a different data object. Within the inner block, the variable **x** will refer to the data object declared therein.

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, ::count means the global version of the variable count (and not the local variable count declared in that block). Program 3.1 illustrates this feature.

SCOPE RESOLUTION OPERATOR

```
#include <iostream>

using namespace std;
int m = 10;            // global m

int main()
{
        int m = 20;    // m redeclared, local to main

        {
                int k = m;
                int m = 30;    // m declared again
                               // local to inner block

                cout <<  "we are in inner block \n";
                cout <<  "k = " << k << "\n";
                cout <<  "m = " << m << "\n";
                cout <<  "::m = " << ::m << "\n";
        }

        cout << "\nWe are in outer block \n";
        cout << "m = " << m << "\n";
        cout << "::m = " << ::m << "\n";

        return 0;
}
```

PROGRAM 3.1

The output of Program 3.1 would be:

```
We are in inner block
k = 20
```

```
m = 30
::m = 10

We are in outer block
m = 20
::m = 10
```

In the above program, the variable **m** is declared at three places, namely, outside the **main()** function, inside the **main()**, and inside the inner block.

note

It is to be noted **::m** will always refer to the global **m**. In the inner block, ::m refers to the value 10 and not 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 the classes are introduced.

3.15 Member Dereferencing Operators

As you know, 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. Table 3.3 shows these operators and their functions.

Table 3.3 *Member dereferencing operators*

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.

3.16 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. We use dynamic allocation techniques when it is not known in advance how much of memory space is needed. 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. Since these operators manipulate memory on the free store, they are also known as *free store* operators.

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, the 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;
```

Here, *pointer-variable* is a pointer of type *data-type*. The **new** operator allocates sufficient memory to hold a data object of type *data-type* and returns the address of the object. The *data-type* may be any valid data type. The *pointer-variable* holds the address of the memory space allocated. Examples:

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

where **p** is a pointer of type **int** and **q** is a pointer of type **float**. Here, **p** and **q** must have already been declared as pointers of appropriate types. Alternatively, we can combine the declaration of pointers and their assignments as follows:

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

Subsequently, the statements

```
*p = 25;
*q = 7.5;
```

assign 25 to the newly created **int** object and 7.5 to the **float** object.

We can also initialize the memory using the **new** operator. This is done as follows:

```
pointer-variable  =  new data-type(value);
```

Here, value specifies the initial value. Examples:

```
int *p = new int(25);
float *q = new float(7.5);
```

As mentioned earlier, **new** can be used to create a memory space for any data type including user-defined types such as arrays, structures and classes. The general form for a one-dimensional array is:

```
pointer-variable  =  new data-type[size];
```

Here, size specifies the number of elements in the array. For example, the statement

```
int *p = new int[10];
```

creates a memory space for an array of 10 integers. **p[0]** will refer to the first element, **p[1]** to the second element, and so on.

When creating multi-dimensional arrays with **new**, all the array sizes must be supplied.

```
array_ptr = new int[3][5][4];     // legal
array_ptr = new int[m][5][4];     // legal
array_ptr = new int[3][5][ ];     // illegal
array_ptr = new int[ ][5][4];     // illegal
```

The first dimension may be a variable whose value is supplied at runtime. All others must be constants.

The application of **new** to class objects will be discussed later in Chapter 6.

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;
```

The *pointer-variable* is the pointer that points to a data object created with **new**. Examples:

```
delete p;
delete q;
```

If we want to free a dynamically allocated array, we must use the following form of **delete**:

```
delete [size] pointer-variable;
```

The *size* specifies the number of elements in the array to be freed. The problem with this form is that the programmer should remember the size of the array. Recent versions of C++ do not require the size to be specified. For example,

```
delete [ ]p;
```

will delete the entire array pointed to by **p**.

What happens if sufficient memory is not available for allocation? In such cases, like **malloc()**, **new** returns a null pointer. Therefore, it may be a good idea to check for the pointer produced by **new** before using it. It is done as follows:

```
.....
.....
p = new int;
if(!p)
{
  cout << "allocation failed \n";
}
.....
.....
```

SRINIVAS COLLEGE OF
PG MANAGEMENT STUDIES
ACC No.:........1205....
CALL No.:..............................

The **new** operator offers the following advantages over the function **malloc()**.

1. It automatically computes the size of the data object. We need not use the operator **sizeof**.
2. It automatically returns the correct pointer type, so that there is no need to use a type cast.
3. It is possible to initialize the object while creating the memory space.
4. Like any other operator, **new** and **delete** can be overloaded.

3.17 Manipulators

Manipulators are operators that are used to format the data display. The most commonly used manipulators are **endl** and **setw**.

The **endl** manipulator, when used in an output statement, causes a linefeed to be inserted. It has the same effect as using the newline character "\n". For example, the statement

```
.....
.....
cout << "m = " << m << endl
     << "n = " << n << endl
     << "p = " << p << endl;
.....
.....
```

would cause three lines of output, one for each variable. If we assume the values of the variables as 2597, 14, and 175 respectively, the output will appear as follows:

```
m = | 2 | 5 | 9 | 7 |
n = | 1 | 4 |
p = | 1 | 7 | 5 |
```

It is important to note that this form is not the ideal output. It should rather appear as under:

```
m =  2597
n =    14
p =   175
```

Here, the numbers are *right-justified*. This form of output is possible only if we can specify a common field width for all the numbers and force them to be printed right-justified. The **setw** manipulator does this job. It is used as follows:

```
cout << setw(5) << sum << endl;
```

The manipulator **setw(5)** specifies a field width 5 for printing the value of the variable sum. This value is right-justified within the field as shown below:

```
|   |   | 3 | 4 | 5 |
```

Program 3.2 illustrates the use of **endl** and **setw**.

USE OF MANIPULATORS

```
#include <iostream>
#include <iomanip>    // for setw

using namespace std;

int main()
{
        int Basic = 950, Allowance = 95, Total = 1045;

        cout << setw(10) << "Basic" << setw(10) << Basic << endl
             << setw(10) << "Allowance" << setw(10) << Allowance << endl
             << setw(10) << "Total" << setw(10) << Total << endl;

        return 0;
}
```

PROGRAM 3.2

Output of this program is given below:

```
    Basic   950
Allowance    95
    Total  1045
```

note

Character strings are also printed right-justified.

We can also write our own manipulators as follows:

```
#include <iostream>
ostream & symbol(ostream & output)
{
   return output << "\tRs";
}
```

The **symbol** is the new manipulator which represents **Rs**.The identifier **symbol** can be used whenever we need to display the string **Rs**.

3.18 Type Cast Operator

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

Traditional C casts are augmented in C++ by a function-call notation as a syntactic alternative. 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
```

A type-name behaves as if it is a function for converting values to a designated type. The function-call notation usually leads to simplest expressions. However, it can be used only if the type is an identifier. For example,

```
p = int * (q);
```

is illegal. In such cases, we must use C type notation.

```
p = (int *) q;
```

Alternatively, we can use **typedef** to create an identifier of the required type and use it in the functional notation.

```
typedef int * int_pt;
p = int_pt(q);
```

ANSI C++ adds the following new cast operators:

- const_cast
- static_cast
- dynamic_cast
- reinterpret_cast

Application of these operators is discussed in Chapter 16.

3.19 Expressions and Their Types

An expression is a combination of operators, constants and variables arranged as per the rules of the language. It may also include function calls which return values. An expression may consist of one or more operands, and zero or more operators to produce a value. 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.*

Constant Expressions

Constant Expressions consist of only constant values. Examples:

```
15
20 + 5 / 2.0
'x'
```

Integral Expressions

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

```
m
m * n - 5
m * 'x'
5 + int(2.0)
```

where **m** and **n** are integer variables.

Float Expressions

Float Expressions are those which, after all conversions, produce floating-point results. Examples:

```
x + y
x * y / 10
5 + float(10)
10.75
```

where **x** and **y** are floating-point variables.

Pointer Expressions

Pointer Expressions produce address values. Examples:

```
&m
ptr
ptr + 1
"xyz"
```

where **m** is a variable and **ptr** is a pointer.

Relational Expressions

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

```
x <= y
a+b == c+d
m+n > 100
```

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*.

Logical Expressions

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

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

Bitwise Expressions

Bitwise Expressions are used to manipulate data at 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.

ANSI C++ has introduced what are termed as *operator keywords* that can be used as alternative representation for operator symbols. Operator keywords are given in Chapter 16.

3.20 Special Assignment Expressions

Chained Assignment

```
x = (y = 10);
    or
x = y = 10;
```

First 10 is assigned to y and then to x.

A chained statement cannot be used to initialize variables at the time of declaration. For instance, the statement

```
float a = b = 12.34;          // wrong
```

is illegal. This may be written as

```
float a=12.34, b=12.34        // correct
```

Embedded Assignment

```
x = (y = 50) + 10;
```

(y = 50) is an assignment expression known as embedded assignment. Here, the value 50 is assigned to y and then the result 50+10 = 60 is assigned to x. This statement is identical to

```
y = 50;
x = y + 10;
```

Compound Assignment

Like C, C++ supports a *compound assignment operator* which is a combination of the assignment operator with a binary arithmetic operator. For example, the simple assignment statement

```
x = x + 10;
```

may be written as

```
x += 10;
```

The operator += is known as *compound assignment operator* or *short-hand assignment operator*. The general form of the compound assignment operator is:

```
variable1 op= variable2;
```

where *op* is a binary arithmetic operator. This means that

```
variable1 = variable1 op variable2;
```

3.21 Implicit Conversions

We can mix data types in expressions. For example,

```
m = 5+2.75;
```

is a valid statement. Wherever data types are mixed in an expression, C++ performs the conversions automatically. This process is known as *implicit* or *automatic conversion.*

When the compiler encounters an expression, it divides the expressions into sub-expressions consisting of one operator and one or two operands. For a binary operator, if the operands type differ, the compiler converts one of them to match with the other, using the rule that the "smaller" type is converted to the "wider" type. For example, if one of the operand is an **int** and the other is a **float**, the **int** is converted into a **float** because a **float** is wider than an **int**. The "water-fall" model shown in **Fig. 3.3** illustrates this rule.

Table 3.5 *Operator precedence and associativity*

<table>
<tr><th>Operator</th><th>Associativity</th></tr>
<tr><td>::</td><td>left to right</td></tr>
<tr><td>–> . ( ) [ ] postfix ++ postfix – –</td><td>left to right</td></tr>
<tr><td>prefix ++ prefix – – ~ ! unary + unary –
unary * unary & (type) sizeof new delete</td><td>right to left</td></tr>
<tr><td>–> * *</td><td>left to right</td></tr>
<tr><td>* / %</td><td>left to right</td></tr>
<tr><td>+ –</td><td>left to right</td></tr>
<tr><td><< >></td><td>left to right</td></tr>
<tr><td><< = >> =</td><td>left to right</td></tr>
<tr><td>= = !=</td><td>left to right</td></tr>
<tr><td>&</td><td>left to right</td></tr>
<tr><td>^</td><td>left to right</td></tr>
<tr><td>|</td><td>left to right</td></tr>
<tr><td>&&</td><td>left to right</td></tr>
<tr><td>| |</td><td>left to right</td></tr>
<tr><td>?:</td><td>left to right</td></tr>
<tr><td>= * = / = % = + = =</td><td>right to left</td></tr>
<tr><td><< = >> = & = ^= |=</td><td>left to right</td></tr>
<tr><td>, (comma)</td><td></td></tr>
</table>

The unary operations assume higher precedence.

3.24 Control Structures

In C++, a large number of functions are used that pass messages, and process the data contained in objects. A function is set up to perform a task. When the task is complex, many different algorithms can be designed to achieve the same goal. Some are simple to comprehend, while others are not. Experience has also shown that the number of bugs that occur is related to the format of the program. The format should be such that it is easy to trace the flow of execution of statements. This would help not only in debugging but also in the review and maintenance of the program later. One method of achieving the objective of an accurate, error-resistant and maintainable code is to use one or any combination of the following three control structures:

1. Sequence structure (straight line)
2. Selection structure (branching)
3. Loop structure (iteration or repetition)

Figure 3.4 shows how these structures are implemented using *one-entry, one-exit* concept, a popular approach used in modular programming.

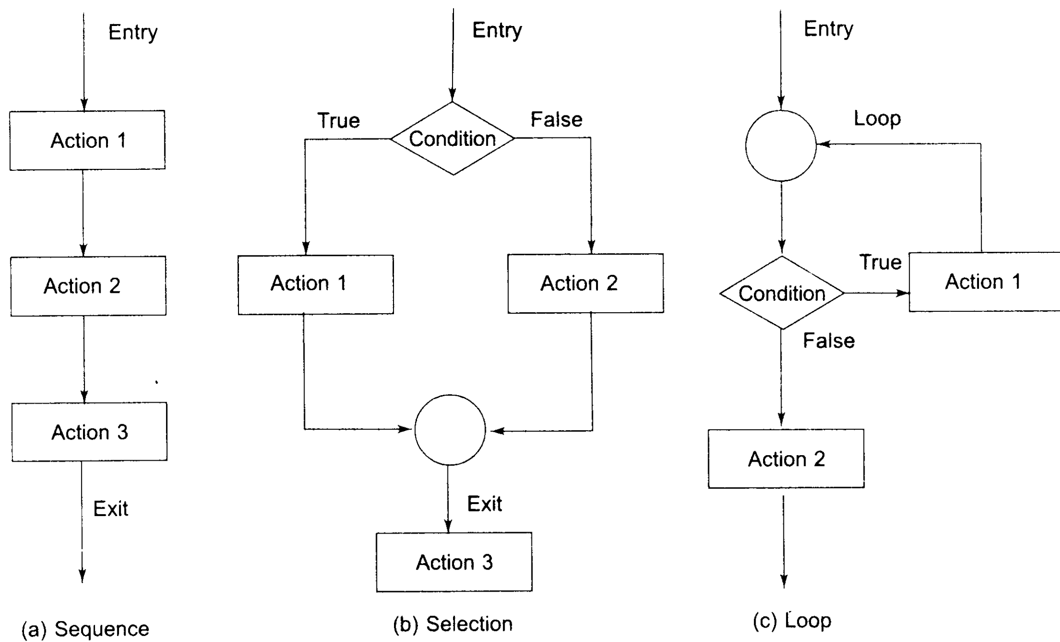

Fig. 3.4 ⇔ *Basic control structures*

It is important to understand that all program processing can be coded by using only these three logic structures. The approach of using one or more of these basic control constructs in programming is known as *structured programming*, an important technique in software engineering.

Using these three basic constructs, we may represent a function structure either in detail or in summary form as shown in Figs 3.5 (a), (b) and (c).

Like C, C++ also supports all the three basic control structures, and implements them using various control statements as shown in Fig. 3.6. This shows that C++ combines the power of structured programming with the object-oriented paradigm.

The if statement

The **if** statement is implemented in two forms:

- Simple **if** statement
- **if...else** statement

```
switch(expression)
{
  case1:
  {
  action1;
  }
  case2:
  {
  action2;
  }
  case3:
  {
  action3;
  }
  default:
  {
  action4;
  }
}
action5;
```

The do-while statement

The **do-while** is an *exit-controlled* loop. Based on a condition,the control is transferred back to a particular point in the program. The syntax is as follows:

```
do
{
  action1;
}
while(condition is true);
action2;
```

The while statement

This is also a loop structure, but is an *entry-controlled* one. The syntax is as follows:

```
while(condition is true)
{
  action1;
}
action2;
```

The for statement

The **for** is an *entry-entrolled* loop and is used when an action is to be repeated for a predetermined number of times. The syntax is as follows:

```
for(initial value; test; increment)
{
  action1;
}
action2;
```

The syntax of the control statements in C++ is very much similar to that of C and therefore they are implemented as and when they are required.

SUMMARY

⇔ C++ provides various types of tokens that include keywords, identifiers, constants, strings, and operators.

⇔ Identifiers refer to the names of variables, functions, arrays, classes, etc.

⇔ 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.

⇔ In C++, the size of character array should be one larger than the number of characters in the string.

⇔ C++ adds the concept of constant pointer and pointer to constant. In case of constant pointer we can not modify the address that the pointer is initialized to. In case of pointer to a constant, contents of what it points to cannot be changed.

⇔ Pointers are widely used in C++ for memory management and to achieve polymorphism.

⇔ C++ provides a qualifier called **const** to declare named constants which are just like variables except that their values can not be changed. A **const** modifier defaults to an **int**.

⇔ 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.

⇔ 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.

⇔ A reference variable provides an alternative name for a previously defined variable. Both the variables refer to the same data object in the memory. Hence, change in the value of one will also be reflected in the value of the other variable.

⇔ A reference variable must be initialized at the time of declaration, which establishes the correspondence between the reference and the data object that it names.

3.2 *An* ***unsigned int*** *can be twice as large as the* ***signed int****. Explain how?*

3.3 *Why does C++ have type modifiers?*

3.4 *What are the applications of* ***void*** *data type in C++?*

3.5 *Can we assign a* ***void*** *pointer to an* ***int*** *type pointer? If not, why? How can we achieve this?*

3.6 *Describe, with examples, the uses of enumeration data types.*

3.7 *Describe the differences in the implementation of* ***enum*** *data type in ANSI C and C++.*

3.8 *Why is an array called a derived data type?*

3.9 *The size of a* ***char*** *array that is declared to store a string should be one larger than the number of characters in the string. Why?*

3.10 *The* ***const*** *was taken from C++ and incorporated in ANSI C, although quite differently. Explain.*

3.11 *How does a constant defined by* ***const*** *differ from the constant defined by the preprocessor statement* ***#define****?*

3.12 *In C++, a variable can be declared anywhere in the scope. What is the significance of this feature?*

3.13 *What do you mean by dynamic initialization of a variable? Give an example.*

3.14 *What is a reference variable? What is its major use?*

3.15 *List at least four new operators added by C++ which aid OOP.*

3.16 *What is the application of the scope resolution operator :: in C++?*

3.17 *What are the advantages of using* ***new*** *operator as compared to the function* ***malloc()****?*

3.18 *Illustrate with an example, how the* ***setw*** *manipulator works.*

3.19 *How do the following statements differ?*

(a) *char * const p;*

(b) *char const *p;*

Debugging Exercises

3.1 What will happen when you execute the following code?

```
#include <iostream.h>
void main()
{
        int i=0;
        i=400*400/400;
        cout << i;
}
```

3.2 Identify the error in the following program.

```
#include <iostream.h>
void main()
```

```
{
        int num[]={1,2,3,4,5,6};
        num[1]==[1]num  ?    cout<<"Success"    :     cout<<"Error";
}
```

3.3 Identify the errors in the following program.

```
#include <iostream.h>
void main()
{
  int i=5;
  while(i)
  {
        switch(i)
        {
        default:
        case 4:
        case 5:

        break;

        case 1:
        continue;

        case 2:
        case 3:
        break;

        }
        i-;
  }
}
```

3.4 Identify the error in the following program.

```
#include <iostream.h>
#define pi 3.14
int squareArea(int &);
int circleArea(int &);

void main()
{
        int a=10;
        cout << squareArea(a) << " ";
```

Write a program to read the figures set out in the above form, to calculate the batting averages and to print out the complete table including the averages.

3.7 *Write programs to evaluate the following functions to 0.0001% accuracy.*

(a) $\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots\cdots$

(b) $\text{SUM} = 1 + (1/2)^2 + (1/3)^3 + (1/4)^4 + \cdots\cdots$

(c) $\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots\cdots$

3.8 *Write a program to print a table of values of the function*

$y = e^{-x}$

for x varying from 0 to 10 in steps of 0.1. The table should appear as follows.

TABLE FOR Y = EXP [-X]

X	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
0.0									
1.0									
.									
.									
.									
9.0									

3.9 *Write a program to calculate the variance and standard deviation of N numbers.*

$$\text{Variance} = \frac{1}{N}\sum_{i=1}^{N}(x_1 - \bar{x})^2$$

$$\text{Standard Deviation} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

where $\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_1$

3.10 *An electricity board charges the following rates to domestic users to discourage large consumption of energy:*

For the first 100 units	-	*60P per unit*
For next 200 units	-	*80P per unit*
Beyond 300 units	-	*90P per unit*

All users are charged a minimum of Rs. 50.00. If the total amount is more than Rs. 300.00 then an additional surcharge of 15% is added.

Write a program to read the names of users and number of units consumed and print out the charges with names.

4

Functions in C++

Key Concepts

- Return types in main()
- Function prototyping
- Call by reference
- Call by value
- Return by reference
- Inline functions
- Default arguments
- Constant arguments
- Function overloading

4.1 Introduction

We know that functions play an important role in C program development. Dividing a program into functions is one of the major principles of top-down, structured programming. Another advantage of using functions is that it is possible to reduce the size of a program by calling and using them at different places in the program.

Recall that we have used a syntax similar to the following in developing C programs.

```
void show();      /* Function declaration */
main()
{
     .....
     show();          /* Function call */
     .....
}
void show()       /* Function definition */
{
     .....
```

```
        .....          /* Function body */
        .....
}
```

When the function is called, control is transferred to the first statement in the function body. The other statements in the function body are then executed and control returns to the main program when the closing brace is encountered. C++ is no exception. Functions continue to be the building blocks of C++ programs. In fact, C++ has added many new features to functions to make them more reliable and flexible. Like C++ operators, a C++ function can be overloaded to make it perform different tasks depending on the arguments passed to it. Most of these modifications are aimed at meeting the requirements of object-oriented facilities.

In this chapter, we shall briefly discuss the various new features that are added to C++ functions and their implementation.

4.2 The Main Function

C does not specify any return type for the **main()** function which is the starting point for the execution of a program. The definition of **main()** would look like this:

```
main()
{
        // main program statements
}
```

This is perfectly valid because the **main()** in C does not return any value.

In C++, the **main()** returns a value of type **int** to the operating system. C++, therefore, explicitly defines **main()** as matching one of the following prototypes:

```
int main();
int main(int argc, char * argv[]);
```

The functions that have a return value should use the **return** statement for termination. The **main()** function in C++ is, therefore, defined as follows:

```
int main()
{
     ......
     ......
     return 0;
}
```

Since the return type of functions is **int** by default, the keyword **int** in the **main()** header is optional. Most C++ compilers will generate an error or warning if there is no **return**

statement. Turbo C++ issues the warning

```
Function should return a value
```

and then proceeds to compile the program. It is good programming practice to actually return a value from **main()**.

Many operating systems test the return value (called *exit value*) to determine if there is any problem. The normal convention is that an exit value of zero means the program ran successfully, while a nonzero value means there was a problem. The explicit use of a **return(0)** statement will indicate that the program was successfully executed.

4.3 Function Prototyping

Function *prototyping* is one of the major improvements added to C++ functions. The prototype describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return values. With function prototyping, a *template* is always used when declaring and defining a function. When a function is called, the compiler uses the template to ensure that proper arguments are passed, and the return value is treated correctly. Any violation in matching the arguments or the return types will be caught by the compiler at the time of compilation itself. These checks and controls did not exist in the conventional C functions.

Remember, C also uses prototyping. But it was introduced first in C++ by Stroustrup and the success of this feature inspired the ANSI C committee to adopt it. However, there is a major difference in prototyping between C and C++. While C++ makes the prototyping essential, ANSI C makes it optional, perhaps, to preserve the compatibility with classic C.

Function prototype is a *declaration statement* in the calling program and is of the following form:

```
type function-name (argument-list);
```

The *argument-list* contains the types and names of arguments that must be passed to the function.

Example:

```
float volume(int x, float y, float z);
```

Note that each argument variable must be declared independently inside the parentheses. That is, a combined declaration like

```
float volume(int x, float y, z);
```

is illegal.

In a function declaration, the names of the arguments are *dummy* variables and therefore, they are optional. That is, the form

```
float volume(int, float, float);
```

is acceptable at the place of declaration. At this stage, the compiler only checks for the type of arguments when the function is called.

In general, we can either include or exclude the variable names in the argument list of prototypes. The variable names in the prototype just act as placeholders and, therefore, if names are used, they don't have to match the names used in the *function call or function definition.*

In the function definition, names are required because the arguments must be referenced inside the function. Example:

```
float volume(int a,float b,float c)
{
        float v = a*b*c;
        .....
        .....
}
```

The function **volume()** can be invoked in a program as follows:

```
float cubel = volume(b1,w1,h1);  // Function call
```

The variable **b1, w1**, and **h1** are known as the actual parameters which specify the dimensions of **cube1**. Their types (which have been declared earlier) should match with the types declared in the prototype. Remember, the calling statement should not include type names in the argument list.

We can also declare a function with an *empty argument list*, as in the following example:

```
void display( );
```

In C++, this means that the function does not pass any parameters. It is identical to the statement

```
void display(void);
```

However, in C, an empty parentheses implies any number of arguments. That is, we have foregone prototyping. A C++ function can also have an 'open' parameter list by the use of ellipses in the prototype as shown below:

```
void do_something(...);
```

4.4 Call by Reference

In traditional C, a function call passes arguments by value. The *called function* creates a new set of variables and copies the values of arguments into them. The function does not have access to the actual variables in the calling program and can only work on the copies of values. This mechanism is fine if the function does not need to alter the values of the original variables in the calling program. But, there may arise situations where we would like to change the values of variables in the *calling program*. For example, in bubble sort, we compare two adjacent elements in the list and interchange their values if the first element is greater than the second. If a function is used for *bubble sort*, then it should be able to alter the values of variables in the calling function, which is not possible if the call-by-value method is used.

Provision of the *reference variables* in C++ permits us to pass parameters to the functions by reference. When we pass arguments by reference, the 'formal' arguments in the called function become aliases to the 'actual' arguments in the calling function. This means that when the function is working with its own arguments, it is actually working on the original data. Consider the following function:

```
void swap(int &a,int &b)        // a and b are reference variables
{
     int t = a;                 // Dynamic initialization
     a = b;
     b = t;
}
```

Now, if **m** and **n** are two integer variables, then the function call

```
swap(m, n);
```

will exchange the values of **m** and **n** using their aliases (reference variables) **a** and **b**. Reference variables have been discussed in detail in Chapter 3. In traditional C, this is accomplished using *pointers* and *indirection* as follows:

```
void swapl(int *a, int *b)  /* Function definition */
{
      int t;
      t = *a;       /* assign the value at address a to t */
      *a = *b;      /* put the value at b into a  */
      *b = t;       /* put the value at t into b  */
}
```

This function can be called as follows:

```
swapl(&x, &y);    /* call by passing */
                  /* addresses of variables */
```

This approach is also acceptable in C++. Note that the call-by-reference method is neaterin its approach.

4.5 Return by Reference

A function can also return a reference. Consider the following function:

```
int & max(int &x,int &y)
{
        if (x > y)
              return x;
        else
              return y;
}
```

Since the return type of **max()** is **int &**, the function returns reference to **x** or **y** (and not the values). Then a function call such as **max(a, b)** will yield a reference to either **a** or **b** depending on their values. This means that this function call can appear on the left-hand side of an assignment statement. That is, the statement

```
max(a,b) = -1;
```

is legal and assigns -1 to **a** if it is larger, otherwise -1 to **b**.

4.6 Inline Functions

One of the objectives of using functions in a program is to save some memory space, which becomes appreciable when a function is likely to be called many times. However, every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack, and returning to the calling function. When a function is small, a substantial percentage of execution time may be spent in such overheads.

One solution to this problem is to use macro definitions, popularly known as *macros*. Preprocessor macros are popular in C. The major drawback with macros is that they are not really functions and therefore, the usual error checking does not occur during compilation.

C++ has a different solution to this problem. To eliminate the cost of calls to small functions, C++ proposes a new feature called *inline function*. An inline function is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the

corresponding function code (something similar to macros expansion). The inline functions are defined as follows:

```
inline function-header
{
   function body
}
```

Example:

```
inline double cube(double a)
{
   return(a*a*a);
}
```

The above inline function can be invoked by statements like

```
c = cube(3.0);
d = cube(2.5+1.5);
```

On the execution of these statements, the values of c and d will be 27 and 64 respectively. If the arguments are expressions such as 2.5 + 1.5, the function passes the value of the expression, 4 in this case. This makes the inline feature far superior to macros.

It is easy to make a function inline. All we need to do is to prefix the keyword **inline** to the function definition. All inline functions must be defined before they are called.

We should exercise care before making a function **inline**. The speed benefits of **inline** functions diminish as the function grows in size. At some point the overhead of the function call becomes small compared to the execution of the function, and the benefits of **inline** functions may be lost. In such cases, the use of normal functions will be more meaningful. Usually, the functions are made inline when they are small enough to be defined in one or two lines. Example:

```
inline double cube(double a) {return(a*a*a);}
```

Remember that the inline keyword merely sends a request, not a command, to the compiler. The compiler may ignore this request if the function definition is too long or too complicated and compile the function as a normal function.

Some of the situations where inline expansion may not work are:

1. For functions returning values, if a loop, a **switch**, or a **goto** exists.
2. For functions not returning values, if a return statement exists.
3. If functions contain **static** variables.
4. If **inline** functions are recursive.

note

Inline expansion makes a program run faster because the overhead of a function call and return is eliminated. However, it makes the program to take up more memory because the statements that define the inline function are reproduced at each point where the function is called. So, a trade-off becomes necessary.

Program 4.1 illustrates the use of inline functions.

INLINE FUNCTIONS

```
#include <iostream>

using namespace std;

        inline float mul(float x, float y)
        {
                return(x*y);
        }

        inline double div(double p, double q)
        {
                return(p/q);
        }

    int main()
    {
        float a = 12.345;
        float b = 9.82;

        cout << mul(a,b) << "\n";
        cout << div(a,b) << "\n";

        return 0;
    }
```

PROGRAM 4.1

The output of program 4.1 would be

```
121.228
1.25713
```

4.7 Default Arguments

C++ allows us to call a function without specifying all its arguments. In such cases, the function assigns a *default value* to the parameter which does not have a matching argument

in the function call. Default values are specified when the function is declared. The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values. Here is an example of a prototype (i.e. function declaration) with default values:

```
float amount(float principal,int period,float rate=0.15);
```

The default value is specified in a manner syntactically similar to a variable initialization. The above prototype declares a default value of 0.15 to the argument **rate**. A subsequent function call like

```
value = amount(5000,7);        // one argument missing
```

passes the value of 5000 to **principal** and 7 to **period** and then lets the function use default value of 0.15 for rate. The call

```
value = amount(5000,5,0.12);        // no missing argument
```

passes an explicit value of 0.12 to **rate**.

A default argument is checked for type at the time of declaration and evaluated at the time of call. One important point to note is that only the trailing arguments can have default values and therefore we must add defaults from *right to left*. We cannot provide a default value to a particular argument in the middle of an argument list. Some examples of function declaration with default values are:

```
int mul(int i, int j=5, int k=10);        // legal
int mul(int i=5, int j);                  // illegal
int mul(int i=0, int j, int k=10);        // illegal
int mul(int i=2, int j=5, int k=10);      // legal
```

Default arguments are useful in situations where some arguments always have the same value. For instance, bank interest may remain the same for all customers for a particular period of deposit. It also provides a greater flexibility to the programmers. A function can be written with more parameters than are required for its most common application. Using default arguments, a programmer can use only those arguments that are meaningful to a particular situation. Program 4.2 illustrates the use of default arguments.

DEFAULT ARGUMENTS

```
#include <iostream>

using namespace std;
```

(Contd)

```
int main()
{
        float amount;

        float value(float p, int n, float r=0.15);  // prototype
        void printline(char ch='*', int len=40);    // prototype

        printline();          // uses default values for arguments

        amount = value(5000.00,5);          // default for 3rd argument

        cout << "\n       Final Value = " << amount << "\n\n";

        printline('=');       // use default value for 2nd argument

        return 0;
}
/*----------------------------------------------------*/
float value(float p, int n, float r)
{
        int year = 1;
        float sum = p;

        while(year <= n)
        {
               sum = sum*(1+r);
               year = year+1;
        }
        return(sum);
}

void printline(char ch, int len)
{
        for(int i=1; i<=len; i++) printf("%c",ch);
        printf("\n");
}
```

PROGRAM 4.2

The output of Program 4.2 would be

```
****************************************
      Final Value = 10056.8
========================================
```

Advantages of providing the default arguments are:

1. We can use default arguments to add new parameters to the existing functions.
2. Default arguments can be used to combine similar functions into one.

4.8 const Arguments

In C++, an argument to a function can be declared as const as shown below.

```
int strlen(const char *p);
int length(const string &s);
```

The qualifier **const** tells the compiler that the function should not modify the argument. The compiler will generate an error when this condition is violated. This type of declaration is significant only when we pass arguments by reference or pointers.

4.9 Function Overloading

As stated earlier, *overloading* refers to the use of the same thing for different purposes. C++ also permits overloading of functions. This means that we can use the same function name to create functions that perform a variety of different tasks. This is known as *function polymorphism* in OOP.

Using the concept of function overloading; we can design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type. For example, an overloaded **add()** function handles different types of data as shown below:

```
// Declarations
int add(int a, int b);                    // prototype 1
int add(int a, int b, int c);             // prototype 2
double add(double x, double y);           // prototype 3
double add(int p, double q);              // prototype 4
double add(double p, int q);              // prototype 5

// Function calls
cout << add(5, 10);                   // uses prototype 1
cout << add(15, 10.0);                // uses prototype 4
cout << add(12.5, 7.5);               // uses prototype 3
cout << add(5, 10, 15);               // uses prototype 2
cout << add(0.75, 5);                 // uses prototype 5
```

A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution. A best match must be unique. The function selection involves the following steps:

1. The compiler first tries to find an exact match in which the types of actual arguments are the same, and use that function.
2. If an exact match is not found, the compiler uses the integral promotions to the actual arguments, such as,

   char to **int**
   float to **double**

   to find a match.
3. When either of them fails, the compiler tries to use the built-in conversions (the implicit assignment conversions) to the actual arguments and then uses the function whose match is unique. If the conversion is possible to have multiple matches, then the compiler will generate an error message. Suppose we use the following two functions:

   ```
   long square(long n)
   double square(double x)
   ```

   A function call such as

   ```
   square(10)
   ```

   will cause an error because **int** argument can be converted to either **long** or **double**, thereby creating an ambiguous situation as to which version of **square()** should be used.
4. If all of the steps fail, then the compiler will try the user-defined conversions in combination with integral promotions and built-in conversions to find a unique match. User-defined conversions are often used in handling class objects.

Program 4.3 illustrates function overloading.

FUNCTION OVERLOADING

```
// Function volume() is overloaded three times

#include <iostream>

using namespace std;

// Declarations (prototypes)
int volume(int);
double volume(double, int);
long volume(long, int, int);
```

(Contd)

```
int main()
{
        cout << volume(10) << "\n";
        cout << volume(2.5,8) << "\n";
        cout << volume(100L,75,15) << "\n";

        return 0;
}
// Function definitions

int volume(int s)  // cube
{
        return(s*s*s);
}

double volume(double r, int h)   // cylinder
{
        return(3.14519*r*r*h);
}

long volume(long l, int b, int h) // rectangular box
{
        return(l*b*h);
}
```

PROGRAM 4.3

The output of Program 4.3 would be:

```
1000
157.26
112500
```

Overloading of the functions should be done with caution. We should not overload unrelated functions and should reserve function overloading for functions that perform closely related tasks. Sometimes, the default arguments may be used instead of overloading. This may reduce the number of functions to be defined.

Overloaded functions are extensively used for handling class objects. They will be illustrated later when the classes are discussed in the next chapter.

4.10 Friend and Virtual Functions

C++ introduces two new types of functions, namely, friend function and virtual function. They are basically introduced to handle some specific tasks related to class objects. Therefore, discussions on these functions have been reserved until after the class objects are discussed. The friend functions are discussed in Sec. 5.15 of the next chapter and virtual functions in Sec. 9.5 of Chapter 9.

4.11 Math Library Functions

The standard C++ supports many math functions that can be used for performing certain commonly used calculations. Most frequently used math library functions are summarized in Table 4.1.

Table 4.1 *Commonly used math library functions*

Function	***Purposes***
ceil(x)	Rounds x to the smallest integer not less than x ceil(8.1) = 9.0 and ceil(-8.8) = –8.0
cos(x)	Trigonometric cosine of x (x in radians)
exp(x)	Exponential function e^x.
fabs(x)	Absolute value of x. If x>0 then abs(x) is x If x=0 then abs(x) is 0.0 If x<0 then abs(x) is –x
floor(x)	Rounds x to the largest integer not greater than x floor(8.2) = 8.0 and floor(–8.8 = –9.0
log(x)	Natural logarithm of x(base e)
log10(x)	Logarithm of x(base 10)
pow(x,y)	x raised to power y(x^y)
sin(x)	Trigonometric sine of x (x in radians)
sqrt(x)	Square root of x
tan(x)	Trigonometric tangent of x (x in radians)

note

The argument variables **x** and **y** are of type **double** and all the functions return the data type **double**.

To use the math library functions, we must include the header file **math.h** in conventional C++ and **cmath** in ANSI C++.

SUMMARY

⇔ It is possible to reduce the size of program by calling and using functions at different places in the program.

⇔ In C++ the main() returns a value of type **int** to the operating system. Since the return type of functions is **int** by default, the keyword **int** in the main() header is optional. Most C++ compilers issue a warning, if there is no return statement.

⇔ Function prototyping gives the compiler the details about the functions such as the number and types of arguments and the type of return values.

⇔ Reference variables in C++ permit us to pass parameters to the functions by reference. A function can also return a reference to a variable.

⇔ When a function is declared inline the compiler replaces the function call with the respective function code. Normally, a small size function is made as **inline**.

⇔ The compiler may ignore the inline declaration if the function declaration is too long or too complicated and hence compile the function as a normal function.

⇔ C++ allows us to assign default values to the function parameters when the function is declared. In such a case we can call a function without specifying all its arguments. The defaults are always added from right to left.

⇔ In C++, an argument to a function can be declared as **const**, indicating that the function should not modify the argument.

⇔ C++ allows function overloading. That is, we can have more than one function with the same name in our program. The compiler matches the function call with the exact function code by checking the number and type of the arguments.

⇔ C++ supports two new types of functions, namely **friend** functions and **virtual** functions.

⇔ Many mathematical computations can be carried out using the library functions supported by the C++ standard library.

Key Terms

- actual arguments
- argument list
- bubble sort
- call by reference
- call by value
- called function
- calling program
- calling statement
- **cmath**
- **const** arguments
- declaration statement
- default arguments
- default values
- dummy variables
- ellipses
- empty argument list
- exit value
- formal arguments
- **friend** functions
- function call
- function definition
- function overloading
- function polymorphism
- function prototype
- indirection
- **inline**

(Contd)

- inline functions
- macros
- **main()**
- math library
- **math.h**
- overloading
- pointers
- polymorphism
- prototyping
- reference variable
- return by reference
- **return** statement
- return type
- **return()**
- template
- virtual functions

Review Questions

4.1 *State whether the following statements are TRUE or FALSE.*
   (a) *A function argument is a value returned by the function to the calling program.*
   (b) *When arguments are passed by value, the function works with the original arguments in the calling program.*
   (c) *When a function returns a value, the entire function call can be assigned to a variable.*
   (d) *A function can return a value by reference.*
   (e) *When an argument is passed by reference, a temporary variable is created in the calling program to hold the argument value.*
   (f) *It is not necessary to specify the variable name in the function prototype.*

4.2 *What are the advantages of function prototypes in C++?*

4.3 *Describe the different styles of writing prototypes.*

4.4 *Find errors, if any, in the following function prototypes.*
   (a) *float average(x,y);*
   (b) *int mul(int a,b);*
   (c) *int display(...);*
   (d) *void Vect(int? &V, int & size);*
   (e) *void print(float data [], size = 20);*

4.5 *What is the main advantage of passing arguments by reference?*

4.6 *When will you make a function **inline**? Why?*

4.7 *How does an **inline** function differ from a preprocessor macro?*

4.8 *When do we need to use default arguments in a function?*

4.9 *What is the significance of an empty parenthesis in a function declaration?*

4.10 *What do you meant by overloading of a function? When do we use this concept?*

4.11 *Comment on the following function definitions:*

(a)
```
int *f( )
{
        int m = 1;
        ....
        ....
        return(&m);
}
```

(b)
```
double f( )
{
            ....
            ....
            return(1);
}
```

(c)
```
int & f()
{
        int n = 10;
        ....
        ....
        return(n);
}
```

Debugging Exercises

4.1 Identify the error in the following program.

```
#include <iostream.h>
int fun()
{
        return 1;
}
float fun()
{
        return 10.23;
void main()
{
            cout << (int)fun() << ' ';
            cout << (float)fun() << ' ';
}
```

4.2 Identify the error in the following program.

```
#include <iostream.h>

void display(const int const1=5)
{
        const int const2=5;
        int array1[const1];
        int array2[const2];
        for(int i=0; i<5; i++)
        {
             array1[i] = i;
             array2[i] = i*10;
             cout << array1[i] << ' ' << array2[i] << ' ' ;
        }
}

void main()
{
        display(5);
}
```

4.3 Identify the error in the following program.

```
#include <iostream.h>
int gValue=10;
void extra()
{
        cout << gValue << '  ';
}
void main()
{
        extra();
        {
             int gValue = 20;
             cout << gValue << ' ';
             cout << : gValue << ' ';
        }
}
```

4.4 Find errors, if any, in the following function definition for displaying a matrix:
void display(int A[ ] [ ], int m, int n)

```
{
     for(i=0; i<m; i++)
```

```
            for(j=0; j<n; j++)
                  cout << " " << A[i][j];
            cout << "\n";
}
```

Programming Exercises

4.1 *Write a function to read a matrix of size m x n from the keyboard.*

4.2 *Write a program to read a matrix of size m x n from the keyboard and display the same on the screen using functions.*

4.3 *Rewrite the program of Exercise 4.2 to make the row parameter of the matrix as a default argument.*

4.4 *The effect of a default argument can be alternatively achieved by overloading. Discuss with an example.*

4.5 *Write a macro that obtains the largest of three numbers.*

4.6 *Redo Exercise 4.5 using inline function. Test the function using a* **main** *program.*

4.7 *Write a function* **power()** *to raise a number* **m** *to a power* **n**. *The function takes a* **double** *value for* **m** *and* **int** *value for* **n**, *and returns the result correctly. Use a default value of 2 for n to make the function to calculate squares when this argument is omitted. Write a* **main** *that gets the values of* **m** *and* **n** *from the user to test the function.*

4.8 *Write a function that performs the same operation as that of Exercise 4.7 but takes an* **int** *value for* **m**. *Both the functions should have the same name. Write a* **main** *that calls both the functions. Use the concept of function overloading.*

5

Classes and Objects

Key Concepts

- Using structures
- Creating a class
- Defining member functions
- Creating objects
- Using objects
- Inline member functions
- Nested member functions
- Private member functions
- Arrays as class members
- Storage of objects
- Static data members
- Static member functions
- Using arrays of objects
- Passing objects as parameters
- Making functions friendly to classes
- Functions returning objects
- **const** member functions
- Pointers to members
- Using dereferencing operators
- Local classes

5.1 Introduction

The most important feature of C++ is the "class". Its significance is highlighted by the fact that Stroustrup initially gave the name "C with classes" to his new language. A class is an

extension of the idea of structure used in C. It is a new way of creating and implementing a user-defined data type. We shall discuss, in this chapter, the concept of class by first reviewing the traditional structures found in C and then the ways in which classes can be designed, implemented and applied.

5.2 C Structures Revisited

We know that one of the unique features of the C language is structures. They provide a method for packing together data of different types. A structure is a convenient tool for handling a group of logically related data items. It is a user-defined data type with a *template* that serves to define its data properties. Once the structure type has been defined, we can create variables of that type using declarations that are similar to the built-in type declarations. For example, consider the following declaration:

```
struct student
{
     char   name[20];
     int    roll_number;
     float total_marks;
};
```

The keyword **struct** declares **student** as a new data type that can hold three fields of different data types. These fields are known as *structure members* or *elements*. The identifier student, which is referred to as *structure name* or *structure tag*, can be used to create variables of type student. Example:

```
struct student A;   // C declaration
```

A is a variable of type student and has three member variables as defined by the template. Member variables can be accessed using the *dot* or *period operator* as follows:

```
strcpy(A.name, "John");
A.roll_number = 999;
A.total_marks = 595.5;
Final_total = A.total_marks + 5;
```

Structures can have arrays, pointers or structures as members.

Limitations of C Structure

The standard C does not allow the struct data type to be treated like built-in types. For example, consider the following structure:

```
struct complex
{
      float x;
      float y;
};
      struct complex c1, c2, c3;
```

The complex numbers c1, c2, and c3 can easily be assigned values using the dot operator, but we cannot add two complex numbers or subtract one from the other. For example,

```
c3 = c1 + c2;
```

is illegal in C.

Another important limitation of C structures is that they do not permit *data hiding*. Structure members can be directly accessed by the structure variables by any function anywhere in their scope. In other words, the structure members are public members.

Extensions to Structures

C++ supports all the features of structures as defined in C. But C++ has expanded its capabilities further to suit its OOP philosophy. It attempts to bring the user-defined types as close as possible to the built-in data types, and also provides a facility to hide the data which is one of the main principles of OOP. *Inheritance*, a mechanism by which one type can inherit characteristics from other types, is also supported by C++.

In C++, a structure can have both variables and functions as members. It can also declare some of its members as 'private' so that they cannot be accessed directly by the external functions.

In C++, the structure names are stand-alone and can be used like any other type names. In other words, the keyword struct can be omitted in the declaration of structure variables. For example, we can declare the student variable A as

```
student A;    // C++ declaration
```

Remember, this is an error in C.

C++ incorporates all these extensions in another user-defined type known as **class**. There is very little syntactical difference between structures and classes in C++ and, therefore, they can be used interchangeably with minor modifications. Since class is a specially introduced data type in C++, most of the C++ programmers tend to use the structures for holding only data, and classes to hold both the data and functions. Therefore, we will not discuss structures any further.

note

The only difference between a structure and a class in C++ is that, by default, the members of a class are *private*, while, by default, the members of a structure are *public*.

5.3 Specifying a Class

A class is a way to bind the data and its associated functions together. It allows the data (and functions) to be hidden, if necessary, from external use. When defining a class, we are creating a new *abstract data type* that can be treated like any other built-in data type. Generally, a class specification has two parts:

1. Class declaration
2. Class function definitions

The class declaration describes the type and scope of its members. The class function definitions describe how the class functions are implemented.

The general form of a class declaration is:

```
class class_name
{
        private:
            variable declarations;
            function declarations;
        public:
            variable declarations;
            function declaration;
};
```

The **class** declaration is similar to a **struct** declaration. The keyword **class** specifies, that what follows is an abstract data of type *class_name*. The body of a class is enclosed within braces and terminated by a semicolon. The class body contains the declaration of variables and functions. These functions and variables are collectively called *class members*. They are usually grouped under two sections, namely, *private* and *public* to denote which of the members are *private* and which of them are *public*. The keywords **private** and **public** are known as visibility labels. Note that these keywords are followed by a colon.

The class members that have been declared as private can be accessed only from within the class. On the other hand, public members can be accessed from outside the class also. The data hiding (using private declaration) is the key feature of object-oriented programming. The use of the keyword private is optional. By default, the members of a class are **private**. If both the labels are missing, then, by default, all the members are **private**. Such a class is completely hidden from the outside world and does not serve any purpose.

The variables declared inside the class are known as *data members* and the functions are known as *member functions*. Only the member functions can have acçess to the private data members and private functions. However, the public members (both functions and data) can be accessed from outside the class. This is illustrated in Fig. 5.1. The binding of data and functions together into a single class-type variable is referred to as *encapsulation*.

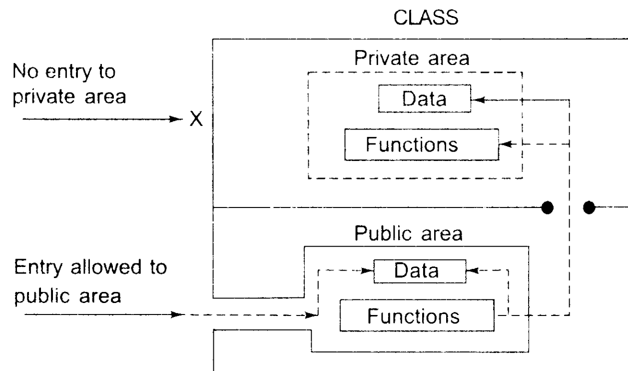

Fig. 5.1 ⇔ *Data hiding in classes*

A Simple Class Example

A typical class declaration would look like:

```
class item
{
     int number;                    // variables declaration
     float cost;                    // private by default
  public:
     void getdata(int a, float b);  // functions declaration
     void putdata(void);            // using prototype
  };// ends with semicolon
```

We usually give a class some meaningful name, such as **item**. This name now becomes a new type identifier that can be used to declare *instances* of that class type. The class item contains two data members and two function members. The data members are private by default while both the functions are public by declaration. The function **getdata()** can be used to assign values to the member variables number and cost, and **putdata()** for displaying their values. These functions provide the only access to the data members from outside the class. This means that the data cannot be accessed by any function that is not a member of the class **item**. Note that the functions are declared, not defined. Actual function definitions will appear later in the program. The data members are usually declared as **private** and the member functions as **public**. Figure 5.2 shows two different notations used by the OOP analysts to represent a class.

Creating Objects

Remember that the declaration of **item** as shown above does not define any objects of **item** but only specifies *what* they will contain. Once a class has been declared, we can create variables of that type by using the class name (like any other built-in type variable). For example,