The McGraw-Hill Companies

FOURTH EDITION

OBJECT ORIENTED PROGRAMMING WITH C++

E BALAGURUSAMY

1.5 Basic Concepts of Object-Oriented Programming

It is necessary to understand some of the concepts used extensively in object-oriented programming. These include:

- Objects
- Classes

- Data abstraction and encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing

We shall discuss these concepts in some detail in this section.

Objects

Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represent user-defined data such as vectors, time and lists. Programming problem is analyzed in terms of objects and the nature of communication between them. Program objects should be chosen such that they match closely with the real-world objects. Objects take up space in the memory and have an associated address like a record in Pascal, or a structure in C.

When a program is executed, the objects interact by sending messages to one another. For example, if "customer"and "account" are two objects in a program, then the customer object may send a message to the account object requesting for the bank balance. Each object contains data, and code to manipulate the data. Objects can interact without having to know details of each other's data or code. It is sufficient to know the type of message accepted, and the type of response returned by the objects. Although different authors represent them differently, Fig. 1.7 shows two notations that are popularly used in object-oriented analysis and design.

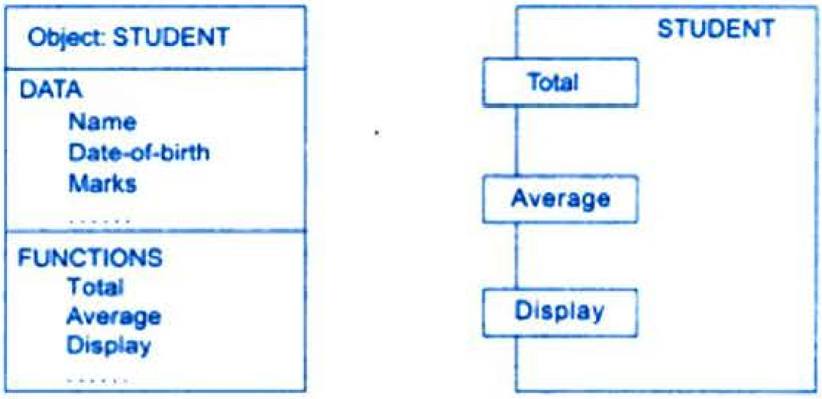


Fig. 1.7 ⇔ *Two ways of representing an object*

Classes

We just mentioned that objects contain data, and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of a *class*. In fact, objects are variables of the type *class*. Once a class has been defined, we can create any number of objects belonging to that class. Each object is associated with the data of type class with which they are created. A class is thus a collection of objects of similar type. For example, mango, apple and orange are members of the class fruit. Classes are user-defined data types and behave like the built-in types of a programming language. The syntax used to create an object is no different than the syntax used to create an integer object in C. If fruit has been defined as a class, then the statement

```
fruit mango;
```

will create an object **mango** belonging to the class **fruit**.

Data Abstraction and Encapsulation

The wrapping up of data and functions into a single unit (called class) is known as *encapsulation*. Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called *data hiding or information hiding*.

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract *attributes* such as size, weight and cost, and *functions* to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created. The attributes are sometimes called *data members* because they hold information. The functions that operate on these data are sometimes called *methods or member functions*.

Since the classes use the concept of data abstraction, they are known as *Abstract Data Types* (ADT).

Inheritance

Inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of *hierarchical classification*. For example, the bird 'robin' is a part of the class 'flying bird' which is again a part of the class 'bird'. The principle behind this sort of division is that each derived class shares common characteristics with the class from which it is derived as illustrated in Fig. 1.8.

In OOP, the concept of inheritance provides the idea of *reusability*. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes. The real appeal and power of the inheritance mechanism is that it allows the programmer to reuse a class that is almost, but not exactly, what he wants, and to tailor the class in such a way that it does not introduce any undesirable side-effects into the rest of the classes.

Note that each sub-class defines only those features that are unique to it. Without the use of classification, each class would have to explicitly include all of its features.

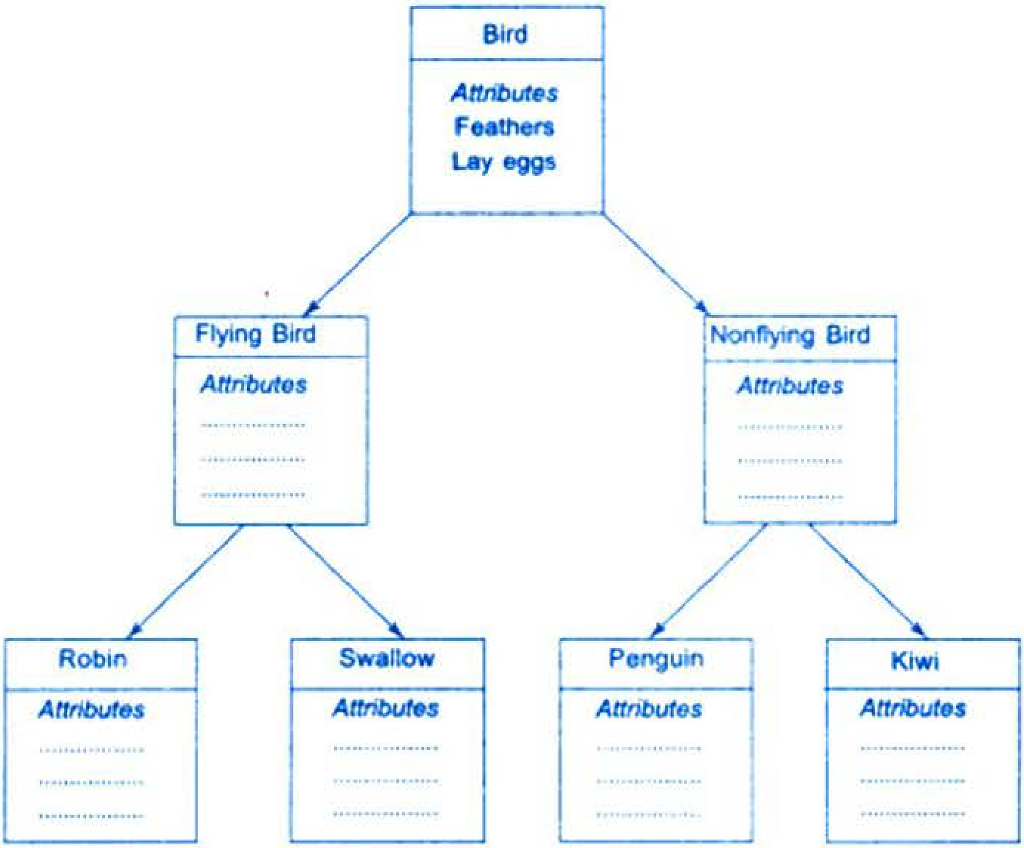


Fig. 1.8 ⇔ *Property inheritance*

Polymorphism

Polymorphism is another important OOP concept. Polymorphism, a Greek term, means the ability to take more than one form. An operation may exhibit different behaviours in different instances. The behaviour depends upon the types of data used in the operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation. The process of making an operator to exhibit different behaviours in different instances is known as *operator overloading*.

Figure 1.9 illustrates that a single function name can be used to handle different number and different types of arguments. This is something similar to a particular word having several different meanings depending on the context. Using a single function name to perform different types of tasks is known as *function overloading*.

Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operation may differ. Polymorphism is extensively used in implementing inheritance.

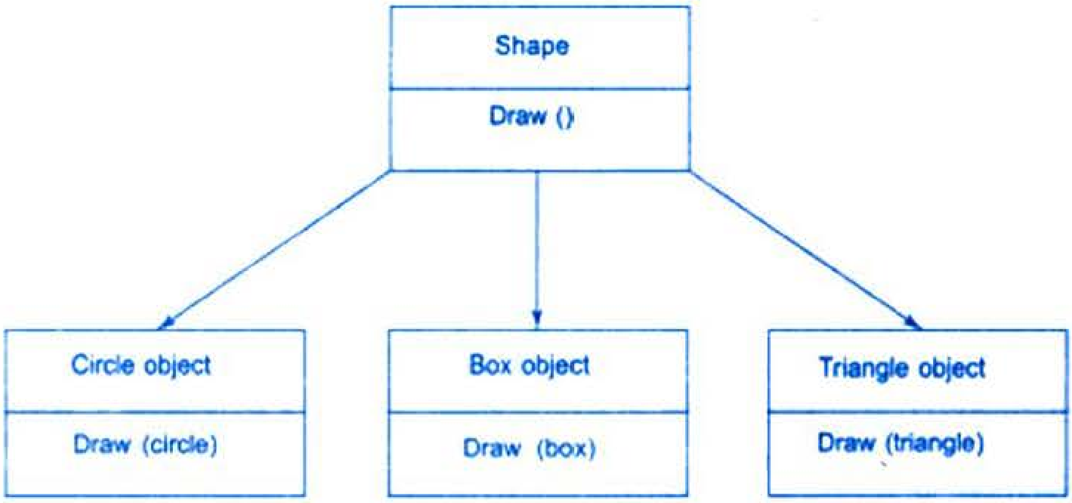


Fig. 1.9 ⇔ Polymorphism

Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. *Dynamic binding* (also known as late binding) means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

Consider the procedure "draw" in Fig. 1.9. By inheritance, every object will have this procedure. Its algorithm is, however, unique to each object and so the draw procedure will be redefined in each class that defines the object. At run-time, the code matching the object under current reference will be called.

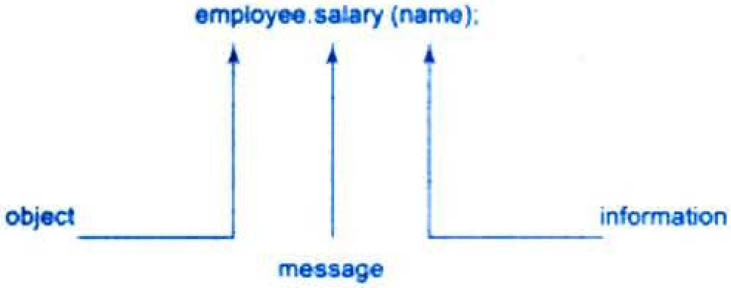
Message Passing

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, therefore, involves the following basic steps:

1. Creating classes that define objects and their behaviour,
2. Creating objects from class definitions, and
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.

A message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired result. *Message passing* involves specifying the name of the object, the name of the function (message) and the information to be sent. Example:



Objects have a life cycle. They can be created and destroyed. Communication with an object is feasible as long as it is alive.

1.6 Benefits of OOP

OOP offers several benefits to both the program designer and the user. Object-orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost. The principal advantages are:

- Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map objects in the problem domain to those in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more details of a model in implementable form.
- Object-oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

While it is possible to incorporate all these features in an object-oriented system, their importance depends on the type of the project and the preference of the programmer. There are a number of issues that need to be tackled to reap some of the benefits stated above. For

1.8 Applications of OOP

OOP has become one of the programming buzzwords today. There appears to be a great deal of excitement and interest among software engineers in using OOP. Applications of OOP are beginning to gain importance in many areas. The most popular application of object-oriented programming, up to now, has been in the area of user interface design such as windows. Hundreds of windowing systems have been developed, using the OOP techniques.

Real-business systems are often much more complex and contain many more objects with complicated attributes and methods. OOP is useful in these types of applications because it can simplify a complex problem. The promising areas for application of OOP include:

- Real-time systems
- Simulation and modeling
- Object-oriented databases
- Hypertext, hypermedia and expertext
- AI and expert systems
- Neural networks and parallel programming
- Decision support and office automation systems
- CIM/CAM/CAD systems

The richness of OOP environment has enabled the software industry to improve not only the quality of software systems but also its productivity. Object-oriented technology is certainly changing the way the software engineers think, analyze, design and implement systems.

3.2 Tokens

As we know, the smallest individual units in a program are known as tokens. C++ has the following tokens:

- Keywords
- Identifiers
- Constants
- Strings
- Operators

A C++ program is written using these tokens, white spaces, and the syntax of the language. Most of the C++ tokens are basically similar to the C tokens with the exception of some additions and minor modifications.

3.3 Keywords

The keywords implement specific C++ language features. They are explicitly reserved identifiers and cannot be used as names for the program variables or other user-defined program elements.

Table 3.1 gives the complete set of C++ keywords. Many of them are common to both C and C++. The ANSI C keywords are shown in boldface. Additional keywords have been added to the ANSI C keywords in order to enhance its features and make it an object-oriented language. ANSI C++ standards committee has added some more keywords to make the language more versatile. These are shown separately. Meaning and purpose of all C++ keywords are given in Appendix D.

3.4 Identifiers and Constants

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

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

note

C++ supports two types of string representation — the C-style character string and the string class type introduced with Standard C++. Although the use of the string class type is recommended, it is advisable to understand and use C-style strings in some situations. The string class type strings support many features and are discussed in detail in Chapter 15.

3.5 Basic Data Types

Data types in C++ can be classified under various categories as shown in Fig. 3.1.

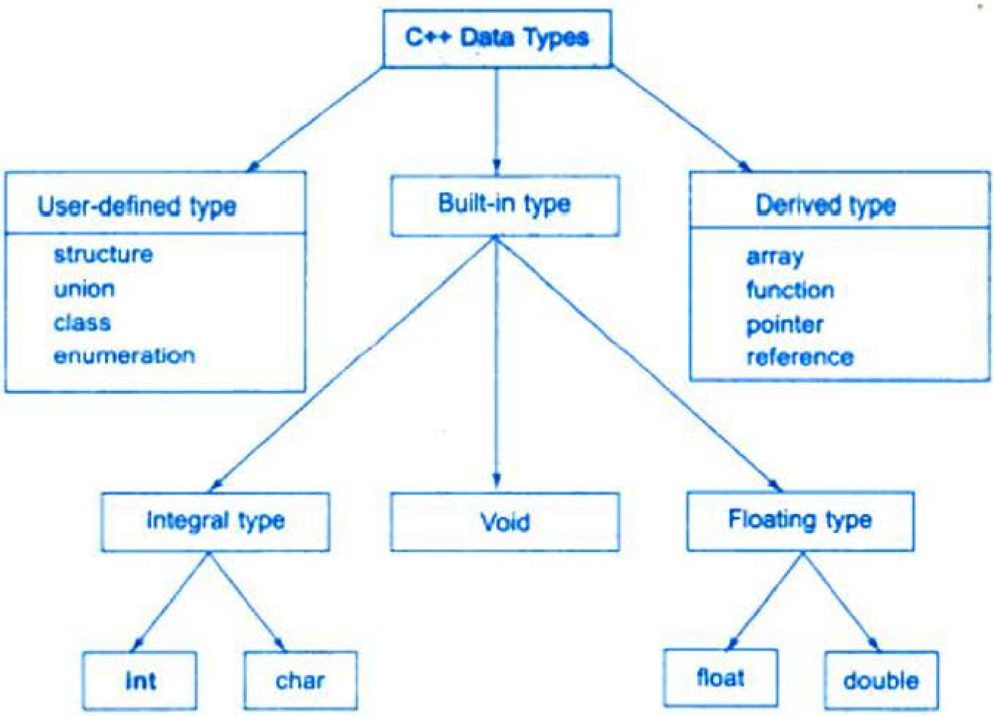


Fig. 3.1 ⇔ *Hierarchy of C++ data types*

Both C and C++ compilers support all the built-in (also known as *basic* or *fundamental*) data types. With the exception of **void**, the basic data types may have several *modifiers* preceding them to serve the needs of various situations. The modifiers **signed, unsigned, long**, and **short** may be applied to character and integer basic data types. However, the modifier **long** may also be applied to **double**. Data type representation is machine specific in C++. Table 3.2 lists all combinations of the basic data types and modifiers along with their size and range for a 16-bit word machine.

Table 3.2 *Size and range of C++ basic data types*

Type	*Bytes*	*Range*
char	1	–128 to 127
unsigned char	1	0 to 255
signed char	1	– 128 to 127
int	2	– 32768 to 32767
unsigned int	2	0 to 65535
signed int	2	– 31768 to 32767
short int	2	– 31768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	–32768 to 32767
long int	4	–2147483648 to 2147483647
signed long int	4	–2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
float	4	3.4E–38 to 3.4E+38
double	8	1.7E–308 to 1.7E+308
long double	10	3.4E–4932 to 1.1E+4932

ANSI C++ committee has added two more data types, **bool** and **wchar_t**. They are discussed in Chapter 16.

The type **void** was introduced in ANSI C. Two normal uses of **void** are (1) to specify the return type of a function when it is not returning any value, and (2) to indicate an empty argument list to a function. Example:

```
void functl(void);
```

Another interesting use of **void** is in the declaration of generic pointers. Example:

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

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

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

are valid statements. But, the statement,

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

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

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

```
void *ptrl;
char *ptr2;
ptr2 = ptrl;
```

are all valid statements in ANSI C but not in C++. A **void** pointer cannot be directly assigned to other type pointers in C++. We need to use a cast operator as shown below:

```
ptr2 = (char *)ptrl;
```

3.6 User-Defined Data Types

Structures and Classes

We have used user-defined data types such as **struct** and **union** in C. While these data types are legal in C++, some more features have been added to make them suitable for object-oriented programming. C++ also permits us to define another user-defined data type known as **class** which can be used, just like any other basic data type, to declare variables. The class variables are known as objects, which are the central focus of object-oriented programming. More about these data types is discussed later in Chapter 5.

Enumerated Data Type

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

```
enum shape{circle, square, triangle};
enum colour{red, blue, green, yellow};
enum position{off, on};
```

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

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

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

```
colour background = blue;           // allowed
colour background = 7;              // Error in C++
colour background = (colour) 7;     // OK
```

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

```
int c = red;      // valid, colour type promoted to int
```

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

```
enum colour{red, blue=4, green=8};
enum colour{red=5, blue, green};
```

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

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

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

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

```
int switch_1 = off;
int switch_2 = on;
```

In practice, enumeration is used to define symbolic constants for a **switch** statement. Example:

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

int main()
{
  cout << "Enter shape code:";
  int code;
  cin >> code;
  while(code >= circle && code <= triangle)
  {
        switch(code)
```

```
        {
                case circle:
                ......
                ......
                break;
                case rectangle:
                ......
                ......
                break;
                case triangle:
                ......
                ......
                break;
        }
        cout << "Enter shape code:";
        cin >> code;
    }
    cout << "BYE \n";

    return 0;
}
```

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

3.7 Derived Data Types

Arrays

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

```
char string[3] = "xyz";
```

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

```
char string[4] = "xyz"; // O.K. for C++
```

Functions

Functions have undergone major changes in C++. While some of these changes are simple, others require a new way of thinking when organizing our programs. Many of these modifications and improvements were driven by the requirements of the object-oriented concept of C++. Some of these were introduced to make the C++ program more reliable and readable. All the features of C++ functions are discussed in Chapter 4.

Pointers

Pointers are declared and initialized as in C. Examples:

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

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

```
char * const ptr1 = "GOOD";    // constant pointer
```

We cannot modify the address that **ptr1** is initialized to.

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

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

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

```
const char * const cp = "xyz";
```

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

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

3.13 Operators in C++

C++ has a rich set of operators. All C operators are valid in C++ also. In addition, C++ introduces some new operators. We have already seen two such operators, namely, the insertion operator <<, and the extraction operator >>. Other new operators are:

`::`	Scope resolution operator
`::*`	Pointer-to-member declarator
`->*`	Pointer-to-member operator
`.*`	Pointer-to-member operator
`delete`	Memory release operator
`endl`	Line feed operator
`new`	Memory allocation operator
`setw`	Field width operator

In addition, C++ also allows us to provide new definitions to some of the built-in operators. That is, we can give several meanings to an operator, depending upon the types of arguments used. This process is known as *operator overloading*.

3.14 Scope Resolution Operator

Like C, C++ is also a block-structured language. Blocks and scopes can be used in constructing programs. We know that the same variable name can be used to have different meanings in different blocks. The scope of the variable extends from the point of its declaration till the end of the block containing the declaration. A variable declared inside a block is said to be local to that block. Consider the following segment of a program:

```
.....
.....
{
   int x = 10;
   .....
   .....
}
.....
.....
{
   int x = 1;
   .....
   .....
}
```

The two declarations of x refer to two different memory locations containing different values. Statements in the second block cannot refer to the variable x declared in the first block, and vice versa. Blocks in C++ are often nested. For example, the following style is common:

```
. . . .
. . . .
{
  int x = 10;
  . . . .
  . . . .
  {
        int x = 1;
        . . . .          Block 2     Block 1
        . . . .
  }
  . . . .
}
```

Block2 is contained in block1. Note that a declaration in an inner block *hides* a declaration of the same variable in an outer block and, therefore, each declaration of **x** causes it to refer to 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

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";
}
.....
.....
```

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:

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

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.

3.24 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.

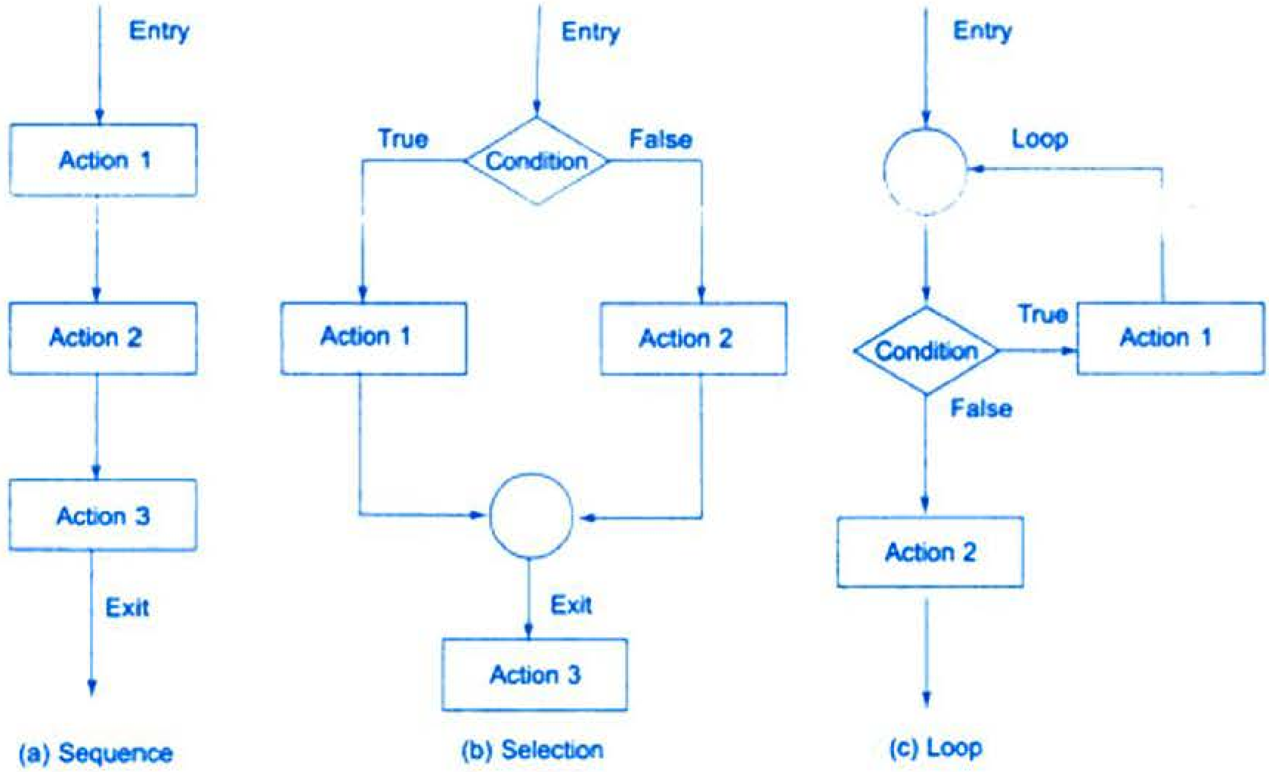


Fig. 3.4 ⇔ *Basic control structures*

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

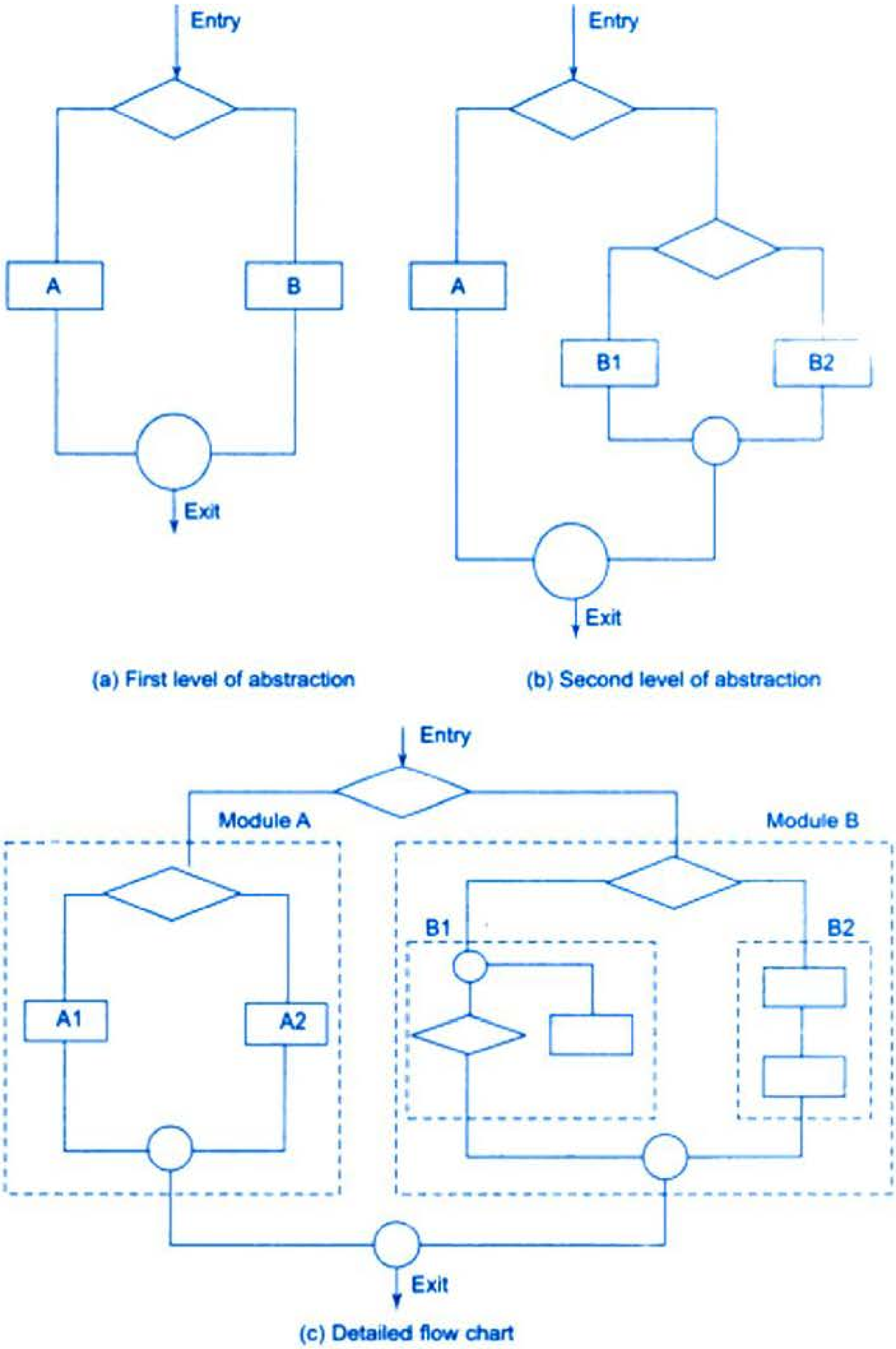


Fig. 3.5 ⇔ *Different levels of abstraction*

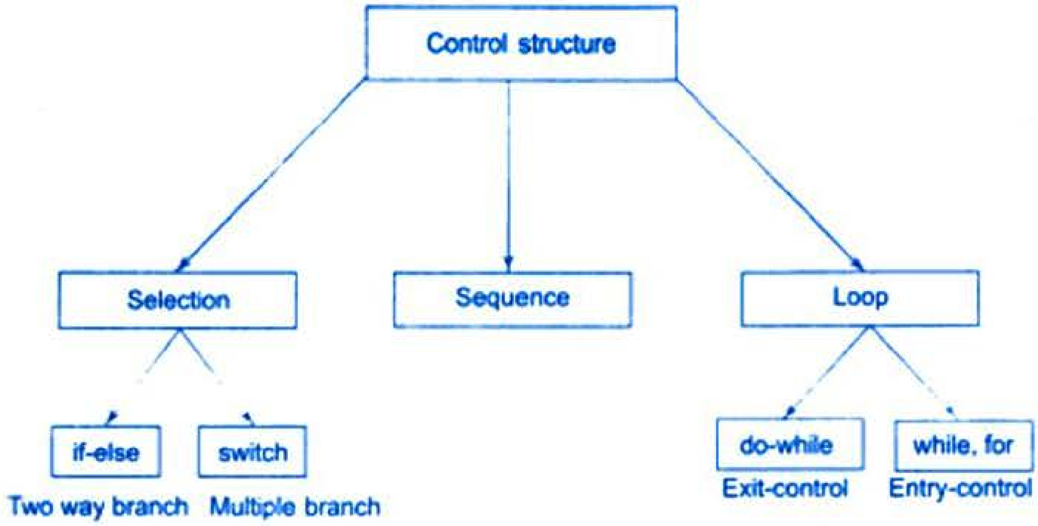


Fig. 3.6 ⇔ C++ *statements to implement in two forms*

Examples:

Form 1

```
if(expression is true)
{
   action1;
}
action2;
action3;
```

Form 2

```
if(expression is true)
{
   action1;
}
else
{
   action2;
}
action3;
```

The switch statement

This is a multiple-branching statement where, based on a condition, the control is transferred to one of the many possible points. This is implemented as follows:

```
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.

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:

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++.

5.11 Static Data Members

A data member of a class can be qualified as static. The properties of a **static** member variable are similar to that of a C static variable. A static member variable has certain special characteristics. These are :

- It is initialized to zero when the first object of its class is created. No other initialization is permitted.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its lifetime is the entire program.

Static variables are normally used to maintain values common to the entire class. For example, a static data member can be used as a counter that records the occurrences of all the objects. Program 5.4 illustrates the use of a static data member.

STATIC CLASS MEMBER

```
#include <iostream>

using namespace std;

class item
{
        static int count;
        int number;
  public:
        void getdata(int a)
        {
                number = a;
                count ++;
        }
        void getcount(void)
        {
                cout << "count: ";
                cout << count  << "\n";
        }
};

int item :: count;

int main()
{
```

(Contd)

```
        item a, b, c;           // count is initialized to zero
        a.getcount();           // display count
        b.getcount();
        c.getcount();

        a.getdata(100);         // getting data into object a
        b.getdata(200);         // getting data into object b
        c.getdata(300);         // getting data into object c

        cout << "After reading data" << "\n";

        a.getcount();           // display count
        b.getcount();
        c.getcount();
        return 0;
}
```

PROGRAM 5.4

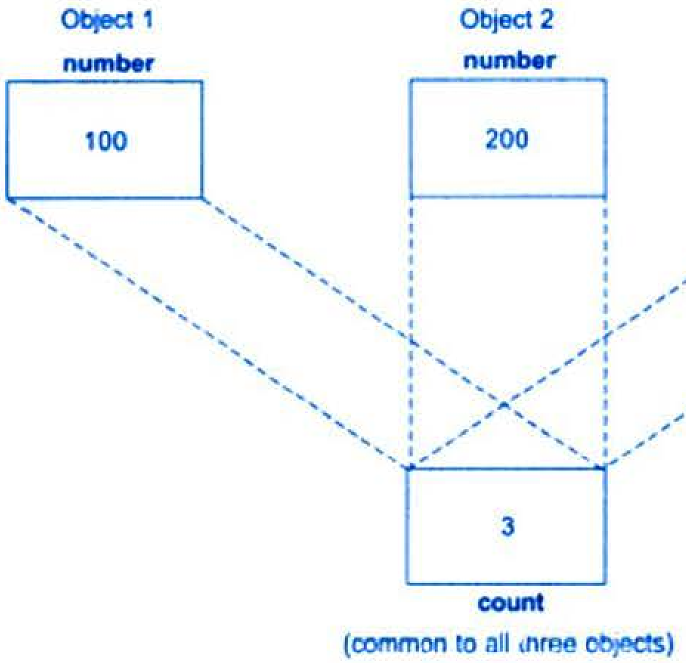


Fig. 5.4 ⇔ *Sharing of a static data member*

Static variables are like non-inline member functions as they are declared in a class declaration and defined in the source file. While defining a static variable, some initial value can also be assigned to the variable. For instance, the following definition gives count the initial value 10.

```
int item :: count = 10;
```

5.12 Static Member Functions

Like **static** member variable, we can also have **static** member functions. A member function that is declared **static** has the following properties:

- A **static** function can have access to only other static members (functions or variables) declared in the same class.
- A **static** member function can be called using the class name (instead of its objects) as follows:

```
class-name :: function-name;
```

Program 5.5 illustrates the implementation of these characteristics. The **static** function **showcount()** displays the number of objects created till that moment. A count of number of objects created is maintained by the static variable count.

The function **showcode()** displays the code number of each object.

STATIC MEMBER FUNCTION

```
#include <iostream>

using namespace std;

class test
{
        int code;
        static int count;               // static member variable
public:
        void setcode(void)
        {
        code = ++count;
        }
        void showcode(void)
        {
        cout << "object number: " << code << "\n";
        }
        static void showcount(void)     // static member function
        {
        cout << "count: " << count << "\n";
        }
};

int test :: count;
int main()
{
        test t1, t2;

        t1.setcode();
        t2.setcode();

        test :: showcount();    // accessing static function

        test t3;
        t3.setcode();

        test :: showcount();

        t1.showcode();
        t2.showcode();
        t3.showcode();

        return 0;
}
```

PROGRAM 5.5

5.14 Objects as Function Arguments

Like any other data type, an object may be used as a function argument. This can be done in two ways:

- A copy of the entire object is passed to the function.
- Only the address of the object is transferred to the function.

The first method is called *pass-by-value*. Since a copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call the function. The second method is called *pass-by-reference*. When an address of the object is passed, the called function works directly on the actual object used in the call. This means that any changes made to the object inside the function will reflect in the actual object. The pass-by reference method is more efficient since it requires to pass only the address of the object and not the entire object.

Program 5.7 illustrates the use of objects as function arguments. It performs the addition of time in the hour and minutes format.

OBJECTS AS ARGUMENTS

```
#include <iostream>

using namespace std;

class time
{
        int  hours;
        int  minutes;
  public:
        void gettime(int h, int m)
        { hours = h; minutes = m; }
        void puttime(void)
        {
              cout << hours << " hours and ";
              cout << minutes << " minutes " << "\n";
        }
        void sum(time, time);  // declaration with objects as arguments
};
void time :: sum(time t1, time t2)        // t1, t2 are objects
{
        minutes = t1.minutes + t2.minutes;
        hours = minutes/60;
        minutes = minutes%60;
        hours = hours + t1.hours + t2.hours;
}
int main()
{
        time T1, T2, T3;

        T1.gettime(2,45);    // get T1
        T2.gettime(3,30);    // get T2

        T3.sum(T1,T2);// T3=T1+T2

        cout << "T1 = "; T1.puttime();      // display T1
        cout << "T2 = "; T2.puttime();      // display T2
        cout << "T3 = "; T3.puttime();      // display T3

        return 0;
}
```

PROGRAM 5.7

The output of Program 5.7 would be:

```
T1 = 2 hours and 45 minutes
T2 = 3 hours and 30 minutes
T3 = 6 hours and 15 minutes
```

note

Since the member function **sum( )** is invoked by the object **T3**, with the objects **T1** and **T2** as arguments, it can directly access the hours and minutes variables of **T3**. But, the members of **T1** and **T2** can be accessed only by using the dot operator (like **T1.hours** and **T1.minutes**). Therefore, inside the function sum(), the variables **hours** and **minutes** refer to **T3**, **T1.hours** and **T1.minutes** refer to **T1**, and **T2.hours** and **T2.minutes** refer to **T2**.

Figure 5.6 illustrates how the members are accessed inside the function **sum()**.

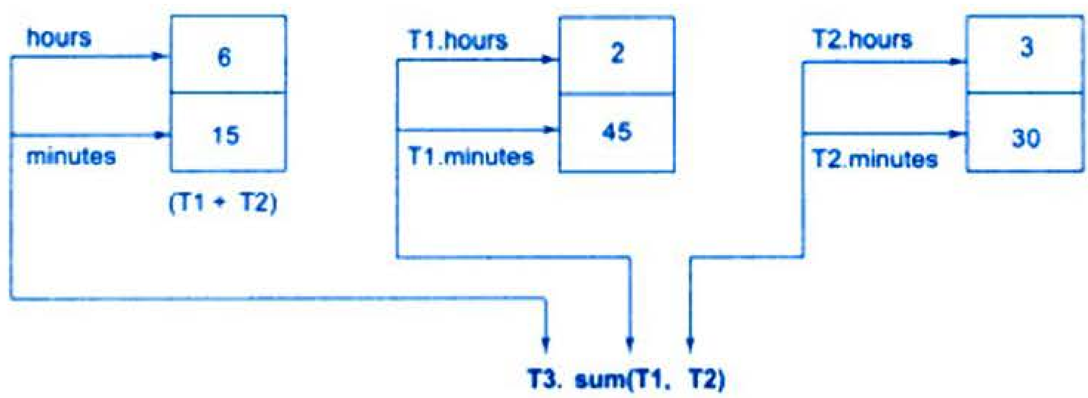


Fig. 5.6 ⇔ *Accessing members of objects within a called function*

An object can also be passed as an argument to a non-member function. However, such functions can have access to the **public member** functions only through the objects passed as arguments to it. These functions cannot have access to the private data members.

5.15 Friendly Functions

We have been emphasizing throughout this chapter that the private members cannot be accessed from outside the class. That is, a non-member function cannot have an access to the private data of a class. However, there could be a situation where we would like two classes to share a particular function. For example, consider a case where two classes, **manager** and **scientist**, have been defined. We would like to use a function **income_tax()** to operate on the objects of both these classes. In such situations, C++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of these classes. Such a function need not be a member of any of these classes.

6.2 Constructors

A constructor is a 'special' member function whose task is to initialize the objects of its class. It is special because its name is the same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.

A constructor is declared and defined as follows:

```
// class with a constructor

class integer
{
        int m, n;
    public:
        integer(void);           // constructor declared
        .....
        .....
};
integer :: integer(void)         // constructor defined
{
      m = 0; n = 0;
}
```

The constructor functions have some special characteristics. These are :

- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return types, not even void and therefore, and they cannot return values.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other C++ functions, they can have default arguments.
- Constructors cannot be **virtual**. (Meaning of virtual will be discussed later in Chapter 9.)
- We cannot refer to their addresses.
- An object with a constructor (or destructor) cannot be used as a member of a union.
- They make 'implicit calls' to the operators **new** and **delete** when memory allocation is required.

Remember, when a constructor is declared for a class, initialization of the class objects becomes mandatory.

6.3 Parameterized Constructors

The constructor **integer()**, defined above, initializes the data members of all the objects to zero. However, in practice it may be necessary to initialize the various data elements of different objects with different values when they are created. C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created. The constructors that can take arguments are called *parameterized constructors*.

The constructor **integer()** may be modified to take arguments as shown below:

```
class integer
{
      int m, n;
  public:
      integer(int x, int y);   // parameterized constructor
      .....
      .....
};
integer :: integer(int x, int y)
{
      m = x; n = y;
}
```

When a constructor has been parameterized, the object declaration statement such as

```
integer int1;
```

may not work. We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways:

- By calling the constructor explicitly.
- By calling the constructor implicitly.

The following declaration illustrates the first method:

```
integer int1 = integer(0,100); // explicit call
```

This statement creates an integer object int1 and passes the values 0 and 100 to it. The second is implemented as follows:

```
integer int1(0,100);                // implicit call
```

This method, sometimes called the shorthand method, is used very often as it is shorter, looks better and is easy to implement.

Remember, when the constructor is parameterized, we must provide appropriate arguments for the constructor. Program 6.1 demonstrates the passing of arguments to the constructor functions.

CLASS WITH CONSTRUCTORS

```
#include <iostream>

using namespace std;

class integer
{
        int m, n;
    public:
        integer(int, int);              // constructor declared

        void display(void)
        {
             cout << " m = " << m  << "\n";
             cout << " n = " << n  << "\n";
        }
};

integer :: integer(int x, int y)        // constructor defined
{
        m = x;  n = y;
}

int main()
{
      integer int1(0,100);              // constructor called implicitly

      integer int2 = integer(25, 75);   // constructor called explicitly

      cout << "\nOBJECT1" << "\n";
      int1.display();

      cout << "\nOBJECT2" << "\n";
      int2.display();

      return 0;
}
```

PROGRAM 6.1

Program 6.1 displays the following output:

```
OBJECT1
m = 0
n = 100
```

```
OBJECT2
m = 25
n = 75
```

The constructor functions can also be defined as **inline functions. Example:**

```
class integer
{
     int m, n;
  public:
     integer(int x, int y)   // Inline constructor
     {
            m = x; y = n;
     }
     .....
     .....
};
```

The parameters of a constructor can be of any type except that of the class to which it belongs. For example,

```
class A
{
      .....
      .....
    public:
      A(A);
};
```

is illegal.

However, a constructor can accept a *reference* to its own class as a parameter. Thus, the statement

```
Class A
{
     .....
     .....
  public:
     A(A&);
};
```

is valid. In such cases, the constructor is called the *copy constructor*.

```
Deposit 2
Principal Amount = 10000
Return Value     = 16430.3

Deposit 3
Principal Amount = 10000
Return Value     = 14049.3
```

The program uses three overloaded constructors. The parameter values to these constructors are provided at run time. The user can provide input in one of the following forms:

1. Amount, period and interest in decimal form.
2. Amount, period and interest in percent form.
3. Amount and period.

note

Since the constructors are overloaded with the appropriate parameters, the one that matches the input values is invoked. For example, the second constructor is invoked for the forms (1) and (3), and the third is invoked for the form (2). Note that, for form (3), the constructor with default argument is used. Since input to the third parameter is missing, it uses the default value for **r**.

6.7 Copy Constructor

We briefly mentioned about the copy constructor in Sec. 6.3. We used the copy constructor

```
integer(integer &i);
```

in Sec. 6.4 as one of the overloaded constructors.

As stated earlier, a copy constructor is used to declare and initialize an object from another object. For example, the statement

```
integer I2(I1);
```

would define the object I2 and at the same time initialize it to the values of I1. Another form of this statement is

```
integer I2 = I1;
```

The process of initializing through a copy constructor is known as *copy initialization*. Remember, the statement

```
I2 = I1;
```

will not invoke the copy constructor. However, if **I1** and **I2** are objects, this stateme
legal and simply assigns the values of **I1** to **I2**, member-by-member. This is the task o
overloaded assignment operator(=). We shall see more about this later.

A copy constructor takes a reference to an object of the same class as itself as an argum
Let us consider a simple example of constructing and using a copy constructor as show
Program 6.4.

COPY CONSTRUCTOR

```
#include <iostream>

using namespace std;

class code
{
        int id;
    public:
        code(){ }                       // constructor
        code(int a) { id = a;}          // constructor again
        code(code & x)                  // copy constructor

        {
                id = x.id;              // copy in the value
        }
        void display(void)
        {
                cout << id;
        }
};

int main()
{
        code A(100);    // object A is created and initialized
        code B(A);      // copy constructor called
        code C = A;     // copy constructor called again

        code D; // D is created, not initialized
        D = A;          // copy constructor not called

        cout << "\n id of A: "; A.display();
        cout << "\n id of B: "; B.display();
        cout << "\n id of C: "; C.display();
        cout << "\n id of D: "; D.display();

        return 0;
}
```

PROGRAM 6.

6.11 Destructors

A *destructor*, as the name implies, is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde. For example, the destructor for the class integer can be defined as shown below:

```
~integer(){ }
```

A destructor never takes any argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program (or block or function as the case may be) to clean up storage that is no longer accessible. It is a good practice to declare destructors in a program since it releases memory space for future use.

Whenever **new** is used to allocate memory in the constructors, we should use **delete** to free that memory. For example, the destructor for the **matrix** class discussed above may be defined as follows:

```
matrix :: ~matrix()
{
        for(int i=0; i<d1; i++)
        delete p[i];
        delete p;
}
```

This is required because when the pointers to objects go out of scope, a destructor is not called implicitly.

'he example below illustrates that the destructor has been invoked implicitly by ıpiler.

MPLEMENTATION OF DESTRUCTORS

```
#include <iostream>

using namespace std;

int count = 0;

class alpha
{
  public:
     alpha()
     {
          count++;
          cout << "\nNo.of object created " << count;
     }

     ~alpha()
     {
          cout << "\nNo.of object destroyed " << count;
          count--;
     }
};

int main()
{
     cout << "\n\nENTER MAIN\n";

     alpha A1, A2, A3, A4;
     {
          cout << "\n\nENTER BLOCK1\n";
          alpha A5;
     }

     {
          cout << "\n\nENTER BLOCK2\n";
          alpha  A6;
     }
     cout << "\n\nRE-ENTER MAIN\n";

     return 0;
}
```

PROGRAM 6.

```
vector operator+(vector);                    // vector addition
vector operator-();                          // unary minus
friend vector operator+(vector,vector);      // vector addition
friend vector operator-(vector);             // unary minus
vector operator-(vector &a);                 // subtraction
int operator==(vector);                      // comparison
friend int operator==(vector,vector)         // comparison
```

vector is a data type of **class** and may represent both magnitude and direction (as in physics and engineering) or a series of points called elements (as in mathematics)

The process of overloading involves the following steps:

1. Create a class that defines the data type that is to be used in the overloading operaion.
2. Declare the operator function **operator** *op*() in the public part of the class. It may be either a member function or a **friend** function.
3. Define the operator function to implement the required operations.

Overloaded operator functions can be invoked by expressions such as

```
op x or x op
```

for unary operators and

```
x op y
```

for binary operators. *op* x (or x *op*) would be interpreted as

```
operator op (x)
```

for **friend** functions. Similarly, the expression x op y would be interpreted as either

```
x.operator op (y)
```

in case of member functions, or

```
operator op (x,y)
```

in case of **friend** functions. When both the forms are declared, standard argument matching is applied to resolve any ambiguity.

7.3 Overloading Unary Operators

Let us consider the unary minus operator. A minus operator when used as a unary, takes just one operand. We know that this operator changes the sign of an operand when applied to a basic data item. We will see here how to overload this operator so that it can be applied to an object in much the same way as is applied to an **int** or **float** variable. The unary minus when applied to an object should change the sign of each of its data items.

Program 7.1 shows how the unary minus operator is overloaded.

OVERLOADING UNARY MINUS

```
#include <iostream>

using namespace std;

class space
{
        int x;
        int y;
        int z;
  public:
        void getdata(int a, int b, int c);
        void display(void);
        void operator-();      // overload unary minus
 };
 void space :: getdata(int a, int b, int c)
 {
        x = a;
        y = b;
        z = c;
 }
 void space :: display(void)
 {
        cout << x << " ";
        cout << y << " " ;
        cout << z << "\n";
 }
 void space :: operator-()
 {
        x = -x;
        y = -y;
        z = -z;
 }

 int main()
 {
        space S;
        S.getdata(10, -20, 30);
```

(Contd)

```
        cout << "S : ";
        S.display();

        -S;                    // activates operator-() function

        cout << "S : ";
        S.display();

        return 0;
}
```

PROGRAM 7.1

The Program 7.1 produces the following output:

```
S : 10 -20 30
S : -10 20 -30
```

note

The function **operator** – ( ) takes no argument. Then, what does this operator function do?. It changes the sign of data members of the object **S**. Since this function is a member function of the same class, it can directly access the members of the object which activated it.

Remember, a statement like

```
S2 = -S1;
```

will not work because, the function **operator–()** does not return any value. It can work if the function is modified to return an object.

It is possible to overload a unary minus operator using a friend function as follows:

```
friend  void operator-(space &s);          // declaration
void operator-(space &s)                   // definition
  {
        s.x = -s.x;
        s.y = -s.y;
        s.z = -s.z;
  }
```

note

Note that the argument is passed by reference. It will not work if we pass argument by value because only a copy of the object that activated the call is passed to operator-(). Therefore, the changes made inside the operator function will not reflect in the called object.

7.4 Overloading Binary Operators

We have just seen how to overload an unary operator. The same mechanism can be used to overload a binary operator. In Chapter 6, we illustrated, how to add two complex numbers using a friend function. A statement like

```
C = sum(A, B);          // functional notation.
```

was used. The functional notation can be replaced by a natural looking expression

```
C = A + B;              // arithmetic notation
```

by overloading the + operator using an operator+() function. The Program7.2 illustrates how this is accomplished.

OVERLOADING + OPERATOR

```
#include <iostream>

using namespace std;

class complex
{
        float x;                              // real part
        float y;                              // imaginary part
  public:
        complex(){ }                          // constructor 1
        complex(float real, float imag)       // constructor 2
        { x = real; y = imag; }
        complex operator+(complex);
        void display(void);
};

complex complex :: operator+(complex c)
{
        complex temp;                  // temporary
        temp.x = x + c.x;              // these are
        temp.y = y + c.y;              // float additions
        return(temp);
}

void complex :: display(void)
{
        cout << x << " + j" << y << "\n";
```

(Contd)

```
}

int main()
{
        complex C1, C2, C3;          // invokes constructor 1
        C1 = complex(2.5, 3.5);      // invokes constructor 2
        C2 = complex(1.6, 2.7);
        C3 = C1 + C2;

        cout << "C1 = "; C1.display();
        cout << "C2 = "; C2.display();
        cout << "C3 = "; C3.display();

        return 0;
}
```

PROGRAM 7.2

The output of Program 7.2 would be:

```
C1 = 2.5 + j3.5
C2 = 1.6 + j2.7
C3 = 4.1 + j6.2
```

note

Let us have a close look at the function **operator+( )** and see how the operator overloading is implemented.

```
complex complex :: operator+(complex c)
{
      complex temp;
      temp.x = x + c.x;
      temp.y = y + c.y;
      return(temp);
}
```

We should note the following features of this function:

1. It receives only one **complex** type argument explicitly.
2. It returns a **complex** type value.
3. It is a member function of **complex**.

The function is expected to add two complex values and return a complex value as the result but receives only one value as argument. Where does the other value come from? Now let us look at the statement that invokes this function:

```
C3 = C1 + C2;                    // invokes operator+() function
```

We can avoid the creation of the **temp** object by replacing the entire function body by the following statement:

```
return complex((x+c.x),(y+c.y));        // invokes constructor 2
```

What does it mean when we use a class name with an argument list? When the compiler comes across a statement like this, it invokes an appropriate constructor, initializes an object with no name and returns the contents for copying into an object. Such an object is called a temporary object and goes out of space as soon as the contents are assigned to another object. Using *temporary objects* can make the code shorter, more efficient and better to read.

7.5 Overloading Binary Operators Using Friends

As stated earlier, **friend** functions may be used in the place of member functions for overloading a binary operator, the only difference being that a **friend** function requires two arguments to be explicitly passed to it, while a member function requires only one.

The complex number program discussed in the previous section can be modified using a **friend** operator function as follows:

1. Replace the member function declaration by the **friend** function declaration.
   ```
   friend complex operator+(complex, complex);
   ```
2. Redefine the operator function as follows:
   ```
   complex operator+(complex a, complex b)
   {
         return complex((a.x+b.x),(a.y+b.y));
   }
   ```

In this case, the statement

```
C3 = C1 + C2;
```

is equivalent to

```
C3 = operator+(C1, C2);
```

In most cases, we will get the same results by the use of either a **friend** function or a member function. Why then an alternative is made available? There are certain situations where we would like to use a **friend** function rather than a member function. For instance, consider a situation where we need to use two different types of operands for a binary operator, say, one an object and another a built-in type data as shown below,

```
A = B + 2; (or A = B * 2;)
```

where **A** and **B** are objects of the same class. This will work for a member function but the statement

```
A = 2 + B; (or A = 2 * B)
```

will not work. This is because the left-hand operand which is responsible for invoking the member function should be an object of the same class. However **friend** function allows both approaches. How?

It may be recalled that an object need not be used to invoke a **friend** function but can be passed as an argument. Thus, we can use a friend function with a built-in type data as the *left-hand* operand and an object as the *right-hand* operand. Program 7.3 illustrates this, using scalar *multiplication* of a vector. It also shows how to overload the input and output operators >> and <<.

OVERLOADING OPERATORS USING FRIENDS

```
#include <iostream.h>

   const size = 3;

   class vector
   {
           int v[size];
    public:
           vector();                        // constructs null vector
           vector(int *x);                  // constructs vector from array
           friend vector operator *(int a, vector b);          // friend 1
           friend vector operator *(vector b, int a);          // friend 2
           friend istream & operator >> (istream &, vector &);
           friend ostream & operator << (ostream &, vector &);
   };

   vector :: vector()
   {
           for(int i=0; i<size; i++)
                   v[i] = 0;
   }

   vector :: vector(int *x)
   {
           for(int i=0; i<size; i++)
                   v[i] = x[i];
   }
```

(Contd)

```
vector operator *(int a, vector b)
{
        vector c;

        for(int i=0; i < size; i++)
                c.v[i] = a * b.v[i];
        return c;
}

        vector operator *(vector b, int a)
{
        vector c;

        for(int i=0; i<size; i++)
                c.v[i] = b.v[i] * a;
        return c;
}

istream & operator >> (istream &din, vector &b)

{
        for(int i=0; i<size; i++)
                din >> b.v[i];
        return(din);
}
ostream & operator <<  (ostream &dout, vector &b)
{
        dout << "(" << b.v [0];
        for(int i=1; i<size; i++)
                dout << ", " << b.v[i];
        dout << ")";
        return(dout);
}

int x[size] = {2,4,6};

int main()
{
        vector m;              // invokes constructor 1
        vector n = x;          // invokes constructor 2

        cout << "Enter elements of vector m " << "\n";
        cin >> m;              // invokes operator>>() function
```

(Contd)

```
        cout << "\n";
        cout << "m = " << m << "\n";          // invokes operator <<()

        vector p, q;

        p = 2 * m;              // invokes friend 1
        q = n * 2;              // invokes friend 2

        cout << "\n";
        cout << "p = " << p << "\n";          // invokes operator<<()
        cout << "q = " << q << "\n";

        return 0;
}
```

PROGRAM 7.3

Shown below is the output of Program 7.3:

```
Enter elements of vector m
5 10 15
m = (5, 10, 15)
p = (10, 20, 30)
q = (4, 8, 12)
```

The program overloads the operator * two times, thus overloading the operator function operator*() itself. In both the cases, the functions are explicitly passed two arguments and they are invoked like any other overloaded function, based on the types of its arguments. This enables us to use both the forms of scalar multiplication such as

```
p = 2 * m;          // equivalent to p = operator*(2,m);
q = n * 2;          // equivalent to q = operator*(n,2);
```

The program and its output are largely self-explanatory. The first constructor

```
vector();
```

constructs a vector whose elements are all zero. Thus

```
vector m;
```

creates a vector m and initializes all its elements to 0. The second constructor

```
vector(int &x);
```

creates a vector and copies the elements pointed to by the pointer argument x into it. Therefore, the statements

```
        if(t1 <= t3)
        {
                show(t1);
                cout << " smaller than ";
                show(t3);
                cout << "\n";
        }
        else
        {
                show(t3);
                cout << " smaller than ";
                show(t1);
                cout << "\n";
        }

        return 0;
}
```

PROGRAM 7.4

The following is the output of Program 7.4

```
t1 = New
t2 = York

t3 = New Delhi

New smaller than New Delhi
```

7.7 Rules for Overloading Operators

Although it looks simple to redefine the operators, there are certain restrictions and limitations in overloading them. Some of them are listed below:

1. Only existing operators can be overloaded. New operators cannot be created.
2. The overloaded operator must have at least one operand that is of user-defined type.
3. We cannot change the basic meaning of an operator. That is to say, we cannot redefine the plus(+) operator to subtract one value from the other.
4. Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
5. There are some operators that cannot be overloaded. (See Table 7.1.)
6. We cannot use **friend** functions to overload certain operators. (See Table 7.2.) However, member functions can be used to overload them.

7. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argument (the object of the relevant class).
8. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
9. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
10. Binary arithmetic operators such as +, –, *, and / must explicitly return a value. They must not attempt to change their own arguments.

Table 7.1 *Operators that cannot be overloaded*

Sizeof	Size of operator
.	Membership operator
.*	Pointer-to-member operator
::	Scope resolution operator
?:	Conditional operator

Table 7.2 *Where a friend cannot be used*

=	Assignment operator
( )	Function call operator
[ ]	Subscripting operator
->	Class member access operator

8

Inheritance: Extending Classes

Key Concepts

- Reusability
- Inheritance
- Single inheritance
- Multiple inheritance
- Multilevel inheritance
- Hybrid inheritance
- Hierarchical inheritance
- Defining a derived class
- Inheritiing private members
- Virtual base class
- Direct base class
- Indirect base class
- Abstract class
- Defining derived class constructors
- Nesting of classes

8.1 Introduction

Reusability is yet another important feature of OOP. It is always nice if we could reuse something that already exists rather than trying to create the same all over again. It would not only save time and money but also reduce frustration and increase reliability. For instance, the reuse of a class that has already been tested, debugged and used many times can save us the effort of developing and testing the same again.

Fortunately, C++ strongly supports the concept of *reusability*. The C++ classes can be reused in several ways. Once a class has been written and tested, it can be adapted by other programmers to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones. The mechanism of deriving a new class from an old one is called *inheritance (or derivation)*. The old class is referred to as the *base class* and the new one is called the *derived class or subclass*.

The derived class inherits some or all of the traits from the base class. A class can also inherit properties from more than one class or from more than one level. A derived class with only one base class, is called *single inheritance* and one with several base classes is called *multiple inheritance*. On the other hand, the traits of one class may be inherited by more than one class. This process is known as *hierarchical inheritance*. The mechanism of deriving a class from another 'derived class' is known as *multilevel inheritance*. Figure 8.1 shows various forms of inheritance that could be used for writing extensible programs. The direction of arrow indicates the direction of inheritance. (Some authors show the arrow in opposite direction meaning "inherited from".)

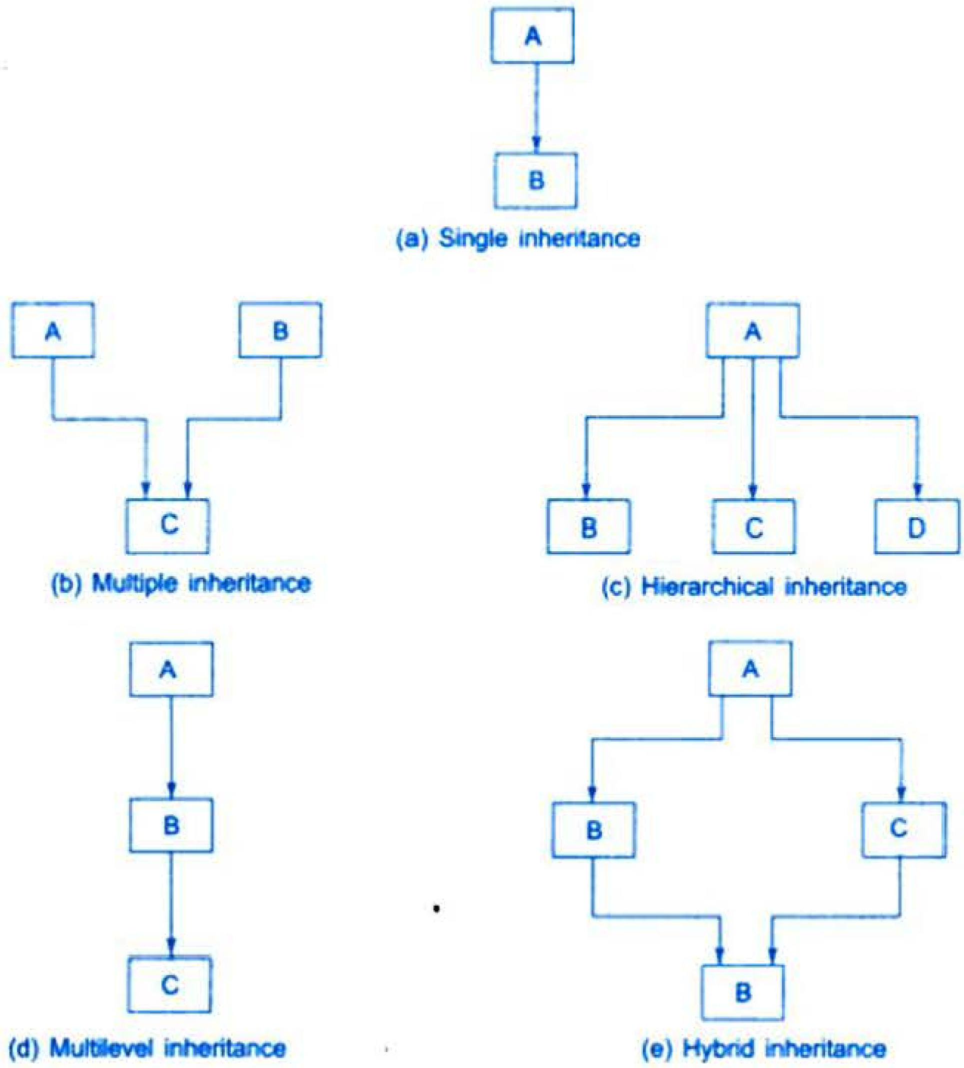


Fig. 8.1 ⇔ Forms of inheritance

8.2 Defining Derived Classes

A derived class can be defined by specifying its relationship with the base class in addition to its own details. The general form of defining a derived class is:

```
class derived-class-name  : visibility-mode base-class-name
{
        .....//
        .....//  members of derived class
        .....//
};
```

The colon indicates that the *derived-class-name* is derived from the *base-class-name*. The *visibility-mode* is optional and, if present, may be either **private** or **public**. The default visibility-mode is **private**. Visibility mode specifies whether the features of the base class are *privately derived or publicly derived*.

8.3 Single Inheritance

Let us consider a simple example to illustrate inheritance. Program 8.1 shows a base class B and a derived class D. The class B contains one private data member, one public data member, and three public member functions. The class D contains one private data member and two public member functions.

SINGLE INHERITANCE : PUBLIC

```
#include <iostream>

using namespace std;

class B
{
        int a;                  // private; not inheritable
   public:
        int b;                  // public; ready for inheritance
        void get_ab();
        int  get_a(void);
        void show_a(void);
};

class D : public B              // public derivation
{
        int c;
   public:
        void mul(void);
        void display(void);
};
//--------------------------------------------------------------
void B :: get_ab(void)
{
        a = 5; b = 10;
}
int B :: get_a()
{
        return a;
}
void B :: show_a()
{
```

(Contd)

```
        cout << "a = " << a << "\n";
}
void D :: mul()
{
        c = b * get_a();
}
void D :: display()
{
        cout << "a = " << get_a() << "\n";
        cout << "b = " << b << "\n";
        cout << "c = " << c << "\n\n";
}
//--------------------------------------------------------
int main()
{
        D d;

        d.get_ab();
        d.mul();
        d.show_a();
        d.display();

        d.b = 20;
        d.mul();
        d.display();

        return 0;
}
```

PROGRAM 8.1

8.5 Multilevel Inheritance

It is not uncommon that a class is derived from another derived class as shown in Fig. 8.7. The class **A** serves as a base class for the derived class **B**, which in turn serves as a base class for the derived class **C**. The class **B** is known as *intermediate* base class since it provides a link for the inheritance between **A** and **C**. The chain **ABC** is known as *inheritance path*.

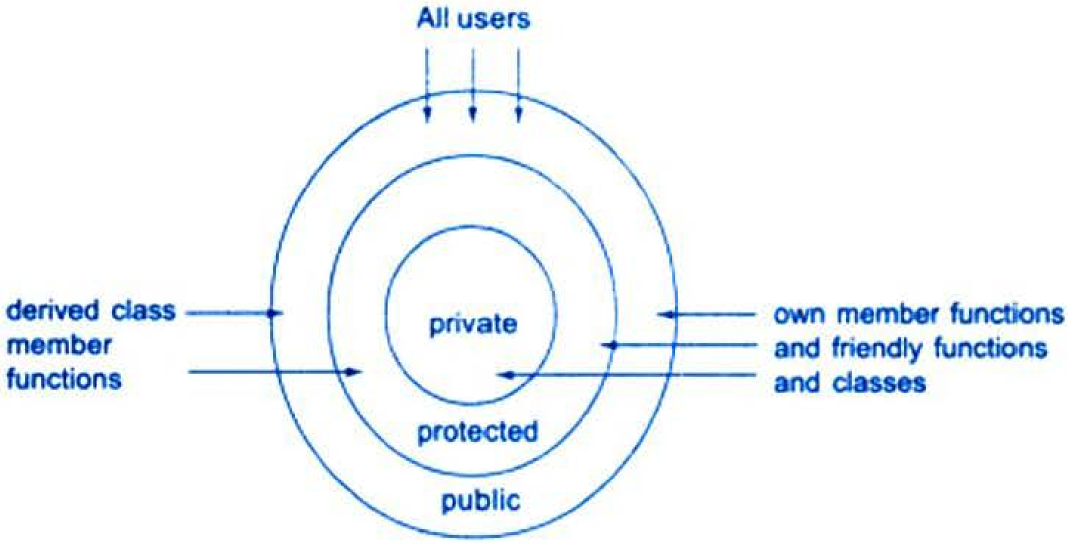


Fig. 8.6 ⇔ *A simple view of access control to the members of a class*

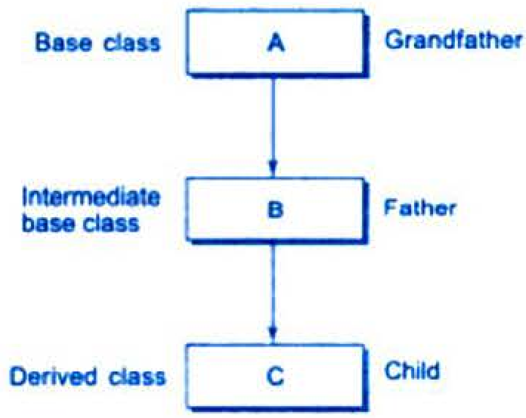


Fig. 8.7 ⇔ *Multilevel inheritance*

A derived class with multilevel inheritance is declared as follows:

```
class A{.....};                  // Base class
class B: public A {.....};       // B derived from A
class C: public B {.....};       // C derived from B
```

This process can be extended to any number of levels.

Let us consider a simple example. Assume that the test results of a batch of students are stored in three different classes. Class **student** stores the roll-number, class **test** stores the marks obtained in two subjects and class **result** contains the **total** marks obtained in the test. The class **result** can inherit the details of the marks obtained in the test and the roll-number of students through multilevel inheritance. Example:

MULTILEVEL INHERITANCE

```
#include <iostream>

using namespace std;

class student
```

(Contd)

```
{
  protected:
          int roll_number;
  public:
          void get_number(int);
          void put_number(void);
};

void student :: get_number(int a)
{
          roll_number = a;
}

void student :: put_number()
{
          cout << "Roll Number: " << roll_number << "\n";
}

class test : public student              // First level derivation
{
  protected:
      float sub1;
      float sub2;
  public:
        void get_marks(float, float);
        void put_marks(void);
};

void test :: get_marks(float x, float y)
{
     sub1 = x;
     sub2 = y;
}

void test :: put_marks()
{
        cout << "Marks in SUB1 = " << sub1 << "\n";
        cout << "Marks in SUB2 = " << sub2 << "\n";
}

class result : public test        // Second level derivation
{
      float total;                // private by default
  public:
      void display(void);
};

void result :: display(void)
{
```

(Contd)

```
            total = sub1 + sub2;
            put_number();
            put_marks();
            cout << "Total = " << total << "\n";
      }

      int main()
      {
            result student1;         // student1 created

            student1.get_number(111);
            student1.get_marks(75.0, 59.5);

            student1.display();

         return 0;
      }
```

PROGRAM 8.3

Program 8.3 displays the following output:

```
Roll Number: 111
Marks in SUB1 = 75
Marks in SUB2 = 59.5
Total = 134.5
```

8.6 Multiple Inheritance

A class can inherit the attributes of two or more classes as shown in Fig. 8.8. This is known as *multiple inheritance*. Multiple inheritance allows us to combine the features of several existing classes as a starting point for defining new classes. It is like a child inheriting the physical features of one parent and the intelligence of another.

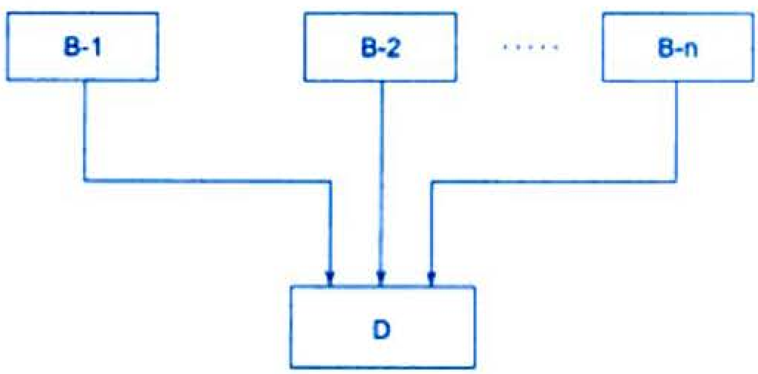


Fig. 8.8 ⇔ *Multiple inheritance*

The syntax of a derived class with multiple base classes is as follows:

```
class D: visibility B-1, visibility B-2 ...
{
        .....
        .....(Body of D)
        .....
};
```

where, *visibility* may be either **public** or **private**. The base classes are separated by commas.

MULTIPLE INHERITANCE

```
#include <iostream>

using namespace std;

class M
{
  protected:
        int m;
  public:
        void get_m(int);
};

class N
{
  protected:
      int n;
  public:
      void get_n(int);
};

class P : public M, public N
{
  public:
      void display(void);
};

void M :: get_m(int x)
{
      m = x;
}

void N :: get_n(int y)
{
      n = y;
}

void P :: display(void)
{
      cout << "m = " << m << "\n";
      cout << "n = " << n << "\n";
      cout << "m*n = " << m*n << "\n";
}

int main()
{
```

(Contd)

```
        P p;

        p.get_m(10);
        p.get_n(20);
        p.display();

        return 0;
}
```

PROGRAM 8.4

8.7 Hierarchical Inheritance

We have discussed so far how inheritance can be used to modify a class when it did not satisfy the requirements of a particular problem on hand. Additional members are added through inheritance to extend the capabilities of a class. Another interesting application of inheritance is to use it as a support to the hierarchical design of a program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below that level.

As an example, Fig. 8.9 shows a hierarchical classification of students in a university. Another example could be the classification of accounts in a commercial bank as shown in Fig. 8.10. All the students have certain things in common and, similarly, all the accounts possess certain common features.

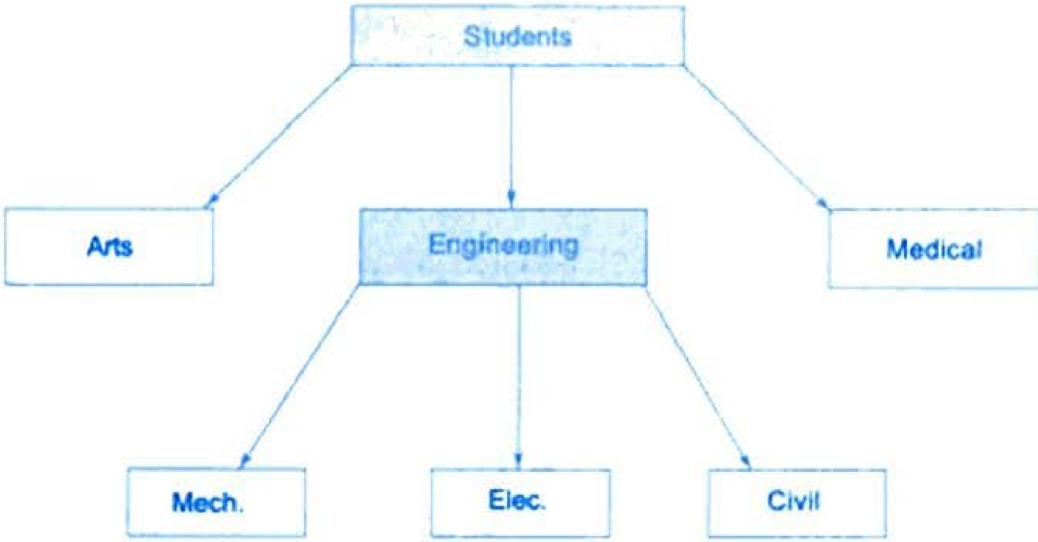


Fig. 8.9 ⇔ *Hierarchical classification of students*

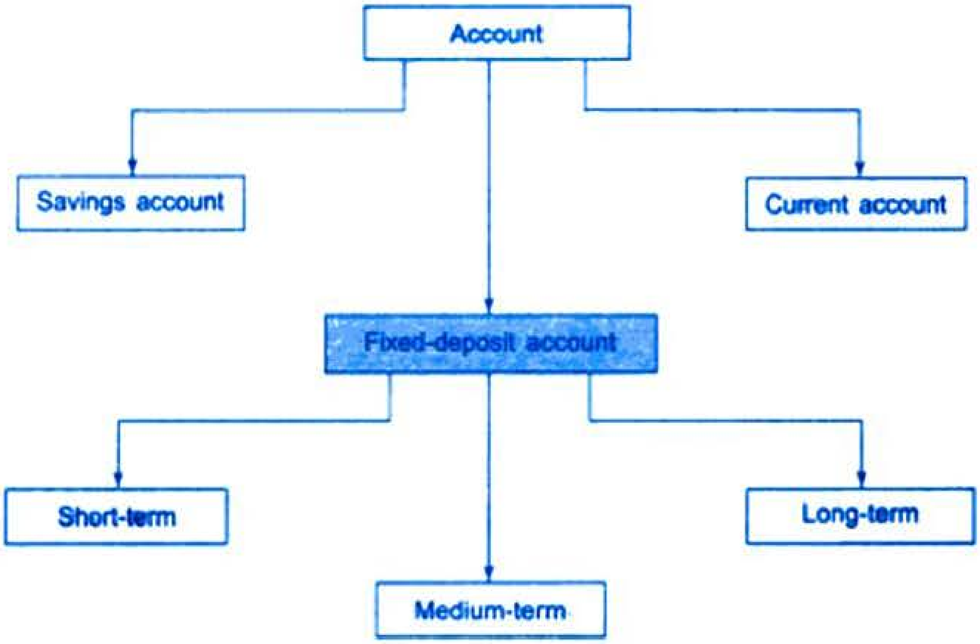


Fig. 8.10 ⇔ *Classification of bank accounts*

In C++, such problems can be easily converted into class hierarchies. The base class will include all the features that are common to the subclasses. A *subclass* can be constructed by inheriting the properties of the base class. A subclass can serve as a base class for the lower level classes and so on.

8.8 Hybrid Inheritance

There could be situations where we need to apply two or more types of inheritance to design a program. For instance, consider the case of processing the student results discussed in Sec. 8.5. Assume that we have to give weightage for sports before finalising the results. The weightage for sports is stored in a separate class called **sports**. The new inheritance relationship between the various classes would be as shown in Fig. 8.11.

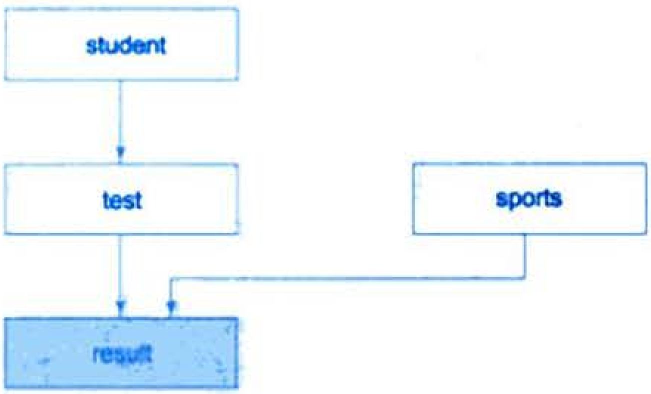


Fig. 8.11 ⇔ *Multilevel, multiple inheritance*

HYBRID INHERITANCE

```
#include <iostream>

using namespace std;

class student
{
  protected:
      int  roll_number;
  public:
      void get_number(int a)
      {
          roll_number = a;
```

(Contd)

```
        }
        void put_number(void)
        {
            cout << "Roll No: " << roll_number << "\n";
        }
};

class test : public student
{
  protected:
        float part1, part2;
  public:
        void get_marks(float x, float y)
        {
            part1 = x;  part2 = y;
        }
        void put_marks(void)
        {
            cout << "Marks obtained: " << "\n"
                 << "Part1 = " << part1 << "\n"
                 << "Part2 = " << part2 << "\n";
        }
};

class sports
{
   protected:
        float score;
   public:
        void get_score(float s)
        {
            score = s;
        }
        void put_score(void)
        {
            cout << "Sports wt: " << score << "\n\n";
        }
};

class result : public test, public sports
{
        float total;
  public:
        void display(void);
```

(Contd)

```
};

void result :: display(void)
{
        total = part1 + part2 + score;

        put_number();
        put_marks();
        put_score();

        cout << "Total Score: " << total << "\n";
}

int main()
{
        result student_1;
        student_1.get_number(1234);
        student_1.get_marks(27.5, 33.0);
        student_1.get_score(6.0);
        student_1.display();

        return 0;
}
```

PROGRAM 8.5

Here is the output of Program 8.5:

```
Roll No: 1234
Marks obtained:
Part1 = 27.5
Part2 = 33
Sports wt: 6

Total Score: 66.5
```

8.9 Virtual Base Classes

We have just discussed a situation which would require the use of both the multiple and multilevel inheritance. Consider a situation where all the three kinds of inheritance, namely, multilevel, multiple and hierarchical inheritance, are involved. This is illustrated in Fig. 8.12. The 'child' has two *direct base classes* 'parent1' and 'parent2' which themselves have a common base class 'grandparent'. The 'child' inherits the traits of 'grandparent' via two separate paths. It can also inherit directly as shown by the broken line. The 'grandparent' is sometimes referred to as *indirect base class*.

9.6 Virtual Functions

As mentioned earlier, polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but in different forms. An essential requirement of polymorphism is therefore the ability to refer to objects without any regard to their classes. This necessitates the use of a single pointer variable to refer to the objects of different classes. Here, we use the pointer to base class to refer to all the derived objects. But, we just discovered that a base pointer, even when it is made to contain the address of a derived class, always executes the function in the base class. The compiler simply ignores the contents of the pointer and chooses the member function that matches the type of the pointer. How do we then achieve polymorphism?. It is achieved using what is known as 'virtual' functions.

```
#include <iostream>

using namespace std;

class Base
{
  public:
        void display() {cout << "\n Display base ";}
        virtual void show() {cout << "\n show base";}
};
class Derived : public Base
{
  public:
        void display() {cout << "\n Display derived";}
        void show() {cout << "\n show derived";}
};

int main()
{
        Base B;
        Derived D;
        Base *bptr;

        cout << "\n bptr points to Base \n";
        bptr = &B;
        bptr -> display();    // calls Base version
        bptr -> show();       // calls Base version

        cout << "\n\n bptr points to Derived\n";
        bptr = &D;
        bptr -> display();    // calls Base version
        bptr -> show();       // calls Derived version

        return 0;
}
```

PROGRAM 9.12

One important point to remember is that, we must access **virtual** functions through the use of a pointer declared as a pointer to the base class. Why can't we use the object name (with the dot operator) the same way as any other member function to call the virtual functions?. We can, but remember, run time polymorphism is achieved only when a virtual function is accessed through a pointer to the base class.

Let us take an example where **virtual** functions are implemented in practice. Consider a book shop which sells both books and video-tapes. We can create a class known as **media** that stores the title and price of a publication. We can then create two derived classes, one for storing the number of pages in a book and another for storing the playing time of a tape. Figure 9.2 shows the class hierarchy for the book shop.

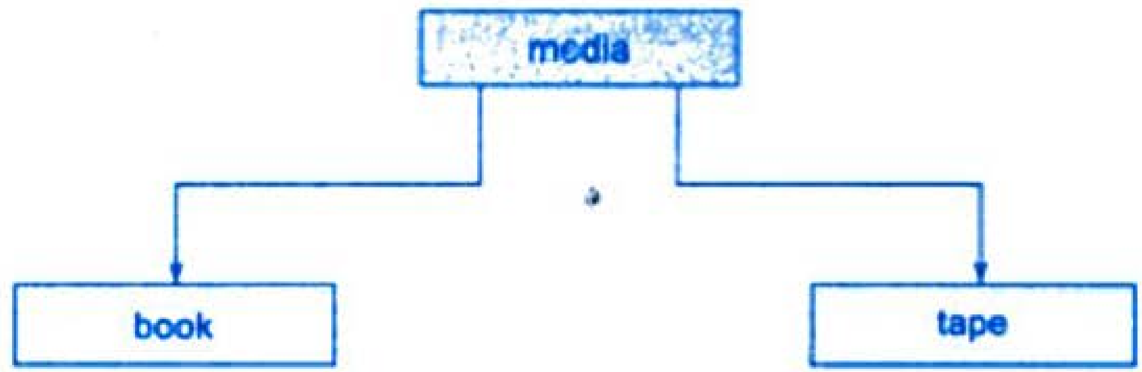


Fig. 9.2 ⇔ *The class hierarchy for the book shop*

```
        tape tape1(title, price, time);

        media* list[2];
        list[0] = &book1;
        list[1] = &tape1;

        cout << "\n MEDIA DETAILS";

        cout << "\n ......BOOK......";
        list[0] -> display(); // display book details

        cout << "\n ......TAPE......";
        list[1] -> display(); // display tape details

        result 0;
}
```

PROGRAM 9.13

The output of Program 9.13 would be:

```
ENTER BOOK DETAILS
Title:Programming_in_ANSI_C
Price: 88
Pages: 400

ENTER TAPE DETAILS
Title: Computing_Concepts
Price: 90
Play time (mins): 55

MEDIA DETAILS
......BOOK......
Title:Programming_in_ANSI_C
Pages: 400
Price: 88

.....TAPE......
Title: Computing_Concepts
Play time: 55mins
Price: 90
```

Rules for Virtual Functions

When virtual functions are created for implementing late binding, we should observe some basic rules that satisfy the compiler requirements:

1. The virtual functions must be members of some class.
2. They cannot be static members.
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in a base class must be defined, even though it may not be used.
6. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. If two functions with the same name have different prototypes, C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
7. We cannot have virtual constructors, but we can have virtual destructors.
8. While a base pointer can point to any type of the derived object, the reverse is not true. That is to say, we cannot use a pointer to a derived class to access an object of the base type.
9. When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore, we should not use this method to move the pointer to the next object.
10. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

9.7 Pure Virtual Functions

It is normal practice to declare a function virtual inside the base class and redefine it in the derived classes. The function inside the base class is seldom used for performing any task. It only serves as a *placeholder*. For example, we have not defined any object of class media and therefore the function display() in the base class has been defined 'empty'. Such functions are called "do-nothing" functions.

A "do-nothing" function may be defined as follows:

```
virtual void display() = 0;
```

Such functions are called *pure virtual* functions. A pure virtual function is a function declared in a base class that has no definition relative to the base class. In such cases, the compiler requires each derived class to either define the function or redeclare it as a pure virtual function. Remember that a class containing pure virtual functions cannot be used to declare any objects of its own. As stated earlier, such classes are called *abstract base classes*. The main objective of an abstract base class is to provide some traits to the derived classes and to create a base pointer required for achieving run time polymorphism.

COMMAND-LINE ARGUMENTS

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

int main(int argc, char * argv[])
{
        int number[9] = {11,22,33,44,55,66,77,88,99};

        if(argc != 3)
        {
                cout << "argc = " << argc << "\n";
                cout << "Error in arguments \n";
                exit(1);
        }
        ofstream fout1, fout2;

        fout1.open(argv[1]);

        if(fout1.fail())
        {
                cout << "could not open the file"
                     << argv[1] << "\n";
                exit(1);
        }

        fout2.open(argv[2]);

        if(fout2.fail())
        {
                cout << "could not open the file "
                     << argv[2]  << "\n";
                exit(1);
        }

for(int i=0; i<9; i++)
{
        if(number[i] % 2 == 0)
        fout2 << number[i] << " ";              // write to EVEN file
        else
        fout1 << number[i] << " ";              // write to ODD file
}
```

(Contd)

```
    fout1.close();
    fout2.close();

    ifstream  fin;
    char  ch;
    for(i=1; i<argc; i++)
    {
            fin.open(argv[i]);
            cout << "Contents of " << argv[i] << "\n";
            do
            {
                    fin.get(ch);  // read a value
                    cout << ch;   // display it
            }
            while(fin);
            cout << "\n\n";
            fin.close();
    }
    return 0;
}
```

PROGRAM 11.8

The output of Program 11.8 would be:

```
Contents of ODD
11 33 55 77 99

Contents of EVEN
22 44 66 88
```

SUMMARY