** (\). Here you have a list of some of such escape codes:

Name	**C++ Name**
Newline	\n
Horizontal tab	\t
Vertical tab	\v
Backspace	\b
Carriage return	\r
Form feed	\f
Alert	\a
Backslash	\\
Question mark	\?
Single quote	\’
Double quote	\”
Octal number	\ooo
Hexadecimal number	\xhhh

Despite their appearances, these are single characters.
For example:

```
'\n'
'\t'
"Left \t Right"
"one\ntwo\nthree"
```

Additionally, you can express any character by its numerical ASCII code by writing a backslash (\) followed by the ASCII code expressed as an octal (base-8) or hexadecimal (base-16) number. In the first case (octal) the digits must immediately follow the backslash (for example \23 or \40), in the second case (hexadecimal), an x character must be written before the digits themselves (for example \x20 or \x4A).

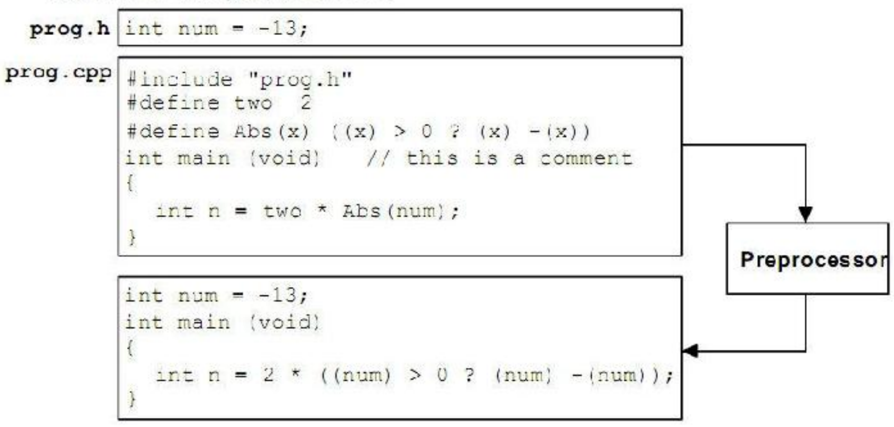
Decimal	Octal	Hexadecimal
6	‘\6’	‘\x6’

48	‘\60’	‘\x30’
95	‘\137’	‘\x05F’

5.6 PREPROCESS OR DIRECTIVES

Prior to compiling a program source file, the C++ compiler passes the file through a **preprocessor**. The role of the **preprocessor** is to transform the source file into an equivalent file by performing the preprocessing instructions contained by it. These instructions facilitate a number of features, such as: file inclusion, conditional compilation, and macro substitution. The figure below illustrates the effect of the preprocessor on a simple file.

The role of the preprocessor.



```
int num = -13;
```

```
#include "prog.h"
#define two  2
#define Abs(x)  ((x) > 0 ? (x) -(x))
int main (void)    // this is a comment
{
   int n = two * Abs(num);
}
```

```
int num = -13;
int main (void)
{
   int n = 2 * ((num) > 0 ? (num) -(num));
}
```

The **preprocessor** performs very minimal error checking of the preprocessing instructions. Because it operates at a text level, it is unable to check for any sort of language-level syntax errors. This function is performed by the compiler.

The preprocessor is executed before the actual compilation of code begins; therefore the preprocessor digests all these directives before any code is generated by the statements. These preprocessor directives extend only across a single line of code. As soon as a newline character is found, the preprocessor directive is considered to end. No semicolon (;) is expected at the end of a preprocessor directive.

5.6.1 Macro definitions (#define, #undef):

To define preprocessor **macros** we can use **#define**. Its format is:

```
#define identifier replacement
```

When the preprocessor encounters this directive, it replaces any occurrence of ***identifier*** in the rest of the code by ***replacement***. This ***replacement*** can be an expression, a statement, a block or simply anything. The preprocessor does not understand C++; it simply replaces any occurrence of identifier by replacement.

```
#define TABLE_SIZE 100
int table1[TABLE_SIZE];
int table2[TABLE_SIZE];
```

After the preprocessor has replaced TABLE_SIZE, the code becomes equivalent to:

```
int table1[100];
int table2[100];
```

Consider the example given below:

```
// function macro
#include <iostream>
using namespace std;
 #define getmax(a,b) ((a)>(b)?(a):(b))
 int main()
{
        int x=5, y;
        y= getmax(x,2);
        cout << y << endl;
        cout << getmax(7,x) << endl;
        return 0;
}
Output:
5
7
```

#define can work also with parameters to define function macros. Any occurrence of getmax followed by two arguments is replaced by the replacement expression, also replacing each argument by its identifier, exactly as you would expect if it was a function.

Defined macros are not affected by block structure. A macro lasts until it is undefined with the **#undef** preprocessor directive.

```
#define TABLE_SIZE 100
int table1[TABLE_SIZE];
#undef TABLE_SIZE
#define TABLE_SIZE 200
int table2[TABLE_SIZE];
```

This would generate the same code as:

```
int table1[100];
int table2[200];
```

Because preprocessor replacements happen before any C++ syntax check, macro definitions can be a tricky feature, but be careful: code that relies heavily on complicated macros may result obscure to other programmers, since the syntax they expect is on many occasions different from the regular expressions programmers expect in C++.

5.6.2 Conditional inclusions (#ifdef, #ifndef, #if, #endif, #else and #elif):

These directives allow including or discarding part of the code of a program if a certain condition is met. **#ifdef** allows a section of a program to be compiled only if the macro that is specified as the parameter has been defined, no matter which its value is. For example:

```
#ifdef TABLE_SIZE
int table[TABLE_SIZE];
#endif
```

In this case, the line of code int table [TABLE_SIZE]; is only compiled if TABLE_SIZE was previously defined with #define, independently of its value. If it was not defined, that line will not be included in the program compilation.

#ifndef serves for the exact opposite: the code between **#ifndef** and **#endif** directives is only compiled if the specified identifier has not been previously defined. For example:

```
#ifndef TABLE_SIZE
#define TABLE_SIZE 100
#endif
int table[TABLE_SIZE];
```

In this case, if when arriving at this piece of code, the TABLE_SIZE macro has not been defined yet, it would be defined to a value of 100. If it already existed it would keep its previous value since the #define directive would not be executed.

The **#if**, **#else** and **#elif** (i.e., "else if") directives serve to specify some condition to be met in order for the portion of code they surround to be compiled. The condition that follows **#if** or **#elif** can only evaluate constant expressions, including macro expressions. For example:

```
#if TABLE_SIZE>200
      #undef TABLE_SIZE
      #define TABLE_SIZE 200
  #elif TABLE_SIZE<50
      #undef TABLE_SIZE
      #define TABLE_SIZE 50
  #else
      #undef TABLE_SIZE
      #define TABLE_SIZE 100
#endif
int table[TABLE_SIZE];
```

Notice how the whole structure of **#if**, **#elif** and **#else** chained directives ends with **#endif**.

The table below summarizes the general forms of these directives (code denotes zero or more lines of program text, and expression denotes a constant expression).

General Form of Conditional Inclusion Directives:

Form	**Explanation**

`#ifdef identifier`
`code`
`#endif`
	If identifier is a **#defined** symbol then code is included in the compilation process. Otherwise, it is excluded.

`#ifndef identifier`
`code`
`#endif`
	If identifier is not a **#defined** symbol then code is included in the compilation process. Otherwise, it is excluded.

`#if expression`
`code`
`#endif`
	If expression evaluates to nonzero then code is included in the compilation process. Otherwise, it is excluded.

`#ifdef identifier`
`code1`
`#else`
`code2`
`#endif`
	If identifier is a **#defined** symbol then code1 is included in the compilation process and code2 is excluded. Otherwise, code2 is included and code1 is excluded.
	Similarly, **#else** can be used with **#ifndef** and **#if**.

`#if expression1`	If expression1 evaluates to nonzero then only

`code1`
`#elif expression2`
`code2`
`#else`
`code3`
`#endif`
	code1 is included in the compilation process. Otherwise, if expression2 evaluates to nonzero then only code2 is included. Otherwise, code3 is included.
	As before, the **#else** part is optional. Also, any number of **#elif** directives may appear after **#if** directive.

3. Source file inclusion (#include):

This directive has been used assiduously in previous sections of this book. When the preprocessor finds an **#include** directive it replaces it by the entire content of the specified file. There are two ways to specify a file to be included:

```
#include "file"
#include <file>
```

The only difference between both expressions is the places (directories) where the compiler is going to look for the file. In the first case where the file name is specified between **double-quotes**, the file is searched first in the same directory that includes the file containing the directive. In case that it is not there, the compiler searches the file in the default directories where it is configured to look for the standard header files.

If the file name is enclosed between **angle-brackets** <> the file is searched directly where the compiler is configured to look for the standard header files. Therefore, standard header files are usually included in angle-brackets, while other specific header files are included using quotes.

4. Pragma directive (#pragma):

This directive is used to specify diverse options to the compiler. These options are specific for the platform and the compiler you use. Consult the manual or the reference of your compiler for more information on the possible parameters that you can define with **#pragma**. If the compiler does not support a specific argument for **#pragma**, it is ignored - no error is generated.

5. Error directive (#error):

This directive aborts the compilation process when it is found, generating a compilation the error that can be specified as its parameter:

```
#ifndef __cplusplus
      #error A C++ compiler is required!
#endif
```

This example aborts the compilation process if the macro name **__cplusplus** is not defined (this macro name is defined by default in all C++ compilers).

5.7 NAMESPACES

A namespace is a mechanism for expressing logical grouping. That is, if some declarations logically belong together according to some criteria, they can be put in a common namespace to express that fact. Namespaces allow to group entities like classes, objects and functions under a name. This way the global scope can be divided in "sub-scopes", each one with its own name. The format of namespaces is:

```
namespace identifier
{
      entities
}
```

Where identifier is any valid identifier and entities is the set of classes, objects and functions that are included within the namespace. For example:

```
namespace myNamespace
{
  int a, b;
}
```

In this case, the variables **a** and **b** are normal variables declared within a namespace called myNamespace. In order to access these variables from outside the myNamespace namespace we have to use the scope operator ::. For example, to access the previous variables from outside myNamespace we can write:

```
myNamespace::a
myNamespace::b
```

The functionality of namespaces is especially useful in the case that there is a possibility that a global object or function uses the same identifier as another one, causing redefinition errors. For example:

```
// namespaces
#include <iostream>
using namespace std;
 namespace first
{
      int var = 5;
}
 namespace second
{
      double var = 3.1416;
}
 int main ()
{
      cout << first::var << endl;
      cout << second::var << endl;
      return 0;
}
Output:
5
3.1416
```

In this case, there are two global variables with the same name: **var**. One is defined within the namespace **first** and the other one in **second**. No redefinition errors happen thanks to namespaces.

A **namespace** is a scope. Thus, ‘‘namespace’’ is a very fundamental and relatively simple concept. The larger a program is, the more useful namespaces are to express logical separations of its parts. Ordinary local scopes, global scopes, and classes are namespaces.

Ideally, every entity in a program belongs to some recognizable logical unit (‘‘module’’). Therefore, every declaration in a nontrivial program should ideally be in some namespace named to indicate its logical role in the program. The exception is **main()**, which must be global in order for the run-time environment to recognize it as special.

5.7.1 Using Declaration:

When a name is frequently used outside its namespace, it can be a bother to repeatedly qualify it with its namespace name. The redundancy can be eliminated by a *using-declaration* to state in one place that the **member** used in the scope belongs to a **namespace**. The format of using-declaration is:

```
using name :: member;
```

where *name* refers to the name of the namespace.
For example:

```
// using
#include <iostream>
using namespace std;
namespace first
{
      int x = 5;
      int y = 10;
}
namespace second
{
      double x = 3.1416;
      double y = 2.7183;
}
int main ()
{
      using first::x;
      using second::y;
      cout << x << endl;
      cout << y << endl;
      cout << first::y << endl;
      cout << second::x << endl;
      return 0;
}
Output:
5
2.7183
10
3.1416
```

Notice how in this code, x (without any name qualifier) refers to first::x whereas y refers to second::y, exactly as our using declarations have specified. We still have access to first::y and second::x using their fully qualified names.

A using-declaration introduces a local synonym. It is often a good idea to keep local synonyms as local as possible to avoid confusion.

2. Using Directive:

The keyword using can also be used as a directive to introduce an entire namespace. A using-directive makes names from a namespace available almost as if they had been declared outside their namespace. The format of using-directive is:

```
using namespace identifier;
```

For example:

```
// using
#include <iostream>
using namespace std;
namespace first
{
      int x = 5;
      int y = 10;
}
namespace second
{
      double x = 3.1416;
double y = 2.7183;
}
int main ()
{
      using namespace first;
      cout << x << endl;
      cout << y << endl;
      cout << second::x << endl;
      cout << second::y << endl;
      return 0;
}
Output:
5
10
3.1416
2.7183
```

In this case, since we have declared that we were using ***namespace first***, all direct uses of **x** and **y** without name qualifiers was referring to their declarations in namespace **first**.

Global using-directives are a tool for transition and are otherwise best avoided. In a namespace, a using-directive is a tool for namespace composition. In a function (only), a using-directive can be safely used as a notational convenience.

using (*using-declaration*) and ***using namespace*** (*using-directive*) have validity only in the same block in which they are stated or in the entire code if they are used directly in the global scope.

3. Namespace std:

All the files in the C++ standard library declare all of its entities within the ***std*** namespace. That is why we have generally included the using namespace ***std***; statement in all programs that used any entity defined in ***iostream***.

5.8 COMMENTS AND INDENTATION

A comment is a piece of descriptive text which explains some aspect of a program. Program comments are totally ignored by the compiler and are only intended for human readers. C++ provides two types of comment delimiters:

- ➢ Anything after // (until the end of the line on which it appears) is considered a comment.
- ➢ Anything enclosed by the pair /* and */ is considered a comment.

The following program illustrates the use of both forms:

```
#include <iostream.h>
/* This program calculates the weekly gross pay for
a worker based on the total number of hours worked
and the hourly pay rate. */
int main (void)
{
        int workDays = 5; // Number of work days per
        week
        float workHours = 7.5; // Number of work
        hours per day
        float payRate = 33.50; // Hourly pay rate
        float weeklyPay; // Gross weekly pay
        weeklyPay = workDays * workHours * payRate;
        cout << "Weekly Pay = " << weeklyPay << '\n';
}
```

Judicious use of comments and consistent use of indentation can make the task of reading and understanding a program much more pleasant. Several different consistent styles of indentation are in use. I see no fundamental reason to prefer one over another (although, like most programmers, I have my preferences). The same applies to styles of comments.

Comments should be used to enhance (not to hinder) the readability of a program. The following points, in particular, should be noted:

- ➢ A comment should be easier to read and understand than the code which it tries to explain. A confusing or unnecessarily-complex comment is worse than no comment at all.
- ➢ Over-use of comments can lead to even less readability. A program which contains so much comment that you can hardly see the code can by no means be considered readable.
- ➢ Use of descriptive names for variables and other entities in a program, and proper indentation of the code can reduce the need for using comments.

Once something has been stated clearly in the language, it should not be mentioned a second time in a comment. For example:

```
a = b+c;        // a becomes b+c
count++;        // increment the counter
```

Such comments are worse than simply redundant. They increase the amount of text the reader has to look at, they often obscure the structure of the program, and they may be wrong. Note, however, that such comments are used extensively for teaching purposes in programming language textbooks such as this. This is one of the many ways a program in a textbook differs from a real program.

My preference is for:

1) A comment for each source file stating what the declarations in it have in common, references to manuals, general hints for maintenance, etc.
2) A comment for each class, template, and namespace
3) A comment for each nontrivial function stating its purpose, the algorithm used (unless it is obvious), and maybe something about the assumptions it makes about its environment
4) A comment for each global and namespace variable and constant
5) A few comments where the code is non-obvious and/or non-portable
6) Very little else.

A well-chosen and well-written set of comments is an essential part of a good program. Writing good comments can be as difficult as writing the program itself. It is an art well worth cultivating.

9. SUMMARY

- ➢ The standard output in C++ is done by applying the overloaded operator of insertion (<<) on the **cout** stream.
- ➢ The standard input in C++ is done by applying the overloaded operator of extraction (>>) on the **cin** stream.
- ➢ **Escape** characters are special characters preceded by a backslash (\).
- ➢ **Preprocessor directives** are lines included in the code of our programs that are not program statements but directives for the preprocessor. These lines are always preceded by a hash sign (#).
- ➢ Preprocessor directives are
  - ❖ Macro definitions - #define, and #undef
  - ❖ Conditional Inclusions - #ifdef, #ifndef, #if, #endif, #else, and #elif
  - ❖ Source File Inclusion - #include
  - ❖ Pragma Directive - #pragma
  - ❖ Error Directive - #error

- ➢ When you have a choice, prefer the standard library to other libraries.
- ➢ Do not think that the standard library is ideal for everything.
- ➢ Remember to **#include** the headers for the facilities you use.
- ➢ Remember that standard library facilities are defined in namespace **std**.
- ➢ Use **namespaces** to express logical structure.
- ➢ Place every nonlocal name, except main(), in some namespace.
- ➢ Use the **Namespace :: member** notation when defining namespace members.
- ➢ Use **using namespace** only for transition or within a local scope.
- ➢ Keep **comments** crisp.
- ➢ Maintain a consistent **indentation** style.

5.10 UNIT END EXERCISES

5.10.1 Questions:

These questions are intended as a self-test for readers.

1) Explain role of preprocessor?
2) What is a preprocessor directive? Explain them?
3) What is a namespace? Explain its functionality?
4) How does using-declaration differ from using-directive?
5) What is wrong with these macro

   definitions? a. #define PI = 3.141593;

   b. #define MAX(a,b) a>b ? a : b

   c. #define fac(a) (a)*fac((a)-1)
6) Write directives for the following:

   a. Defining **Small** as an **unsigned char** when the symbol **PC** is defined, and as **unsigned short** otherwise.

   b. Including the file **basics.h** in another file when the symbol **CPP** is not defined.

   c. Including the file **debug.h** in another file when **release** is 0, or **beta.h** when **release** is 1, or **final.h** when **release** is greater than 1.

5.10.2 Programming Projects:

1) Writing programs that solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied.

2) Write a program like ''Hello, world!'' that takes a name as a command-line argument and writes ''Hello, name!''. Modify this program to take any number of names as arguments and to say hello to each.

3) What will the following program output when executed?

```
#include <iostream.h>
char *str = "global";
void Print (char *str)
{
        cout << str << '\n';
        {
             char *str = "local";
             cout << str << '\n';
             cout << ::str << '\n';
        }
        cout << str << '\n';
}
int main (void)
{
        Print("Parameter");
        return 0;
}
```

5.11 FURTHER READING

Bjarne Stroustrup, "The C++ Programming Language: Second Edition".

Herbert Schildt, "The C++ Complete Reference"

E Balagurusamy, "Object Oriented Programming C++", Third Edition.
John Hubbard "Fundamentals of Computing with C++"

6

Chapter 6
Decisions

Unit Structure

1. Introduction

The execution of statements in a program by default is sequential. But sometimes there are such situations where the problem statement and the program logic demands that the program execution or flow be directed or branched to a particular set of statements rather than the other. Ex. Finding greater of 2 given numbers.

This kind of situation involves decision making. C++ offers the following decision control structures:

1. if
2. if-else
3. conditional operator
4. switch.

6.2 Decision Control Structure

6.2.1 if statement:

The syntax of if statement is as follows:

```
if (expression is true)
     execute statement;
```

- Here, if is a keyword. It tells the compiler that what follows is a decision control structure.
- An if statement is always followed by an expression or condition which is enclosed within a pair of parenthesis.
- The expression is evaluated and will be either true or false. If true then the statement following the if statement is executed, if false then this statement is skipped and the execution continues from next statement.
- Every non zero value will be considered true be it positive or negative.
- Expression or condition is usually a combination of variables and or constants and operators.
- Examples of expressions used in if statement:

  (a>b) // combination of variables & relational operator
  (x<5) // combination of variables & relational operator

- In general an expression is formed using relational operators. Relational operators are used to compare the values of two variables. We can use the relational operators to construct the following types of expressions:

Expression	**Is true if**
A >B	A is greater than B
A < B	A is less than B
A <= B	A is less than or equal to B
A >= B	A is greater than or equal to B
A == B	A is equal to B
A != B	A is not equal to B

- Example : The following program written in c++ prompts the user for a number. If the user enters the number 3 it prints a particular string, if the user enters any other number it prints nothing.

```
/*****************************************************
*******
Program 6.1.
Author : Nikhil Pawanikar
Description : Program to take one input from user. If number
equals to 3
```

```
                    Print a string on the screen, if not do nothing.
*******************************************************
*******/
#include<iostream.h>
#include<conio.h>
int main()
{
int a;
cout<<"enter the value of a\t";
cin>>a;
if(a==3)
        cout<<"A equals to 3";
getch();
return 0;
}
```

- **Output**

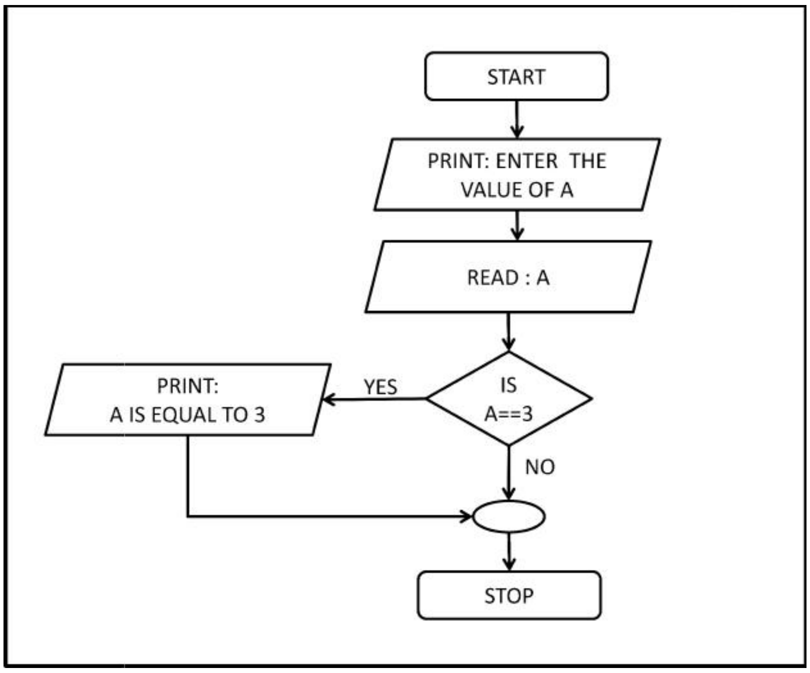
First Run:

```
enter the value of a          4               //nothing happens
```

Second Run:

```
enter the value of a          3
A equals to 3
```

- The above program and the execution of if statement will be better understood by the following flowchart:



- In case multiple statements are to be executed if the expression is true, those statements can be put inside a set of parenthesis { } . If the condition is true the statements in the block following the parenthesis is executed.
- Ex.

```
if(a<b)
{
cout<<"A is less than b";
a=a+1;
}
```

Note: A semicolon is not used with if statement after the } end bracket, it is used only with simple if statement that executes single instruction.

2. if-else statement:

- The general syntax of if – else statement is as follows

```
if (expression is true)
                execute statement;
else
                execute statement;
```

Note: single statement is executed if condition is true else single statement is excuted if condition is false

OR

```
if (expression is true)
{
                execute statement;
                execute statement;
                execute statement;
}
else
{
                execute statement;
                execute statement;
                execute statement;
}
```

Note: The Block of statements in parenthesis {} is executed if condition is true else the other block of statements following else is executed if condition is false.

- The if-else statement is slightly different version of if statement. Here if the condition is found to be false other statements are executed. If the condition or expression evaluates to true the statement (single)/ block of statements following if is executed, if it evaluates to false then the statement(single)/block of statement s following the keyword else is executed.
- The expressions/conditions are the same as described above in if statement and may be relational expressions.
- Example : Consider the following program that prompts the user to enter 2 numbers and prints the greater of two numbers on the screen.

```
/*****************************************************
*******
Program 6.2
Author : Nikhil Pawanikar
Description : Program to take two inputs from user and printe
the greater
               Of the two
*****************************************************
*******/

#include<iostream.h>
#include<conio.h>
```

Thank You