11

Working with Files

Key Concepts

- Console-user interaction
- Input stream
- Output stream
- File stream classes
- Opening a file with open()
- Opening a file with constructors
- End-of-file detection
- File modes
- File pointers
- Sequential file operations
- Random access files
- Error handling
- Command-line arguments

11.1 Introduction

Many real-life problems handle large volumes of data and, in such situations, we need to use some devices such as floppy disk or hard disk to store the data. The data is stored in these devices using the concept of *files*. A file is a collection of related data stored in a particular area on the disk. Programs can be designed to perform the read and write operations on these files.

A program typically involves either or both of the following kinds of data communication:

1. Data transfer between the console unit and the program.
2. Data transfer between the program and a disk file.

This is illustrated in Fig. 11.1.

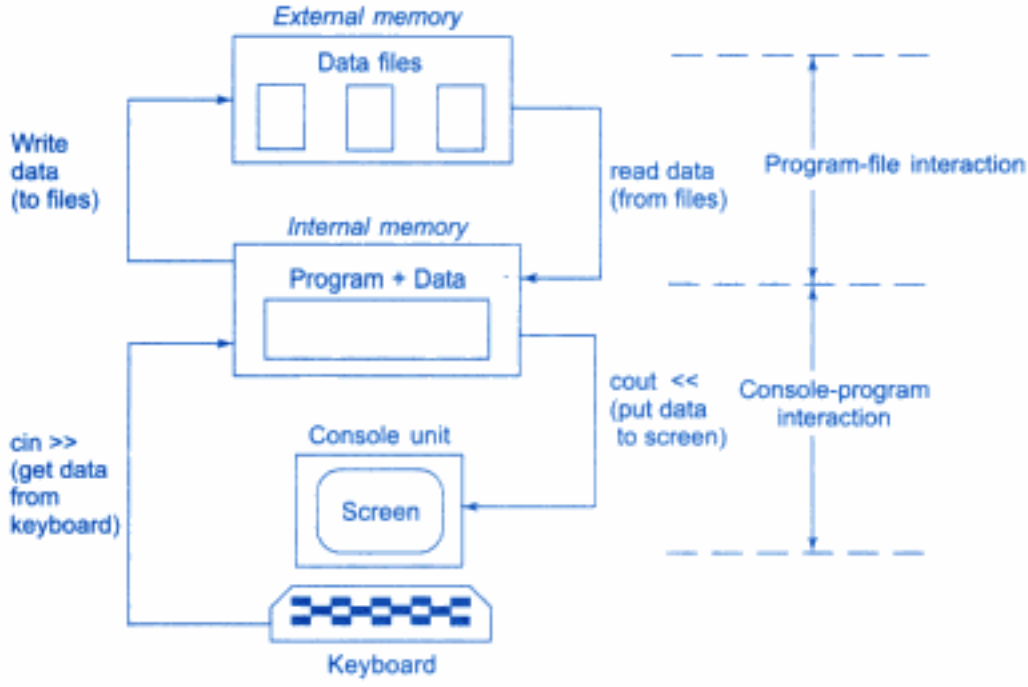


Fig. 11.1 ⇔ *Consol-program-file interaction*

We have already discussed the technique of handling data communication between the console unit and the program. In this chapter, we will discuss various methods available for storing and retrieving the data from files.

The I/O system of C++ handles file operations which are very much similar to the console input and output operations. It uses file streams as an interface between the programs and the files. The stream that supplies data to the program is known as *input stream* and the one that receives data from the program is known as *output stream*. In other words, the input stream extracts (or reads) data from the file and the output stream inserts (or writes) data to the file. This is illustrated in Fig. 11.2.

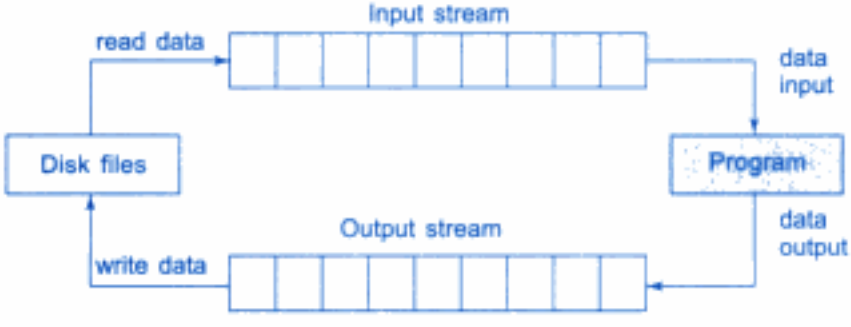


Fig. 11.2 ⇔ *File input and output streams*

The input operation involves the creation of an input stream and linking it with the program and the input file. Similarly, the output operation involves establishing an output stream with the necessary links with the program and the output file.

11.2 Classes for File Stream Operations

The I/O system of C++ contains a set of classes that define the file handling methods. These include **ifstream, ofstream** and **fstream**. These classes are derived from **fstreambase** and from the corresponding *iostream* class as shown in Fig. 11.3. These classes, designed to manage the disk files, are declared in *fstream* and therefore we must include this file in any program that uses files.

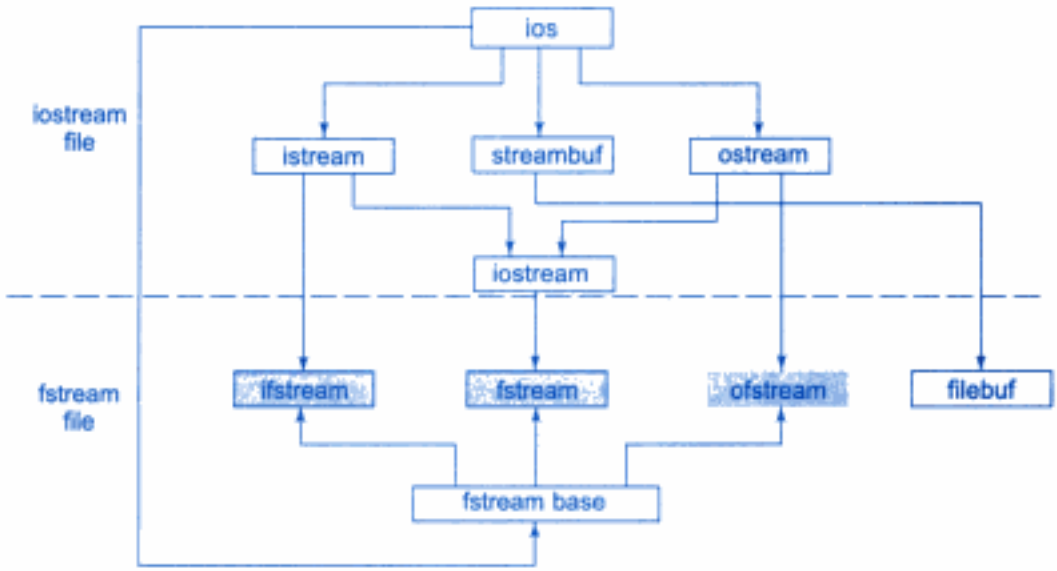


Fig. 11.3 ⇔ *Stream classes for file operations (contained in fstream file)*

Table 11.1 shows the details of file stream classes. Note that these classes contain many more features. For more details, refer to the manual.

11.3 Opening and Closing a File

If we want to use a disk file, we need to decide the following things about the file and its intended use:

1. Suitable name for the file.
2. Data type and structure.

3. Purpose.
4. Opening method.

Table 11.1 *Details of file stream classes*

Class	*Contents*
`filebuf`	Its purpose is to set the file buffers to read and write. Contains **Openprot** constant used in the **open()** of file stream classes. Also contain **close()** and **open()** as members.
`fstreambase`	Provides operations common to the file streams. Serves as a base for **fstream, ifstream** and **ofstream** class. Contains **open()** and **close()** functions.
`ifstream`	Provides input operations. Contains **open()** with default input mode. Inherits the functions **get(), getline(), read(), seekg()** and **tellg()** functions from **istream**.
`Ofstream`	Provides output operations. Contains **open()** with default output mode. Inherits **put(), seekp(), tellp()**, and **write()**, functions from **ostream**.
`fstream`	Provides support for simultaneous input and output operations. Contains **open()** with default input mode. Inherits all the functions from **istream** and **ostream** classes through **iostream**.

The filename is a string of characters that make up a valid filename for the operating system. It may contain two parts, a primary name and an optional period with extension. Examples:

```
Input.data
Test.doc
INVENT.ORY
student
salary
OUTPUT
```

As stated earlier, for opening a file, we must first create a file stream and then link it to the filename. A file stream can be defined using the classes **ifstream, ofstream,** and **fstream** that are contained in the header file *fstream*. The class to be used depends upon the purpose, that is, whether we want to read data from the file or write data to it. A file can be opened in two ways:

1. Using the constructor function of the class.
2. Using the member function **open()** of the class.

The first method is useful when we use only one file in the stream. The second method is used when we want to manage multiple files using one stream.

Opening Files Using Constructor

We know that a constructor is used to initialize an object while it is being created. Here, a filename is used to initialize the file stream object. This involves the following steps:

1. Create a file stream object to manage the stream using the appropriate class. That is to say, the class **ofstream** is used to create the output stream and the class **ifstream** to create the input stream.
2. Initialize the file object with the desired filename.

For example, the following statement opens a file named "results" for output:

```
ofstream outfile("results");   // output only
```

This creates **outfile** as an **ofstream** object that manages the output stream. This object can be any valid C++ name such as **o_file, myfile** or **fout**. This statement also opens the file results and attaches it to the output stream **outfile**. This is illustrated in Fig. 11.4.

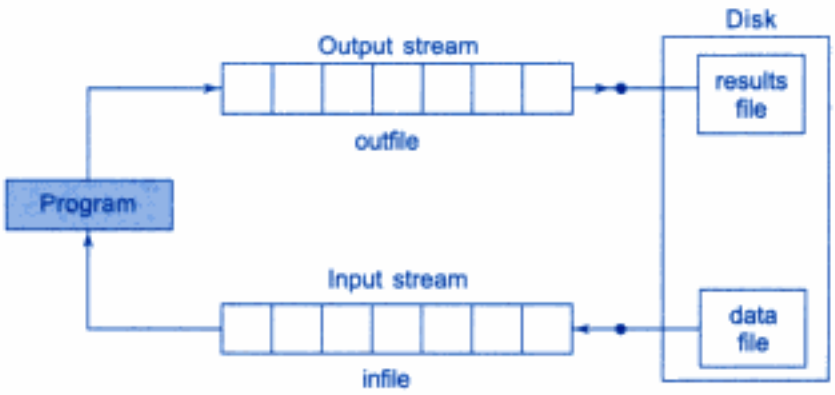


Fig. 11.4 ⇔ *Two file streams working on separate files*

Similarly, the following statement declares **infile** as an **ifstream** object and attaches it to the file **data** for reading (input).

```
ifstream  infile("data");  // input only
```

The program may contain statements like:

```
outfile << "TOTAL";
outfile << sum;
infile  >> number;
infile  >> string;
```

We can also use the same file for both reading and writing data as shown in Fig. 11.5. The programs would contain the following statements:

```
Program1
   .....
   .....
```

```
   ofstream outfile("salary");          // creates outfile and connects
                                        // "salary" to it
   .....
   .....
Program2
   .....
   .....
   ifstream infile("salary");           // creates infile and connects
                              // "salary" to it
   .....
   .....
```

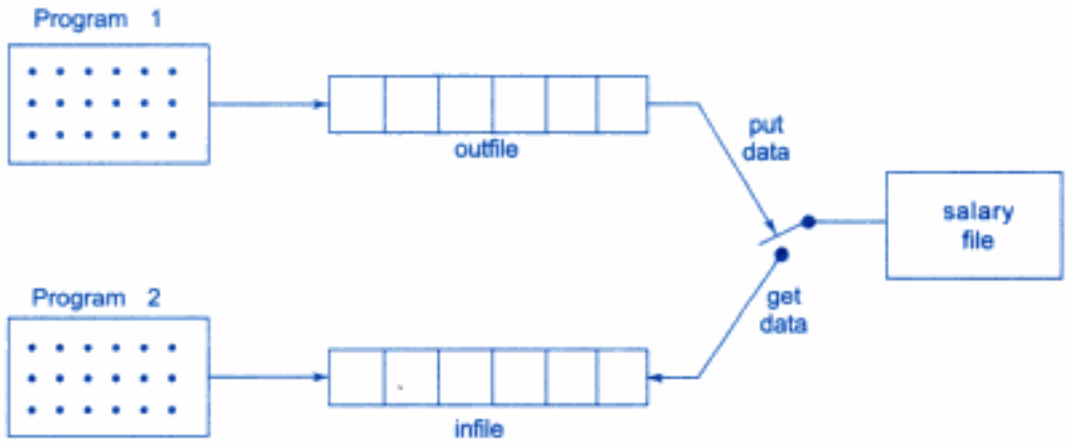


Fig. 11.5 ⇔ *Two file streams working on one file*

The connection with a file is closed automatically when the stream object expires (when the program terminates). In the above statement, when the *program1* is terminated, the **salary** file is disconnected from the **outfile** stream. Similar action takes place when the *program 2* terminates.

Instead of using two programs, one for writing data (output) and another for reading data (input), we can use a single program to do both the operations on a file. Example.

```
.....
.....
     outfile.close();                // Disconnect salary from outfile
     ifstream infile("salary");      // and connect to infile
.....
.....
      infile.close();   // Disconnect salary from infile
```

Although we have used a single program, we created two file stream objects, **outfile** (to put data to the file) and **infile** (to get data from the file). Note that the use of a statement like

```
outfile.close();
```

disconnects the file salary from the output stream **outfile**. Remember, the object **outfile** still exists and the **salary** file may again be connected to **outfile** later or to any other stream. In this example, we are connecting the **salary** file to **infile** stream to read data.

Program 11.1 uses a single file for both writing and reading the data. First, it takes data from the keyboard and writes it to the file. After the writing is completed, the file is closed. The program again opens the same file, reads the information already written to it and displays the same on the screen.

WORKING WITH SINGLE FILE

```
// Creating files with constructor function

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

int main()
{
        ofstream outf("ITEM");          // connect ITEM file to outf

        cout << "Enter item name:";
        char name[30];
        cin >> name;                    // get name from key board and

        outf << name << "\n";           // write to file ITEM

        cout << "Enter item cost:";
        float cost;
        cin >> cost;                    // get cost from key board and

        outf << cost << "\n";           // write to file ITEM

        outf.close();                   // Disconnect ITEM file from outf

        ifstream inf("ITEM");           // connect ITEM file to inf

        inf >>  name;                   // read name from file ITEM
        inf >> cost;                    // read cost from file ITEM
```

(Contd)

```
        cout << "\n";
        cout << "Item name:" << name << "\n";
        cout << "Item cost:" << cost << "\n";

        inf.close();                    // Disconnect ITEM from inf

        return 0;
}
```

PROGRAM 11.1

The output of Program 11.1 would be:

```
Enter item name:CD-ROM
Enter item cost:250

Item name:CD-ROM
Item cost:250
```

caution

When a file is opened for *writing only*, a new file is created if there is no file of that name. If a file by that name exists already, then its contents are deleted and the file is presented as a clean file. We shall discuss later how to open an existing file for updating it without losing its original contents.

Opening Files Using open()

As stated earlier, the function **open()** can be used to open multiple files that use the same stream object. For example, we may want to process a set of files sequentially. In such cases, we may create a single stream object and use it to open each file in turn. This is done as follows:

```
file-stream-class stream-object;
stream-object.open ("filename");
```

Example:

```
ofstream outfile;           // Create stream (for output)
outfile.open("DATA1");      // Connect stream to DATA1
.....
.....
outfile.close();            // Disconnect stream from DATA1
outfile.open("DATA2);       // Connect stream to DATA2
.....
.....
outfile.close();            // Disconnect stream from DATA2
.....
.....
```

The previous program segment opens two files in sequence for writing the data. Note that the first file is closed before opening the second one. This is necessary because a stream can be connected to only one file at a time. See Program 11.2 and Fig. 11.6.

WORKING WITH MULTIPLE FILES

```
// Creating files with open() function

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

int main()
{
  ofstream fout;                          // create output stream
  fout.open("country");                   // connect "country" to it
  fout << "United States of America\n";
  fout << "United Kingdom\n";
  fout << "South Korea\n";

  fout.close();                           // disconnect "country" and

  fout.open("capital");                   // connect "capital"

  fout  << "Washington\n";
  fout  << "London\n";
  fout  << "Seoul\n";

  fout.close();                           // disconnect "capital"

  // Reading the files
  const int N = 80;                       // size of line
  char line[N];

  ifstream fin;                           // create input stream
  fin.open("country");                    // connect "country" to it

  cout <<"contents of country file\n";

  while(fin)                              // check end-of-file
  {
        fin.getline(line, N);             // read a line
        cout << line ;                    // display it
  }

  fin.close();                            // disconnect "country" and
```

(Contd)

```
    fin.open("capital");                    // connect "capital"

    cout << "\nContents of capital file \n";

    while(fin)
    {
          fin.getline(line, N);
          cout << line ;
    }
    fin.close();

    return 0;
}
```

PROGRAM 11.2

The output of Program 11.2 would be:

```
Contents of country file
United States of America
United Kingdom
South Korea

Contents of capital file
Washington
London
Seoul
```

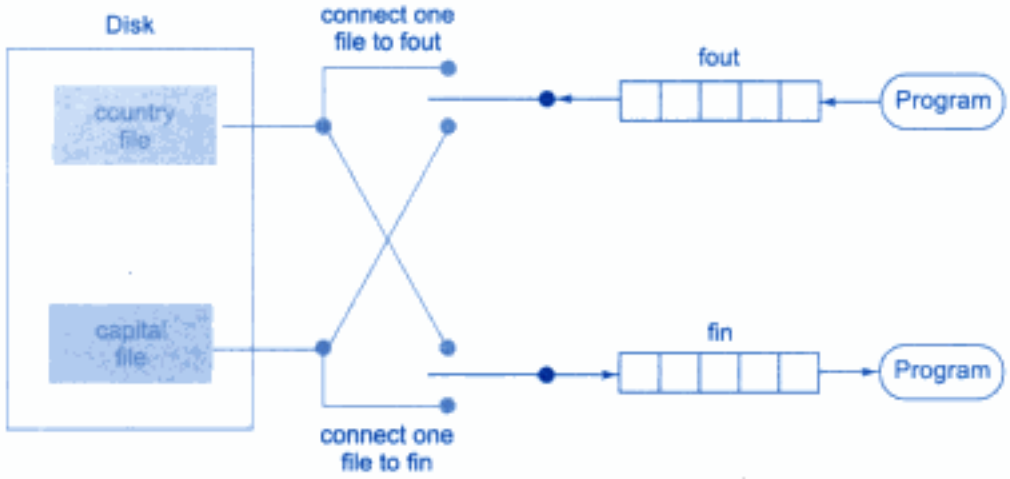


Fig. 11.6 ⇔ *Streams working on multiple files*

At times we may require to use two or more files simultaneously. For example, we may require to merge two sorted files into a third sorted file. This means, both the sorted files have to be kept open for reading and the third one kept open for writing. In such cases, we

need to create two separate input streams for handling the two input files and one output stream for handling the output file. See Program 11.3.

READING FROM TWO FILES SIMULTANEOUSLY

```
// Reads the files created in Program 11.2

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>              // for exit() function

int main()
{
        const  int SIZE = 80;
        char line[SIZE];

        ifstream fin1, fin2;          // create two input streams
        fin1.open("country");
        fin2.open("capital");

        for(int i=1; i<=10; i++)
        {
              if(fin1.eof() != 0)
              {
                    cout << "Exit from country \n";
                    exit(1);
        }
        fin1.getline(line, SIZE);
        cout << "Capital of "<< line ;

        if(fin2.eof() != 0)
        {
              cout << "Exit from capital\n";
              exit(1);
        }

        fin2.getline(line,SIZE);
        cout << line << "\n";
   }
   return 0;
}
```

PROGRAM 11.3

The output of Program 11.3 would be:

```
Capital of United States of America
Washington
```

```
Capital of United Kingdom
London
Capital of South Korea
Seoul
```

11.4 Detecting end-of-file

Detection of the end-of-file condition is necessary for preventing any further attempt to read data from the file. This was illustrated in Program 11.2 by using the statement

```
while(fin)
```

An **ifstream** object, such as **fin**, returns a value of 0 if any error occurs in the file operation including the end-of-file condition. Thus, the **while** loop terminates when **fin** returns a value of zero on reaching the end-of-file condition. Remember, this loop may terminate due to other failures as well. (We will discuss other error conditions later.)

There is another approach to detect the end-of-file condition. Note that we have used the following statement in Program 11.3:

```
if(fin1.eof() != 0) {exit(1);}
```

eof() is a member function of **ios** class. It returns a non-zero value if the end-of-file(EOF) condition is encountered, and a zero, otherwise. Therefore, the above statement terminates the program on reaching the end of the file.

11.5 More about Open(): File Modes

We have used **ifstream** and **ofstream** constructors and the function **open()** to create new files as well as to open the existing files. Remember, in both these methods, we used only one argument that was the filename. However, these functions can take two arguments, the second one for *specifying the file mode*. The general form of the function **open()** with two arguments is:

```
stream-object.open("filename", mode);
```

The second *argument mode* (called file mode parameter) specifies the purpose for which the file is opened. How did we then open the files without providing the second argument in the previous examples?

The prototype of these class member functions contain default values for the second argument and therefore they use the default values in the absence of the actual values. The

default values are as follows:

```
ios::in  for ifstream functions meaning open for reading only.
ios::out for ofstream functions meaning open for writing only.
```

The *file mode* parameter can take one (or more) of such constants defined in the class **ios**. Table 11.2 lists the file mode parameters and their meanings.

Table 11.2 *File mode parameters*

Parameter	Meaning
ios :: app	Append to end-of-file
ios :: ate	Go to end-of-file on opening
ios :: binary	Binary file
ios :: in	Open file for reading only
ios :: nocreate	Open fails if the file does not exist
ios :: noreplace	Open fails if the file already exists
ios :: out	Open file for writing only
ios :: trunc	Delete the contents of the file if it exists

note

1. Opening a file in **ios::out** mode also opens it in the **ios::trunc** mode by default.
2. Both **ios::app** and **ios::ate** take us to the end of the file when it is opened. The difference between the two parameters is that the **ios::app** allows us to add data to the end of the file only, while **ios::ate** mode permits us to add data or to modify the existing data anywhere in the file. In both the cases, a file is created by the specified name, if it does not exist.
3. The parameter **ios::app** can be used only with the files capable of output.
4. Creating a stream using **ifstream** implies input and creating a stream using **ofstream** implies output. So in these cases it is not necessary to provide the mode parameters.
5. The **fstream** class does not provide a mode by default and therefore, we must provide the mode explicitly when using an object of **fstream** class.
6. The *mode* can combine two or more parameters using the bitwise OR operator (symbol |) as shown below:

```
fout.open("data", ios::app | ios:: nocreate)
```

This opens the file in the append mode but fails to open the file if it does not exist.

11.6 File Pointers and Their Manipulations

Each file has two associated pointers known as the *file pointers*. One of them is called the input pointer (*or get pointer*) and the other is called the output pointer (*or put pointer*). We

can use these pointers to move through the files while reading or writing. The input pointer is used for reading the contents of a given file location and the output pointer is used for writing to a given file location. Each time an input or output operation takes place, the appropriate pointer is automatically advanced.

Default Actions

When we open a file in read-only mode, the input pointer is automatically set at the beginning so that we can read the file from the start. Similarly, when we open a file in write-only mode, the existing contents are deleted and the output pointer is set at the beginning. This enables us to write to the file from the start. In case, we want to open an existing file to add more data, the file is opened in 'append' mode. This moves the output pointer to the end of the file (i.e. the end of the existing contents). See Fig. 11.7.

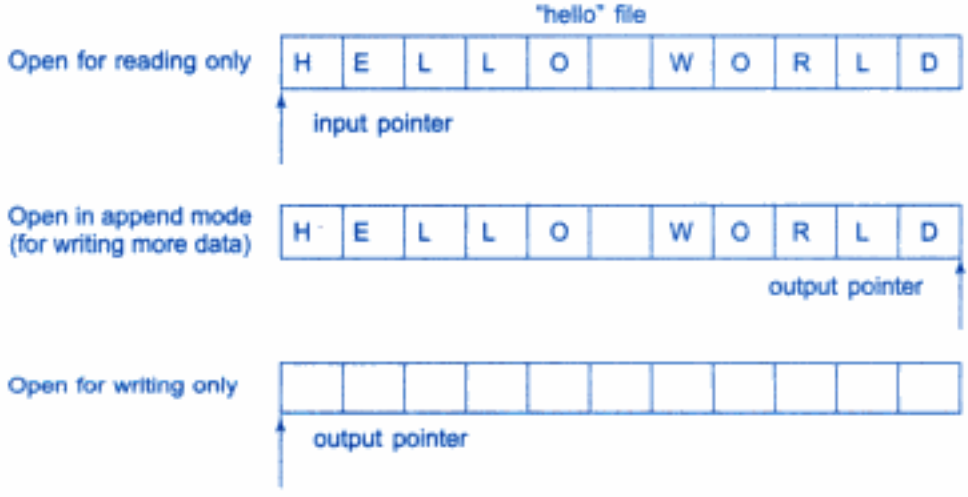


Fig. 11.7 ⇔ *Action on file pointers while opening a file*

Functions for Manipulation of File Pointers

All the actions on the file pointers as shown in Fig. 11.7 take place automatically by default. How do we then move a file pointer to any other desired position inside the file? This is possible only if we can take control of the movement of the file pointers ourselves. The file stream classes support the following functions to manage such situations:

- **seekg()** Moves get pointer (input) to a specified location.
- **seekp()** Moves put pointer(output) to a specified location.
- **tellg()** Gives the current position of the get pointer.
- **tellp()** Gives the current position of the put pointer.

For example, the statement

```
infile.seekg(10);
```

moves the file pointer to the byte number 10. Remember, the bytes in a file are numbered beginning from zero. Therefore, the pointer will be pointing to the 11th byte in the file.

Consider the following statements:

```
ofstream  fileout;
fileout.open("hello", ios::app);
int p = fileout.tellp();
```

On execution of these statements, the output pointer is moved to the end of the file "hello" and the value of **p** will represent the number of bytes in the file.

Specifying the offset

We have just now seen how to move a file pointer to a desired location using the 'seek' functions. The argument to these functions represents the absolute position in the file. This is shown in Fig. 11.8.

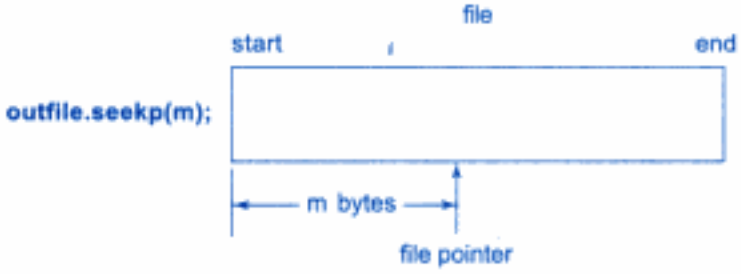


Fig. 11.8 ⇔ *Action of single argument seek function*

'Seek' functions **seekg()** and **seekp()** can also be used with two arguments as follows:

```
seekg (offset, refposition);
seekp (offset, refposition);
```

The parameter *offset* represents the number of bytes the file pointer is to be moved from the location specified by the parameter *refposition*. The *refposition* takes one of the following three constants defined in the **ios** class:

- **ios::beg** start of the file
- **ios::cur** current position of the pointer
- **ios::end** End of the file

The **seekg()** function moves the associated file's 'get' pointer while the **seekp()** function moves the associated file's 'put' pointer. Table 11.3 lists some sample pointer offset calls and their actions. **fout** is an **ofstream** object.

Table 11.3 *Pointer offset calls*

Seek call	*Action*
fout.seekg(o, ios::beg);	Go to start
fout.seekg(o, ios::cur);	Stay at the current position
fout.seekg(o, ios::end);	Go to the end of file
Fout.seekg(m,ios::beg);	Move to (m + 1)th byte in the file
fout.seekg(m,ios::cur);	Go forward by m byte form the current position
fout.seekg(-m,ios::cur);	Go backward by m bytes from the current position
fout.seekg(-m,ios::end);	Go backward by m bytes form the end

11.7 Sequential Input and Output Operations

The file stream classes support a number of member functions for performing the input and output operations on files. One pair of functions, **put()** and **get()**, are designed for handling a single character at a time. Another pair of functions, **write()** and **read()**, are designed to write and read blocks of *binary* data.

put() and get() Functions

The function **put()** writes a single character to the associated stream. Similarly, the function **get()** reads a single character from the associated stream. Program 11.4 illustrates how these functions work on a file. The program requests for a string. On receiving the string, the program writes it, character by character, to the file using the **put()** function in a **for** loop. Note that the length of the string is used to terminate the **for** loop.

The program then displays the contents of the file on the screen. It uses the function **get()** to fetch a character from the file and continues to do so until the end-of-file condition is reached. The character read from the file is displayed on the screen using the operator <<.

I/O OPERATIONS ON CHARACTERS

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

int main()
{
    char string[80];

    cout << "Enter a string \n";
    cin >> string;
```

(Contd)

Hidden page

Hidden page

```
        for(int i=0; i<4; i++)        // clear array from memory
                height[i] = 0;

        ifstream infile;
        infile.open(filename);

        infile.read((char *) & height, sizeof(height));

        for(i=0; i<4; i++)
        {
                cout.setf(ios::showpoint);
                cout << setw(10) << setprecision(2)
                     << height[i];
        }
        infile.close();

        return 0;
}
```

PROGRAM 11.5

The output of Program 11.5 would be:

```
175.50   153.00   167.25
```

Reading and Writing a Class Object

We mentioned earlier that one of the shortcomings of the I/O system of C is that it cannot handle user-defined data types such as class objects. Since the class objects are the central elements of C++ programming, it is quite natural that the language supports features for writing to and reading from the disk files objects directly. The binary input and output functions **read()** and **write()** are designed to do exactly this job. These functions handle the entire structure of an object as a single unit, using the computer's internal representation of data. For instance, the function **write()** copies a class object from memory byte by byte with no conversion. One important point to remember is that only data members are written to the disk file and the member functions are not.

Program 11.6 illustrates how class objects can be written to and read from the disk files. The length of the object is obtained using the **sizeof** operator. This length represents the sum total of lengths of all data members of the object.

READING AND WRITING CLASS OBJECTS

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

(Contd)

```
class INVENTORY
{
        char  name[10];                    // item name
        int   code;                        // item code
        float cost;                        // cost of each item
  public:
        void readdata(void);
        void writedata(void);
};

void INVENTORY :: readdata(void)           // read from keyboard
{
        cout << "Enter name: "; cin >> name;
        cout << "Enter code: "; cin >> code;
        cout << "Enter cost: "; cin >> cost;
}

void INVENTORY :: writedata(void)           // formatted display on
{                                          // screen
        cout << setiosflags(ios::left)
             << setw(10) << name
             << setiosflags(ios::right)
             << setw(10) << code
             << setprecision(2)
             << setw(10) << cost
             << endl;
}

int main()
{
        INVENTORY item[3];                 // Declare array of 3 objects

        fstream file;                      // Input and output file

        file.open("STOCK.DAT", ios::in | ios::out);

        cout << "ENTER DETAILS FOR THREE ITEMS \n";
        for(int i=0;i<3;i++)
        {
              item[i].readdata();

              file.write((char *) & item[i],sizeof(item[i]));
        }
```

(Contd)

```
        file.seekg(0);                          // reset to start

        cout << "\nOUTPUT\n\n";
        for(i = 0;  i < 3; i++)
        {
              file.read((char *) & item[i], sizeof(item[i]));
              item[i].writedata();
        }
        file.close();
        return 0;
}
```

PROGRAM 11.6

The output of Program 11.6 would be:

```
ENTER DETAILS FOR THREE ITEMS
Enter name: C++
Enter code: 101
Enter cost: 175
Enter name: FORTRAN
Enter code: 102
Enter cost: 150
Enter name: JAVA
Enter code: 115
Enter cost: 225

OUTPUT

C++             101   175
FORTRAN         102   150
JAVA            115   225
```

The program uses 'for' loop for reading and writing objects. This is possible because we know the exact number of objects in the file. In case, the length of the file is not known, we can determine the file-size in terms of objects with the help of the file pointer functions and use it in the 'for' loop or we may use **while(file)** test approach to decide the end of the file. These techniques are discussed in the next section.

11.8 Updating a File: Random Acess

Updating is a routine task in the maintenance of any data file. The updating would include one or more of the following tasks:

- Displaying the contents of a file.

Hidden page

Hidden page

```
cout << "CONTENTS OF APPENDED FILE \n";

while(inoutfile.read((char *) & item, sizeof item))
{
       item.putdata();
}


// Find number of objects in the file
int last = inoutfile.tellg();
int n = last/sizeof(item);

cout << "Number of objects = " << n << "\n";
cout << "Total bytes in the file = " << last << "\n";

/* >>>>>>>>> MODIFY THE DETAILS OF AN ITEM <<<<<<<<< */

cout << "Enter object number to be updated \n";
int object;
cin >> object;

cin.get(ch);

int location = (object-1) * sizeof(item);

if(inoutfile.eof())
inoutfile.clear();

inoutfile.seekp(location);

cout << "Enter new values of the object \n";
item.getdata();
cin.get(ch);

inoutfile.write((char *) & item, sizeof item) << flush;

/* >>>>>>>>>>>>> SHOW UPDATED FILE <<<<<<<<<<<<< */

inoutfile.seekg(0); //go to the start

cout << "CONTENTS OF UPDATED FILE \n";

while(inoutfile.read((char *) & item, sizeof item))
{
```

(Contd)

Hidden page

Hidden page

5. We may use an invalid file name.
6. We may attempt to perform an operation when the file is not opened for that purpose.

The C++ file stream inherits a 'stream-state' member from the class ios. This member records information on the status of a file that is being currently used. The stream state member uses bit fields to store the status of the error conditions stated above.

The class **ios** supports several member functions that can be used to read the status recorded in a file stream. These functions along with their meanings are listed in Table 11.4.

Table 11.4 *Error handling functions*

Function	*Return value and meaning*
eof()	Returns *true* (non-zero value) if end-of-file is encountered while reading; Otherwise returns *false*(zero)
fail()	Returns *true* when an input or output operation has failed
bad()	Returns *true* if an invalid operation is attempted or any unrecoverable error has occurred. However, if it is *false*, it may be possible to recover from any other error reported, and continue operation.
good()	Returns true if no error has occurred. This means, all the above functions are false. For instance, if **file.good()** is *true*, all is well with the stream **file** and we can proceed to perform I/O operations. When it returns *false*, no further operations can be carried out.

These functions may be used in the appropriate places in a program to locate the status of a file stream and thereby to take the necessary corrective measures. Example:

```
.....
.....
ifstream infile;
infile.open("ABC");
while(!infile.fail())
{
.....
.....      (process the file)
.....
}
if(infile.eof())
{
    .....     (terminate program normally)
}
else
```

Hidden page

The command-line arguments are typed by the user and are delimited by a space. The first argument is always the filename (command name) and contains the program to be executed. How do these arguments get into the program?

The **main()** functions which we have been using up to now without any arguments can take two arguments as shown below:

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

The first argument **argc** (known as *argument counter*) represents the number of arguments in the command line. The second argument **argv** (known as *argument vector*) is an array of **char** type pointers that points to the command line arguments. The size of this array will be equal to the value of **argc**. For instance, for the command line

```
C > exam data results
```

the value of **argc** would be 3 and the **argv** would be an array of three pointers to strings as shown below:

```
argv[0] ---> exam
argv[1] ---> data
argv[2] ---> results
```

Note that **argv[0]** always represents the command name that invokes the program. The character pointers **argv[1]** and **argv[2]** can be used as file names in the file opening statements as shown below:

```
.....
.....
infile.open(argv[1]);   // open data file for reading
.....
.....
outfile.open(argv[2]);  // open results file for writing
.....
.....
```

Program 11.8 illustrates the use of the command-line arguments for supplying the file names. The command line is

```
test ODD EVEN
```

The program creates two files called **ODD** and **EVEN** using the command-line arguments, and a set of numbers stored in an array are written to these files. Note that the odd numbers are written to the file **ODD** and the even numbers are written to the file **EVEN**. The program then displays the contents of the files.

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

- The C++ **I/O** system contains classes such as **ifstream, ofstream** and **fstream** to deal with file handling. These classes are derived from **fstreambase** class and are declared in a header file *iostream*.
- A file can be opened in two ways by using the constructor function of the class and using the member function **open()** of the class.
- While opening the file using constructor, we need to pass the desired filename as a parameter to the constructor.
- The **open()** function can be used to open multiple files that use the same stream object. The second argument of the **open()** function called file mode, specifies the purpose for which the file is opened.

- If we do not specify the second argument of the **open()** function, the default values specified in the prototype of these class member functions are used while opening the file. The default values are as follows:

```
ios :: in – for ifstream functions, meaning-open for reading only.
ios :: out – for ofstream functions, meaning-open for writing only.
```

- When a file is opened for writing only, a new file is created only if there is no file of that name. If a file by that name already exists, then its contents are deleted and the file is presented as a clean file.
- To open an existing file for updating without losing its original contents, we need to open it in an append mode.
- The **fstream** class does not provide a mode by default and therefore we must provide the mode explicitly when using an object of **fstream** class. We can specify more than one file modes using bitwise OR operator while opening a file.
- Each file has associated two file pointers, one is called input or get pointer, while the other is called output or put pointer. These pointers can be moved along the files by member functions.
- Functions supported by file stream classes for performing I/O operations on files are as follows:

```
put() and get() functions handle single character at a time.
write() and read() functions write and read blocks of binary data.
```

- The class **ios** supports many member functions for managing errors that may occur during file operations.
- File names may be supplied as arguments to the **main()** function at the time of invoking the program. These arguments are known as command-line arguments.

Key Terms

- append mode
- argc
- argument counter
- argument vector
- argv
- bad()
- binary data
- binary format
- character format
- clear()
- command-line
- end-of-file
- eof()
- fail()

(Contd)

- file mode
- file mode parameters
- file pointer
- file stream classes
- file streams
- **filebuf**
- files
- **fstream**
- **fstreambase**
- get pointer
- **get()**
- **good()**
- **ifstream**
- input pointer
- input stream
- **ios**
- **ios::app**
- **ios::ate**
- **ios::beg**
- **ios::binary**
- **ios::cur**
- **ios::end**
- **ios::in**
- **ios::nocreate**
- ios::out
- **ios::noreplace**
- **ios::trunc**
- **iostream**
- **ofstream**
- **open()**
- output pointer
- output stream
- put pointer
- **put()**
- random access
- **read()**
- **seekg()**
- **seekp()**
- **sizeof()**
- streams
- **tellg()**
- **tellp()**
- updating
- **write()**

Review Questions

11.1 *What are input and output streams?*

11.2 *What are the steps involved in using a file in a C++ program?*

11.3 *Describe the various classes available for file operations.*

11.4 *What is the difference between opening a file with a constructor function and opening a file with* ***open()*** *function? When is one method preferred over the other?*

11.5 *Explain how* ***while(fin)*** *statement detects the end of a file that is connected to fin stream*

11.6 *What is a file mode? Describe the various file mode options available.*

11.7 *Write a statement that will create an object called* ***fob*** *for writing, and associate it with a file name DATA.*

11.8 *How many file objects would you need to create to manage the following situations?*
(a) *To process four files sequentially.*
(b) *To merge two sorted files into a third file.*
Explain.

11.9 *Both* ***ios::ate*** *and* ***ios::app*** *place the file pointer at the end of the file (when it is opened). What then, is the difference between them?*

11.10 *What does the "current position" mean when applied to files?*

11.11 *Write statements using* ***seekg()*** *to achieve the following:*
(a) *To move the pointer by 15 positions backward from current position.*
(b) *To go to the beginning after an operation is over.*
(c) *To go backward by 20 bytes from the end.*
(d) *To go to byte number 50 in the file.*

11.12 *What are the advantages of saving data in binary form?*

11.13 *Describe how would you determine number of objects in a file. When do you need such information?*

11.14 *Describe the various approaches by which we can detect the end-of-file condition successfully.*

11.15 *State whether the following statements are* ***TRUE*** *or* ***FALSE****.*
(a) *A stream may be connected to more than one file at a time.*
(b) *A file pointer always contains the address of the file.*
(c) *The statement*
`outfile.write((char *) & obj,sizeof(obj));`
writes only data in **`obj`** *to* **`outfile`****.**
(d) *The* ***ios::ate*** *mode allows us to write data anywhere in the file.*
(e) *We can add data to an existing file by opening in write mode.*
(f) *The parameter* ***ios::app*** *can be used only with the files capable of output.*
(g) *The data written to a file with* ***write()*** *function can be read with the get() function.*
(h) *We can use the functions* ***tellp()*** *and* ***tellg()*** *interchangeably for any file.*
(i) *Binary files store floating point values more accurately and compactly than the text files.*
(j) *The* ***fin.fail()*** *call returns non-zero when an operation on the file has failed.*

Debugging Exercises

11.1 Identify the error in the following program.

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

void main()
```

Hidden page

```
        while(!in.getline(buffer, 80))
        {
             cout << buffer << endl;
        }

        while(!in.getline(buffer, 80).eof())
        {
             cout << buffer << endl;
        }
}
```

11.4 Find errors in the following statements.

(a) `ifstream.infile("DATA");`
(b) `fin1.getline(); //fin1 is input stream`
(c) `if(fin1.eof() == 0) exit(1);`
(d) `close(f1);`
(e) `infile.open(argc);`
(f) `sfinout.open(file,ios::in |ios::out| ios::ate);`

Programming Exercises

11.1 *Write a program that reads a text file and creates another file that is identical except that every sequence of consecutive blank spaces is replaced by a single space.*

11.2 *A file contains a list of telephone numbers in the following form::*

John 23456
Ahmed 9876
....

The names contain only one word and the names and telephone numbers are separated by white spaces. Write a program to read the file and output the list in two columns. The names should be left-justified and the numbers right-justified.

11.3 *Write a program that will create a data file containing the list of telephone numbers given in Exercise 11.2. Use a class object to store each set of data.*

11.4 *Write an interactive, menu-driven program that will access the file created in Exercise 11.3 and implement the following tasks.*

(a) *Determine the telephone number of the specified person.*
(b) *Determine the name if a telephone number is known.*
(c) *Update the telephone number, whenever there is a change.*

12

Templates

Key Concepts

- Generic programming
- Multiple parameters in class templates
- Function templates
- Template functions
- Member function templates
- Class templates
- Template classes
- Multiple parameters in class templates
- Overloading of template functions
- Non-type template arguments

12.1 Introduction

Templates is one of the features added to C++ recently. It is a new concept which enable us to define generic classes and functions and thus provides support for *generic programming*. Generic programming is an approach where generic types are used as parameters in algorithms so that they work for a variety of suitable data types and data structures.

A template can be used to create a family of classes or functions. For example, a class template for an **array** class would enable us to create arrays of various data types such as **int** array and **float** array. Similarly, we can define a template for a function, say **mul()**, that would help us create various versions of **mul()** for multiplying **int**, **float** and **double** type values.

A template can be considered as a kind of macro. When an object of a specific type is defined for actual use, the template definition for that class is substituted with the required data type. Since a template is defined with a *parameter* that would be replaced by a specified data type at the time of actual use of the class or function, the templates are sometimes called *parameterized classes* or *functions*.

12.2 Class Templates

Consider a vector class defined as follows:

```
class vector
{
       int *v;
       int size;
    public:
      vector(int m)              // create a null vector
      {
           v = new int[size = m];
           for(int i=0; i<size; i++)
                v[i] = 0;
      }
      vector(int *a)             // create a vector from an array
      {
           for(int i=0; i<size; i++)
               v[i] = a[i];
      }
      int operator*(vector &y)        // scalar product
      {
      int sum = 0;
      for(int i=0; i<size; i++)
           sum += this -> v[i] * y . v[i];
      return sum;
      }
};
```

The vector class can store an array of **int** numbers and perform the scalar product of two **int** vectors as shown below:

```
int main()
{
     int x[3] = {1,2,3};
     int y[3] = {4,5,6};
     vector v1(3);               // Creates a null vector of 3 integers
     vector v2(3);
     v1 = x;                     // Creates v1 from the array x
     v2 = y;
     int R = v1 * v2;
     cout << "R = " << R;
     return 0;
}
```

Now suppose we want to define a vector that can store an array of **float** values. We can do this by simply replacing the appropriate **int** declarations with **float** in the **vector** class. This means that we have to redefine the entire class all over again.

Assume that we want to define a **vector** class with the data type as a *parameter* and then use this class to create a vector of any data type instead of defining a new class every time. The template mechanism enables us to achieve this goal.

As mentioned earlier, templates allow us to define generic classes. It is a simple process to create a generic class using a template with an anonymous type. The general format of a class template is:

```
template<class T>
class classname
{
      // ......
      // class member specification
      // with anonymous type T
      // wherever appropriate
      // ......
};
```

The template definition of **vector** class shown below illustrates the syntax of a template:

```
template<class T>
class vector
{
        T* v;          // Type T vector
        int size;
  public:
      vector(int m)
      {
           v = new T [size = m];
           for(int i=0; i<size; i++)
             v[i] = 0;
      }
      vector(T* a)
      {
            for(int i=0; i<size, i++)
                 v[i] = a[i];
       }
       T operator*(vector &y)
       {
            T sum = 0;
              for(int i=0; i<size; i++)
                          sum += this -> v[i] * y . v[i];
            return sum;
       }
   };
```

note

The class template definition is very similar to an ordinary class definition except the prefix **template<class T>** and the use of type **T**. This prefix tells the compiler that we are going to declare a template and use **T** as a type name in the declaration. Thus, vector has become a parameterized class with the type **T** as its parameter. **T** may be substituted by any data type including the user-defined types. Now, we can create vectors for holding different data types.

Example:

```
vector <int>   v1(10);        // 10 element int vector
vector <float> v2(25);        // 25 element float vector
```

note

The type **T** may represent a class name as well. Example:

```
vector <complex> v3(5);     // vector of 5 complex numbers
```

A class created from a class template is called a *template class*. The syntax for defining an object of a template class is:

```
classname<type> objectname(arglist);
```

This process of creating a specific class from a class template is called *instantiation*. The compiler will perform the error analysis only when an instantiation takes place. It is, therefore, advisable to create and debug an ordinary class before converting it into a template.

Programs 12.1 and 12.2 illustrate the use of a **vector** class template for performing the scalar product of **int** type vectors as well as **float** type vectors.

Example of Class Template

```
#include <iostream>

using namespace std;

const size = 3;

template <class T>
class vector
{
      T* v;          // type T vector
public:
      vector()
      {
```

(Contd)

```
            v = new T[size];
            for(int i=0;i<size;i++)
                v[i] = 0;
        }
        vector(T* a)
        {
            for(int i=0;i<size;i++)
                v[i] = a[i];
        }
        T operator*(vector &y)
        {
            T sum = 0;
            for(int i=0;i<size;i++)
                sum += this -> v[i] * y.v[i];
            return sum;
        }
};
int main()
{
        int x[3] = {1,2,3};
        int y[3] = {4,5,6};
        vector <int> v1;
        vector <int> v2;
        v1 = x;
        v2 = y;
        int R = v1 * v2;
        cout << "R = " << R << "\n";
        return 0;
}
```

PROGRAM 12.1

The output of the Program 12.1 would be:

```
R = 32
```

ANOTHER EXAMPLE OF CLASS TEMPLATE

```
#include <iostream>

using namespace std;

const size = 3;
template <class T>
```

(Contd)

```
class vector
{
        T* v;           // type T vector
  public:
        vector()
        {
                v = new T[size];
                for(int i=0;i<size;i++)
                        v[i] = 0;
        }
        vector(T* a)
        {
                for(int i=0;i<size;i++)
                        v[i] = a[i];
        }
        T operator*(vector &y)
        {
                T sum = 0;
                for(int i=0;i<size;i++)
                        sum += this -> v[i] * y.v[i];

                return sum;
        }
};

int main()
{
        float x[3] = {1.1,2.2,3.3};
        float y[3] = {4.4,5.5,6.6};
        vector <float> v1;
        vector <float> v2;
        v1 = x;
        v2 = y;
        float R = v1 * v2;
        cout << "R = " << R << "\n";

        return 0;
}
```

PROGRAM 12.2

The output of the Program 12.2 would be:

```
R = 38.720001
```

12.3 Class Templates with Multiple Parameters

We can use more than one generic data type in a class template. They are declared as a comma-separated list within the **template** specification as shown below:

```
template<class T1, class T2, ...>
class classname
{
  .....
  .....     (Body of the class)
  .....
};
```

Program 12.3 demonstrates the use of a template class with two generic data types.

TWO GENERIC DATA TYPES IN A CLASS DEFINITION

```
#include <iostream>

using namespace std;

template<class T1, class T2>
class Test
{
        T1 a;
        T2 b;
  public:
        Test(T1 x, T2 y)
        {
                a = x;
                b = y;
        }
        void show()
        {
                cout << a << " and " << b << "\n";
        }
};

int main()
{
        Test <float,int> test1 (1.23,123);
        Test <int,char> test2 (100,'W');

        test1.show();
        test2.show();

        return 0;
};
```

PROGRAM 12.3

The output of Program 12.3 will be would be:

```
1.23 and 123
100 and W
```

12.4 Function Templates

Like class templates, we can also define function templates that could be used to create a family of functions with different argument types. The general format of a function template is:

```
template<class T>
returntype functionname (arguments of type T)
{
        // .....
        // Body of function
        // with type T
        // wherever appropriate
        // .....
}
```

The function template syntax is similar to that of the class template except that we are defining functions instead of classes. We must use the template parameter **T** as and when necessary in the function body and in its argument list.

The following example declares a **swap()** function template that will swap two values of a given type of data.

```
template<class T>
void swap(T&x, T&y)
{
   T temp = x;
   x = y;
   y = temp;
}
```

This essentially declares a set of overloaded functions, one for each type of data. We can invoke the **swap()** function like any ordinary function. For example, we can apply the **swap()** function as follows:

```
void f(int m,int n,float a,float b)
{
     swap(m,n);          // swap two integer values
     swap(a,b);          // swap two float values
     // .....
```

Hidden page

Hidden page

Hidden page

AN APPLICATION OF TEMPLATE FUNCTIONS

```
#include <iostream>
#include <iomanip>
#include <cmath>

using namespace std;

template <class T>
void roots(T a,T b,T c)
{
        T d = b*b - 4*a*c;
        if(d == 0)                      // Roots are equal
        {
                cout << "R1 = R2 = " << -b/(2*a) << endl;
        }
        else if(d > 0)                  // Two real roots
        {
                cout << "Roots are real \n";
                float R = sqrt(d);
                float R1 = (-b+R)/(2*a);
                float R2 = (-b-R)/(2*a);
                cout << "R1 = " << R1 << " and ";
                cout << "R2 = " << R2 << endl;
        }
        else                            // Roots are complex
        {
                cout << "Roots are complex \n";
                float R1 = -b/(2*a);
                float R2 = sqrt(-d)/(2*a);
                cout << "Real part = " << R1 << endl;
                cout << "Imaginary part = " << R2;
                cout << endl;
        }
}

int main()
{
        cout << "Integer coefficients \n";
        roots(1,-5,6);
        cout << "\nFloat coefficients \n";
        roots(1.5,3.6,5.0);

        return 0;
}
```

PROGRAM 12.6

The output of Program 12.6 would be:

```
Integer coefficients
Roots are real

R1 = 3 and R2 = 2

Float coefficients
Roots are complex
Real part = -1.2
Imaginary part = 1.375985
```

12.5 Function Templates with Multiple Parameters

Like template classes, we can use more than one generic data type in the template statement, using a comma-separated list as shown below:

```
template<class T1, class T2, ...>
returntype functionname(arguments of types T1, T2,...)
{
        .....
        .....   (Body of function)
        .....
}
```

Program 12.7 illustrates the concept of using two generic types in template functions.

FUNCTION WITH TWO GENERIC TYPES

```
#include <iostream>
#include <string>

using namespace std;

template<class T1, class T2>
void display(T1 x, T2 y)
{
    cout << x << " " << y << "\n";
}

int main()
{
    display(1999, "EBG");
    display(12.34, 1234);
    return 0;
}
```

PROGRAM 12.7

The output of Program 12.7 would be:

```
1999 EBG
12.34 1234
```

12.6 Overloading of Template Functions

A template function may be overloaded either by template functions or ordinary functions of its name. In such cases, the overloading resolution is accomplished as follows:

1. Call an ordinary function that has an exact match.
2. Call a template function that could be created with an exact match.
3. Try normal overloading resolution to ordinary functions and call the one that matches.

An error is generated if no match is found. Note that no automatic conversions are applied to arguments on the template functions. Program 12.8 shows how a template function is overloaded with an explicit function.

TEMPLATE FUNCTION WITH EXPLICIT FUNCTION

```
#include <iostream>
#include <string>

using namespace std;

template <class T>
void display(T x)
{
     cout << "Template display: " << x << "\n";
}
void display(int x)        // overloads the generic display()
{
     cout << "Explicit display: " << x << "\n";
}

int main()
{
     display(100);
     display(12.34);
     display('C');

     return 0;
}
```

PROGRAM 12.8

Hidden page

```
vector<T> :: vector(int m)
{
      v = new T[size = m];
      for(int i=0; i<size; i++)
          v[i] = 0;
}

template< class T>
vector<T> :: vector(T* a)
{
      for(int i=0; i<size; i++)
           v[i] = a[i];
}

template< class T>
T vector<T> :: operator*(vector & y)
{
      T sum = 0;
      for(int i = 0; i < size; i++)
      sum += this -> v[i] * y.v[i];
      return sum;
}
```

12.8 Non-Type Template Arguments

We have seen that a template can have multiple arguments. It is also possible to use non-type arguments. That is, in addition to the type argument **T**, we can also use other arguments such as strings, function names, constant expressions and built-in types. Consider the following example:

```
template<class T, int size>
class array
{
      T a[size];          // automatic array initialization
      // .....
      // .....
};
```

This template supplies the size of the **array** as an argument. This implies that the size of the **array** is known to the compiler at the compile time itself. The arguments must be specified whenever a template class is created. Example:

```
array<int,10>  a1;      // Array of 10 integers
array<float,5> a2;      // Array of 5 floats
array<char,20> a3;      // String of size 20
```

The size is given as an argument to the template class.

SUMMARY

- C++ supports a mechanism known as template to implement the concept of generic programming.
- Templates allows us to generate a family of classes or a family of functions to handle different data types.
- Template classes and functions eliminate code duplication for different types and thus make the program development easier and more manageable.
- We can use multiple parameters in both the class templates and function templates.
- A specific class created from a class template is called a template class and the process of creating a template class is known as instantiation. Similarly, a specific function created from a function template is called a template function.
- Like other functions, template functions can be overloaded.
- Member functions of a class template must be defined as function templates using the parameters of the class template.
- We may also use non-type parameters such basic or derived data types as arguments templates.

Key Terms

- bubble sort
- class template
- **display()**
- **explicit** function
- function template
- generic programming
- instantiation
- member function template
- multiple parameters
- overloading
- **parameter**
- parameterized classes
- parameterized functions
- swapping
- **swap()**
- **template**
- template class
- template definition
- template function
- template parameter
- template specification
- templates

Review Questions

12.1 *What is generic programming? How is it implemented in C++?*

12.2 *A template can be considered as a kind of macro. Then, what is the difference between them?*

12.3 *Distinguish between overloaded functions and function templates.*

12.4 *Distinguish between the terms class template and template class.*

12.5 *A class (or function) template is known as a parameterized class (or function). Comment.*

12.6 *State which of the following definitions are illegal.*

(a)
```
template<class T>
class city
{ ..... };
```

(b)
```
template<class P, R, class S>
class city
{ ..... }
```

(c)
```
template<class T, typename S>
class city
{ ..... };
```

(d)
```
template<class T, typename S>
class city
{ ..... };
```

(e)
```
class<class T, int size=10>
class list
{ ..... };
```

(f)
```
class<class T = int, int size>
class list
{ ..... };
```

12.7 *Identify which of the following function template definitions are illegal.*

(a)
```
template<class A, B>
void fun(A, B)
{ ..... };
```

(b)
```
template<class A, class A>
void fun(A, A)
{ ..... };
```

(c)
```
template<class A>
void fun(A, A)
{ ..... };
```

```
(d)  template<class T, typename R>
     T fun(T, R)
     { ..... };
(e)  template<class A>
     A fun(int *A)
     { ..... };
```

Debugging Exercises

12.1 Identify the error in the following program.

```
#include <iostream.h>
class Test
{
        int intNumber;
        float floatNumber;
public:
        Test()
        {
                intNumber = 0;
                floatNumber = 0.0;
        }

        int getNumber()
        {
                return intNumber;
        }

        float getNumber()
        {
                return floatNumber;
        }
};

void main()
{
        Test objTest1;
        objTest1.getNumber();
}
```

12.2 Identify the error in the following program.

```
#include <iostream.h>

template <class T1, class T2>
```

Hidden page

Hidden page

13

Exception Handling

Key Concepts

- Errors and exceptions
- Throwing mechanism
- Multiple catching
- Rethrowing exceptions
- Exception handling mechanism
- Catching mechanism
- Catching all exceptions
- Restricting exceptions thrown

13.1 Introduction

We know that it is very rare that a program works correctly first time. It might have bugs. The two most common types of bugs are *logic errors* and *syntactic errors*. The logic errors occur due to poor understanding of the problem and solution procedure. The syntactic errors arise due to poor understanding of the language itself. We can detect these errors by using exhaustive debugging and testing procedures.

We often come across some peculiar problems other than logic or syntax errors. They are known as *exceptions*. Exceptions are run time anomalies or unusual conditions that a program may encounter while executing. Anomalies might include conditions such as division by zero, access to an array outside of its bounds, or running out of memory or disk space. When a program encounters an exceptional condition, it is important that it is identified and dealt with effectively. ANSI C++ provides built-in language features to detect and handle exceptions which are basically run time errors.

Exception handling was not part of the original C++. It is a new feature added to ANSI C++. Today, almost all compilers support this feature. C++ exception handling provides a

type-safe, integrated approach, for coping with the unusual predictable problems that arise while executing a program.

13.2 Basics of Exception Handling

Exceptions are of two kinds, namely, *synchronous exceptions* and *asynchronous exceptions*. Errors such as "out-of-range index" and "over-flow" belong to the synchronous type exceptions. The errors that are caused by events beyond the control of the program (such as keyboard interrupts) are called asynchronous exceptions. The proposed exception handling mechanism in C++ is designed to handle only synchronous exceptions.

The purpose of the exception handling mechanism is to provide means to detect and report an "exceptional circumstance" so that appropriate action can be taken. The mechanism suggests a separate error handling code that performs the following tasks:

1. Find the problem (*Hit the exception*).
2. Inform that an error has occurred (*Throw the exception*).
3. Receive the error information (*Catch the exception*).
4. Take corrective actions (*Handle the exception*).

The error handling code basically consists of two segments, one to detect errors and to throw exceptions, and the other to catch the exceptions and to take appropriate actions.

13.3 Exception Handling Mechanism

C++ exception handling mechanism is basically built upon three keywords, namely, **try**, **throw**, and **catch**. The keyword **try** is used to preface a block of statements (surrounded by braces) which may generate exceptions. This block of statements is known as *try block*. When an exception is detected, it is thrown using a **throw** statement in the try block. A *catch block* defined by the keyword **catch** 'catches' the exception 'thrown' by the throw statement in the try block, and handles it appropriately. The relationship is shown in Fig. 13.1.

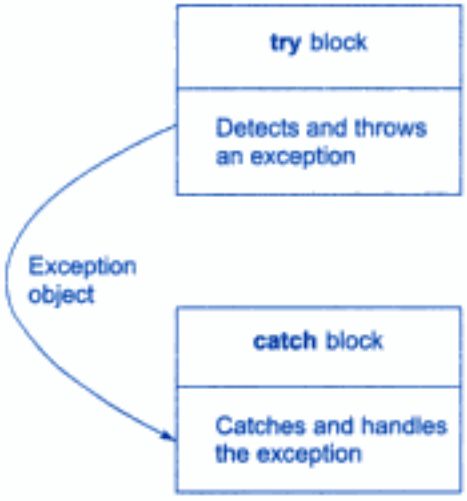


Fig. 13.1 ⇔ *The block throwing exception*

The **catch** block that catches an exception must immediately follow the **try** block that throws the exception. The general form of these two blocks are as follows:

```
.....
.....
try
{
        .....
        throw exception;            // Block of statements which
        .....                       // detects and throws an exception
        .....
}
catch(type arg)                     // Catches exception
{
        .....
        .....                       // Block of statements that
        .....                       // handles the exception
        .....
}
.....
.....
```

When the **try** block throws an exception, the program control leaves the **try** block and enters the **catch** statement of the catch block. Note that exceptions are objects used to transmit information about a problem. If the type of object thrown matches the *arg* type in the **catch** statement, then catch block is executed for handling the exception. If they do not match, the program is aborted with the help of the **abort()** function which is invoked by default. When no exception is detected and thrown, the control goes to the statement immediately after the catch block. That is, the catch block is skipped. This simple try-catch mechanism is illustrated in Program 13.1.

TRY BLOCK THROWING AN EXCEPTION

```
#include <iostream>

using namespace std;
int main()
{
    int a,b;
    cout << "Enter Values of a and b \n";
    cin >> a;
    cin >> b;
    int x = a-b;
    try
    {
        if(x != 0)
        {
```

(Contd)

```
            cout << "Result(a/x) = " << a/x << "\n";
        }
        else                    // There is an exception
        {
                throw(x);   // Throws int object
        }
    }
    catch(int i)                // Catches the exception
    {
            cout << "Exception caught: x = " << x << "\n";
    }

    cout << "END";

    return 0;
}
```

PROGRAM 13.1

The output of Program 13.1:

First Run

```
Enter Values of a and b
20 15
Result(a/x) = 4
END
```

Second Run

```
Enter Values of a and b
10 10
Exception caught: x = 0
END
```

Program detects and catches a division-by-zero problem. The output of first run shows a successful execution. When no exception is thrown, the **catch** block is skipped and execution resumes with the first line after the **catch**. In the second run, the denominator **x** becomes zero and therefore a division-by-zero situation occurs. This exception is thrown using the object **x**. Since the exception object is an **int** type, the **catch** statement containing **int** type argument catches the exception and displays necessary message.

Most often, exceptions are thrown by functions that are invoked from within the **try** blocks. The point at which the **throw** is executed is called the *throw point*. Once an exception is thrown to the catch block, control cannot return to the throw point. This kind of relationship is shown in Fig. 13.2.

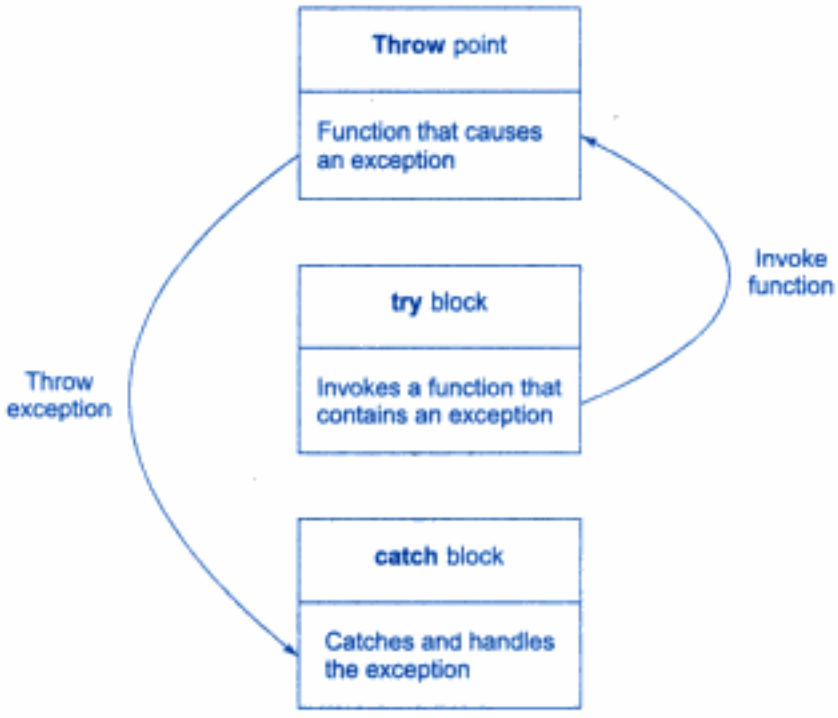


Fig. 13.2 ⇔ *Function invoked by try block throwing exception*

The general format of code for this kind of relationship is shown below:

```
type function(arg list)      // Function with exception
{
        ......
        ......
        throw(object);       // Throws exception
        ......
        ......
}
......
......
try
{
        ......
        ...... Invoke function here
        ......
}
catch(type arg)              // Catches exception
{
        ......
        ...... Handles exception here
        ......
}
......
```

note

The **try** block is immediately followed by the **catch** block, irrespective of the location of the throw point.

Program 13.2 demonstrates how a **try** block invokes a function that generates an exception.

INVOKING FUNCTION THAT GENERATES EXCEPTION

```
// Throw point outside the try block

#include <iostream>

using namespace std;

void divide(int x, int y, int z)
{
        cout << "\nWe are inside the function \n";
        if((x-y) != 0)          // It is OK
        {
              int R = z/(x-y);
              cout << "Result = " << R << "\n";
        }
        else                    // There is a problem
        {
              throw(x-y);       // Throw point
        }
}

int main()
{
        try
        {
              cout << "We are inside the try block \n";
              divide(10,20,30);   // Invoke divide()
              divide(10,10,20);   // Invoke divide()
        }
        catch(int i)      // Catches the exception
        {
              cout << "Caught the exception \n";
        }
        return 0;
}
```

PROGRAM 13.2

Hidden page

braces. The **catch** statement catches an exception whose type matches with the type of **catch** argument. When it is caught, the code in the **catch** block is executed.

If the parameter in the **catch** statement is named, then the parameter can be used in the exception-handling code. After executing the handler, the control goes to the statement immediately following the catch block.

Due to mismatch, if an exception is not caught, abnormal program termination will occur. It is important to note that the **catch** block is simply skipped if the **catch** statement does not catch an exception.

Multiple Catch Statements

It is possible that a program segment has more than one condition to throw an exception. In such cases, we can associate more than one **catch** statement with a **try** (much like the conditions in a **switch** statement) as shown below:

```
try
{
     // try block
}
catch(type1 arg)
{
     // catch block1
}
catch(type2 arg)
{
     // catch block2
}
......
......
catch(typeN arg)
{
     // catch blockN
}
```

When an exception is thrown, the exception handlers are searched *in order* for an appropriate match. The first handler that yields a match is executed. After executing the handler, the control goes to the first statement after the last **catch** block for that **try**. (In other words, all other handlers are bypassed). When no match is found, the program is terminated.

It is possible that arguments of several **catch** statements match the type of an exception. In such cases, the first handler that matches the exception type is executed.

Hidden page

Hidden page

Hidden page

note

Remember, **catch(...)** should always be placed last in the list of handlers. Placing it before other **catch** blocks would prevent those blocks from catching exceptions.

13.6 Rethrowing an Exception

A handler may decide to rethrow the exception caught without processing it. In such situations, we may simply invoke **throw** without any arguments as shown below:

```
throw;
```

This causes the current exception to be thrown to the next enclosing **try/catch** sequence and is caught by a **catch** statement listed after that enclosing **try** block. Program 13.5 demonstrates how an exception is rethrown and caught.

RETHROWING AN EXCEPTION

```
#include <iostream>

using namespace std;

void divide(double x, double y)
{
        cout << "Inside function \n";
        try
        {
                if(y == 0.0)
                        throw y;        // Throwing double
                else
                        cout << "Division = " << x/y << "\n";
        }
        catch(double)                   // Catch a double
        {
                cout << "Caught double inside function \n";
                throw;                  // Rethrowing double
        }
        cout << "End of function \n\n";
}

int main()
{
        cout << "Inside main \n";
```

(Contd)

```
        try
        {
                divide(10.5,2.0);
                divide(20.0,0.0);
        }
        catch(double)
        {
                cout << "Caught double inside main \n";
        }
        cout << "End of main \n";

        return 0;
}
```

PROGRAM 13.5

The output of the Program 13.5:

```
Inside main
Inside function
Division = 5.25
End of function

Inside function
Caught double inside function
Caught double inside main
End of main
```

When an exception is rethrown, it will not be caught by the same **catch** statement or any other **catch** in that group. Rather, it will be caught by an appropriate **catch** in the outer **try/catch** sequence only.

A **catch** handler itself may detect and throw an exception. Here again, the exception thrown will not be caught by any **catch** statements in that group. It will be passed on to the next outer **try/catch** sequence for processing.

13.7 Specifying Exceptions

It is possible to restrict a function to throw only certain specified exceptions. This is achieved by adding a **throw** *list* clause to the function definition. The general form of using an *exception specification* is:

```
type function(arg-list) throw (type-list)
{
        ......
        ...... Function body
        ......
}
```

Hidden page

Hidden page

- A **try** block may throw an exception directly or invoke a function that throws an exception. Irrespective of location of the throw point, the catch block is placed immediately after the try block.
- We can place two or more catch blocks together to catch and handle multiple types of exceptions thrown by a try block.
- It is also possible to make a catch statement to catch all types of exceptions using ellipses as its argument.
- We may also restrict a function to throw only a set of specified exceptions by adding a throw specification clause to the function definition.

Key Terms

- **abort()** function
- asynchronous exceptions
- bugs
- **catch** block
- **catch(...)** statement
- catching mechanism
- errors
- exception handler
- exception handling mechanism
- exception specifying
- exceptions
- logic errors
- multiple catch
- out-of-range index
- overflow
- rethrowing exceptions
- synchronous exceptions
- syntactic errors
- **throw**
- throw point
- **throw** statement
- **throw()**
- throwing mechanism
- **try** block

Review Questions

13.1 *What is an exception?*
13.2 *How is an exception handled in C++?*
13.3 *What are the advantages of using exception handling mechanism in a program?*
13.4 *When should a program throw an exception?*
13.5 *When is a **catch(...)** handler is used?*
13.6 *What is an exception specification? When is it used?*
13.7 *What should be placed inside a **try** block?*
13.8 *What should be placed inside a **catch** block?*
13.9 *When do we used multiple **catch** handlers?*

Hidden page

```
                    break;
                case 30:
                    throw "Employee";
                    break;
                }
            }

        }
        void operator ++()
        {
            age+=10;
        }
};
void main()
{
        Person objPerson(10);
        objPerson.getOccupation();
        ++objPerson;
        objPerson.getOccupation();
        ++objPerson;
        objPerson.getOccupation();
}
```

13.2 Identify the error in the following program.

```
#include <iostream.h>

void callFunction(int i)
{
      if(i)
                  throw 1;
      else
                  throw 0;
}

void callFunction(char *n)
{
        try
        {
                  if(n)
                        throw "StringOK";
```

```
                    else
                        throw "StringError";
        }
        catch(char* name)
        {
                    cout << name << " ";
        }
}

void main()
{
        try
        {
            callFunction("testString");
            callFunction(1);
            callFunction(0);
        }
        catch(int i)
        {
            cout << i << " " ;
        }
        catch(char *name)
        {
            cout << name << " ";
        }
}
```

13.3 Identify the error in the following program.

```
#include <iostream.h>

class Mammal
{
public:
        Mammal()
        {
        }

        class Human
        {
```

```
        };

        class Student  : virtual public Human
        {
        };

        class Employee : virtual public Human
        {
        };

        void getObject()
        {
             throw Employee();
        }
};
void main()
{
        Mammal m;
        try
        {
             m.getObject();
        }
        catch(Mammal::Human&)
        {
             cout << "Human ";
        }
        catch(Mammal::Student&)
        {
             cout << "Student ";
        }
        catch(Mammal::Employee&)
        {
             cout << "Employee ";
        }
        catch(...)
        {
             cout << "All";
        }
}
```

13.4 Identify errors, if any, in the following statements.

(a)
```
catch(int a, float b)
{...}
```
(b)
```
try
{throw 100;};
```
(c)
```
try
{fun1()}
```
(d)
```
throw a, b;
```
(e)
```
void divide(int a, int b) throw(x, y)
{......}
```
(f)
```
catch(int x, ..., float y)
{......}
```
(g)
```
try
{throw x/y;}
```
(h)
```
try
{if(!x) throw x;}
catch(x)
{cout << "x is zero \n";}
```

Programming Exercises

13.1 *Write a program containing a possible exception. Use a try block to throw it and a catch block to handle it properly.*

13.2 *Write a program that illustrates the application of multiple catch statements.*

13.3 *Write a program which uses* ***catch(...)*** *handler.*

13.4 *Write a program that demonstrates how certain exception types are not allowed to be thrown.*

13.5 *Write a program to demonstrate the concept of rethrowing an exception.*

13.6 *Write a program with the following:*

(a) *A function to read two double type numbers from keyboard*

(b) *A function to calculate the division of these two numbers*

(c) *A try block to throw an exception when a wrong type of data is keyed in*

(d) *A try block to detect and throw an exception if the condition "divide-by-zero" occurs*

(e) *Appropriate catch block to handle the exceptions thrown*

13.7 *Write a main program that calls a deeply nested function containing an exception. Incorporate necessary exception handling mechanism.*

14

Introduction to the Standard Template Library

Key Concepts

- Software evolution
- Standard templates
- Standard C++ library
- Containers
- Sequence containers
- Associative containers
- Derived containers
- Algorithms
- Iterators
- Function object

14.1 Introduction

We have seen how templates can be used to create generic classes and functions that could extend support for generic programming. In order to help the C++ users in generic programming, Alexander Stepanov and Meng Lee of Hewlett-Packard developed a set of general-purpose templatized classes (data structures) and functions (algorithms) that could be used as a standard approach for storing and processing of data. The collection of these generic classes and functions is called the *Standard Template Library (STL)*. The STL has now become a part of the ANSI standard C++ class library.

STL is large and complex and it is difficult to discuss all of its features in this chapter. We therefore present here only the most important features that would enable the readers to begin using the STL effectively. Using STL can save considerable time and effort, and lead to high quality programs. All these benefits are possible because we are basically "reusing" the well-written and well-tested components defined in the STL.

STL components which are now part of the Standard C++ Library are defined in the **namespace std**. We must therefore use the **using namespace** directive

```
using namespace std;
```

to inform the compiler that we intend to use the Standard C++ Library. All programs in this chapter use this directive.

14.2 Components of STL

The STL contains several components. But at its core are three key components. They are:

- containers,
- algorithms, and
- iterators.

These three components work in conjunction with one another to provide support to a variety of programming solutions. The relationship between the three components is shown in Fig. 14.1. *Algorithms* employ *iterators* to perform operations stored in *containers*.

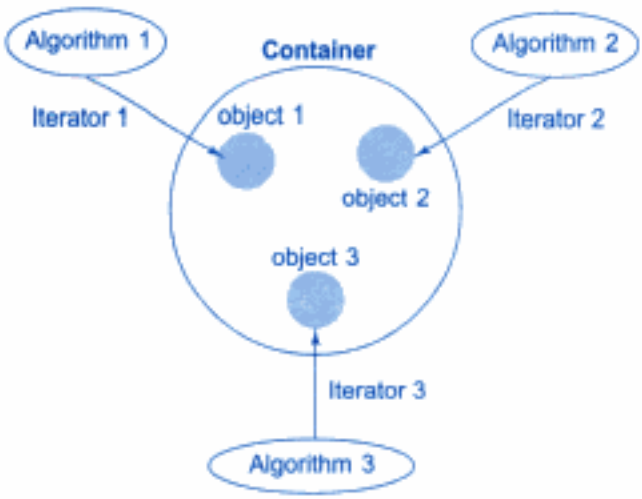


Fig. 14.1 ⇔ *Relationship between the three STL components*

A *container* is an object that actually stores data. It is a way data is organized in memory. The STL containers are implemented by template classes and therefore can be easily customized to hold different types of data.

An *algorithm* is a procedure that is used to process the data contained in the containers. The STL includes many different kinds of algorithms to provide support to tasks such as initializing, searching, copying, sorting, and merging. Algorithms are implemented by tem plate functions.

An *iterator* is an object(like a pointer) that points to an element in a container. We can use iterators to move through the contents of containers. Iterators are handled just like pointers. We can increment or decrement them. Iterators connect algorithms with containers and play a key role in the manipulation of data stored in the containers.

14.3 Containers

As stated earlier, containers are objects that hold data (of same type). The STL defines ten containers which are grouped into three categories as illustrated in Fig. 14.2. Table 14.1 gives the details of all these containers as well as header to be included to use each one of them and the type of iterator supported by each container class.

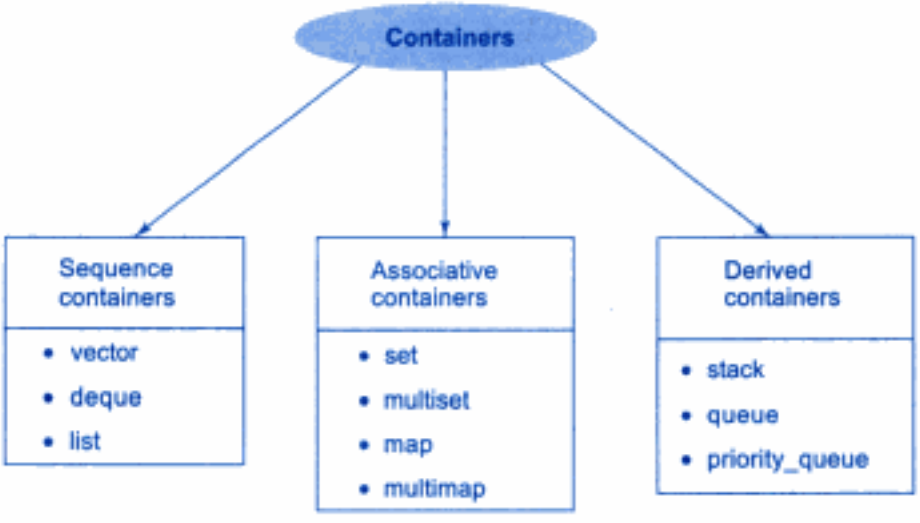


Fig. 14.2 ⇔ *Three major categories of containers*

Table 14.1 *Containers supported by the STL*

Container	*Description*	*Header file*	*Iterator*
vector	A dynamic array. Allows insertions and deletions at back. Permits direct access to any element	<vector>	Random access
list	A bidirectional, linear list. Allows insertions and deletions anywhere.	<list>	Bidirectional
deque	A double-ended queue. Allows insertions and deletions at both the ends. Permits direct access to any element.	<deque>	Random access
set	An associate container for storing unique sets. Allows rapid lookup. (No duplicates allowed)	<set>	Bidirectional

(Contd)

multiset	An associate container for storing non-unique sets. (Duplicates allowed)	<set>	Bidirectional
map	An associate container for storing unique key/value pairs. Each key is associated with only one value (One-to-one mapping). Allows key-based lookup.	<map>	Bidirectional
multimap	An associate container for storing key/value pairs in which one key may be associated with more than one value (one-to-many mapping). Allows key-based lookup.	<map>	Bidirectional
stack	A standard stack. Last-in-first-out(LIFO).	<stack>	No iterator
queue	A standard queue. First-in-first-out(FIFO).	<queue>	No iterator
priority–queue	A priority queue. The first element out is always the highest priority element.	<queue>	No iterator

Each container class defines a set of functions that can be used to manipulate its contents. For example, a vector container defines functions for inserting elements, erasing the contents, and swapping the contents of two vectors.

Sequence Containers

Sequence containers store elements in a linear sequence, like a line as shown in Fig. 14.3. Each element is related to other elements by its position along the line. They all expand themselves to allow insertion of elements and all of them support a number of operations on them.

Fig. 14.3 ⇔ *Elements in a sequence container*

The STL provides three types of sequence containers:

- vector
- list
- deque

Elements in all these containers can be accessed using an iterator. The difference between the three of them is related to only their performance. Table 14.2 compares their performance in terms of speed of random access and insertion or deletion of elements.

Hidden page

Hidden page

Table 14.4 *Contd*

fill_n( )	Fills first n elements with a specified value
generate( )	Replaces all elements with the result of an operation
generate_n( )	Replaces first n elements with the result of an operation
iter_swap( )	Swaps elements pointed to by iterators
random_shuffle( )	Places elements in random order
remove( )	Deletes elements of a specified value
remove_copy( )	Copies a sequence after removing a specified value
remove_copy_if( )	Copies a sequence after removing elements matching a predicate
remove_if( )	Deletes elements matching a predicate
replace( )	Replaces elements with a specified value
replace_copy( )	Copies a sequence replacing elements with a given value
replace_copy_if( )	Copies a sequence replacing elements matching a predicate
replace_if( )	Replaces elements matching a predicate
reverse( )	Reverses the order of elements
reverse_copy( )	Copies a sequence into reverse order
rotate( )	Rotates elements
rotate_copy( )	Copies a sequence into a rotated
swap( )	Swaps two elements
swap_ranges( )	Swaps two sequences
transform( )	Applies an operation to all elements
unique( )	Deletes equal adjacent elements
unique_copy( )	Copies after removing equal adjacent elements

Table 14.5 *Sorting algorithms*

Operations	***Description***
binary_search( )	Conducts a binary search on an ordered sequence
equal_range( )	Finds a subrange of elements with a given value
inplace_merge( )	Merges two consecutive sorted sequences
lower_bound( )	Finds the first occurrence of a specified value
make_heap( )	Makes a heap from a sequence
merge( )	Merges two sorted sequences
nth_element( )	Puts a specified element in its proper place
partial_sort( )	Sorts a part of a sequence
partial_sort_copy()	Sorts a part of a sequence and then copies
Partition( )	Places elements matching a predicate first
pop_heap( )	Deletes the top element
push_heap( )	Adds an element to heap
sort( )	Sorts a sequence
sort_heap( )	Sorts a heap
stable_partition( )	Places elements matching a predicate first matching relative order
stable_sort( )	Sorts maintaining order of equal elements
upper_bound( )	Finds the last occurrence of a specified value

Table 14.6 *Set algorithms*

Operations	Description
includes( )	Finds whether a sequence is a subsequence of another
set_difference( )	Constructs a sequence that is the difference of two ordered sets
set_intersection( )	Constructs a sequence that contains the intersection of ordered sets
set_symmetric_difference()	Produces a set which is the symmetric difference between two ordered sets
set_union( )	Produces sorted union of two ordered sets

Table 14.7 *Relational algorithms*

Operations	Description
equal( )	Finds whether two sequences are the same
lexicographical_compare()	Compares alphabetically one sequence with other
max( )	Gives maximum of two values
max_element( )	Finds the maximum element within a sequence
min( )	Gives minimum of two values
min_element( )	Finds the minimum element within a sequence
mismatch( )	Finds the first mismatch between the elements in two sequences

Table 14.8 *Numeric algorithms*

Operations	Description
accumulate( )	Accumulates the results of operation on a sequence
adjacent_difference( )	Produces a sequence from another sequence
inner_product( )	Accumulates the results of operation on a pair of sequences
partial_sum( )	Produces a sequence by operation on a pair of sequences

14.5 Iterators

Iterators behave like pointers and are used to access container elements. They are often used to traverse from one element to another, a process known as *iterating* through the container.

There are five types of iterators as described in Table 14.9.

Table 14.9 *Iterators and their characteristics*

Iterator	Access method	Direction of movement	I/O capability	Remark
Input	Linear	Forward only	Read only	Cannot be saved
Output	Linear	Forward only	Write only	Cannot be saved
Forward	Linear	Forward only	Read/Write	Can be saved
Bidirectional	Linear	Forward and backward	Read/Write	Can be saved
Random	Random	Forward and backward	Read/Write	Can be saved

Different types of iterators must be used with the different types of containers (See Table 14.1). Note that only sequence and associative containers are traversable with iterators.

Each type of iterator is used for performing certain functions. Figure 14.4 gives the functionality Venn diagram of the iterators. It illustrates the level of functionality provided by different categories of iterators.

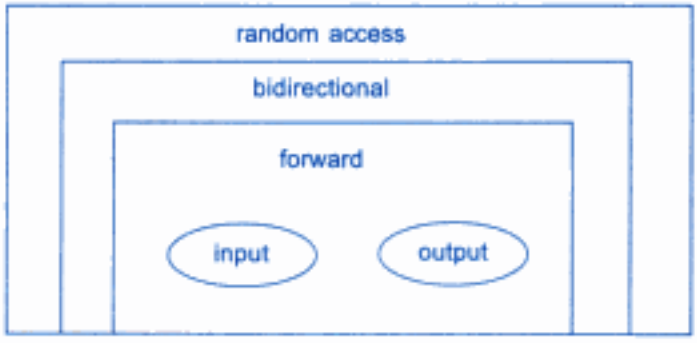


Fig. 14.4 ⇔ *Functionality Venn diagram of iterators*

The *input* and *output* iterators support the least functions. They can be used only to traverse in a container. The *forward* iterator supports all operations of input and output iterators and also retains its position in the container. A *bidirectional* iterator, while supporting all forward iterator operations, provides the ability to move in the backward direction in the container. A *random access* iterator combines the functionality of a bidirectional iterator with an ability to jump to an arbitrary location. Table 14.10 summarizes the operations that can be performed on each iterator type.

Table 14.10 *Operations supported by iterators*

Iterator	*Element access*	*Read*	*Write*	*Increment operation*	*Comparison*
Input	->	v = *p		++	==, !=
Output			*p = v	++	
Forward	->	v = *p	*p = v	++	==, !=
Bidirectional	->	v = *p	*p = v	++, - -	==, !=
Random access	->, [ ]	v = *p	*p = v	++, - -, +, -, +=, -=	==, !=, <, >, <=, >=

14.6 Application of Container Classes

It is beyond the scope of this book to examine all the containers supported in the STL and provide illustrations. Therefore, we illustrate here the use of the three most popular containers, namely, **vector**, **list**, and **map**.

Vectors

The **vector** is the most widely used container. It stores elements in contiguous memory locations and enables direct access to any element using the subscript operator [ ]. A **vector** can change its size dynamically and therefore allocates memory as needed at run time.

The **vector** container supports random access iterators, and a wide range of iterator operations (See Table 14.10) may be applied to a **vector** iterator. Class **vector** supports a number of constructors for creating **vector** objects.

```
vector<int> v1;              // Zero-length int vector
vector<double> v2(10);       // 10-element double vector
vector<int> v3(v4);          // Creates v3 from v4
vector<int> v(5, 2);         // 5-element vector of 2s
```

The **vector** class supports several member functions as listed in Table 14.11. We can also use all the STL algorithms on a **vector**.

Table 14.11 *Important member functions of the vector class*

Function	*Task*
at( )	Gives a reference to an element
back( )	Gives a reference to the last element
begin( )	Gives a reference to the first element
capacity( )	Gives the current capacity of the vector
clear( )	Deletes all the elements from the vector
empty( )	Determines if the vector is empty or not
end( )	Gives a reference to the end of the vector
erase( )	Deletes specified elements
insert( )	Inserts elements in the vector
pop_back( )	Deletes the last element
push_back( )	Adds an element to the end
resize( )	Modifies the size of the vector to the specified value
size( )	Gives the number of elements
swap( )	Exchanges elements in the specified two vectors

Program 14.1 illustrates the use of several functions of the **vector** class template. Note that an iterator is used as a pointer to elements of the vector. We must include header file **<vector>** to use **vector** class in our programs.

USING VECTORS

```
#include <iostream>
#include <vector>     // Vector header file

using namespace std;

void display(vector<int> &v)
```

(Contd)

```
{
     for(int i=0;i<v.size();i++)
   {
       cout << v[i] << "  ";
   }
    cout << "\n";
}

int main()
{
vector<int> v;        // Create a vector of type int
cout << "Initial size = " << v.size() << "\n";
// Putting values into the vector
int x;
cout << "Enter five integer values: ";
for(int i=0; i<5; i++)
{
     cin >> x;
     v.push_back(x);
}
cout << "Size after adding 5 values: ";
cout << v.size() << "\n";

// Display the contents
cout << "Current contents: \n";
display(v);

// Add one more value
v.push_back(6.6);     // float value truncated to int

// Display size and contents
cout << "\nSize = " << v.size() << "\n";
cout << "Contents now: \n";
display(v);

// Inserting elements
vector<int> :: iterator itr = v.begin();   // iterator
itr = itr + 3;        // itr points to 4th element
v.insert(itr,1,9);

// Display the contents
cout << "\nContents after inserting: \n";
```

(Contd)

```
    display(v);

    // Removing 4th and 5th elements
    v.erase(v.begin()+3,v.begin()+5);  // Removes 4th and 5th element

    // Display the contents
    cout << "\nContents after deletion: \n";
    display(v);
    cout << "END\n";
    return(0);
}
```

Program 14.1

Given below is the output of Program 14.1:

```
Initial size = 0

Enter five integer values:  1 2 3 4 5
Size after adding 5 values:  5
Current contents:
1  2  3  4  5

Size = 6
Contents now:
1  2  3  4  5  6

Contents after inserting:
1  2  3  9  4  5  6

Contents after deletion:
1  2  3  5  6
END
```

The program uses a number of functions to create and manipulate a vector. The member function **size()** gives the current size of the vector. After creating an **int** type empty vector **v** of zero size, the program puts five values into the vector using the member function **push_back()**. Note that **push_back()** takes a value as its argument and adds it to the back end of the vector. Since the vector **v** is of type **int**, it can accept only integer values and therefore the statement

```
v.push_back(6.6);
```

truncates the values 6.6 to 6 and then puts it into the vector at its back end.

The program uses an iterator to access the vector elements. The statement

```
vector<int> :: iterator itr = v.begin();
```

declares an iterator **itr** and makes it to point to the first position of the vector. The statements

```
itr = itr + 3;
v.insert(itr,9);
```

inserts the value 9 as the fourth element. Similarly, the statement

```
v.erase(v.begin()+3, v.begin()+5);
```

deletes 4th and 5th elements from the vector. Note that **erase(m,n)** deletes only n-m elements starting from mth element and the nth element is not deleted.

The elements of a vector may also be accessed using subscripts (as we do in arrays). Notice the use of **v[i]** in the function **display()** for displaying the contents of **v**. The call **v.size()** in the **for** loop of **display()** gives the current size of **v**.

Lists

The **list** is another container that is popularly used. It supports a bidirectional, linear list and provides an efficient implementation for deletion and insertion operations. Unlike a vector, which supports random access, a list can be accessed sequentially only.

Bidirectional iterators are used for accessing list elements. Any algorithm that requires input, output, forward, or bidirectional iterators can operate on a **list**. Class **list** provides many member functions for manipulating the elements of a list. Important member functions of the **list** class are given in Table 14.12. Use of some of these functions is illustrated in Program 14.2. Header file **<list>** must be included to use the container class **list**.

USING LISTS

```
#include <iostream>
#include <list>
#include <cstdlib>    // For using rand() function

using namespace std;

void display(list<int> &lst)
{
      list<int> :: iterator p;
```

(Contd)

Hidden page

```
    // Sorting and merging
    listA.sort();
    listB.sort();
    listA.merge(listB);
    cout << "Merged sorted lists \n";
    display(listA);

    // Reversing a list
    listA.reverse();
    cout << "Reversed merged list \n";
    display(listA);

    return(0);
}
```

Program 14.2

Output of the Program 14.2 would be:

```
List1
0, 184, 63,

List2
265, 191, 157, 114, 293,

Now List1
100, 0, 184, 63, 200,

Now List2
191, 157, 114, 293,

Merged unsorted lists
100, 0, 184, 63, 191, 157, 114, 200, 293,

Merged sorted lists
0, 63, 100, 114, 157, 184, 191, 200, 293,

Reversed merged list
293, 200, 191, 184, 157, 114, 100, 63, 0,
```

The program declares two empty lists, **list1** with zero length and **list2** of size 5. The **list1** is filled with three values using the member function **push_back()** and math function rand(). The **list2** is filled using a **list** type iterator **p** and a **for** loop. Remember that

list2.begin() gives the position of the first element while **list2.end()** gives the position immediately after the last element. Values are inserted at both the ends using **push_front()** and **push_back()** functions. The function **pop_front()** removes the first element in the list. Similarly, we may use **pop_back()** to remove the last element.

The objects of list can be initialized with other list objects like

```
listA = list1;
listB = list2;
```

The statement

```
list1.merge(list2);
```

simply adds the **list2** elements to the end of **list1**. The elements in a list may be sorted in increasing order using **sort()** member function. Note that when two sorted lists are merged, the elements are inserted in appropriate locations and therefore the merged list is also a sorted one.

We use a **display()** function to display the contents of various lists. Note the difference between the implementations of **display()** in Program 14.1 and Program 14.2.

Table 14.12 *Important member functions of the list class*

Function	*Task*
back( )	Gives reference to the last element
begin( )	Gives reference to the first element
clear( )	Deletes all the elements
empty( )	Decides if the list is empty or not
end( )	Gives reference to the end of the list
erase( )	Deletes elements as specified
insert( )	Inserts elements as specified
merge( )	Merges two ordered lists
pop_back( )	Deletes the last element
pop_front( )	Deletes the first element
push_back( )	Adds an element to the end
push_front( )	Adds an element to the front
remove( )	Removes elements as specified
resize( )	Modifies the size of the list
reverse( )	Reverses the list
size( )	Gives the size of the list
sort( )	Sorts the list
splice( )	Inserts a list into the invoking list
swap( )	Exchanges the elements of a list with those in the invoking list
unique( )	Deletes the duplicating elements in the list

Maps

A **map** is a sequence of (key, value) pairs where a single value is associated with each unique key as shown in Fig. 14.5. Retrieval of values is based on the key and is very fast. We should specify the key to obtain the associated value.

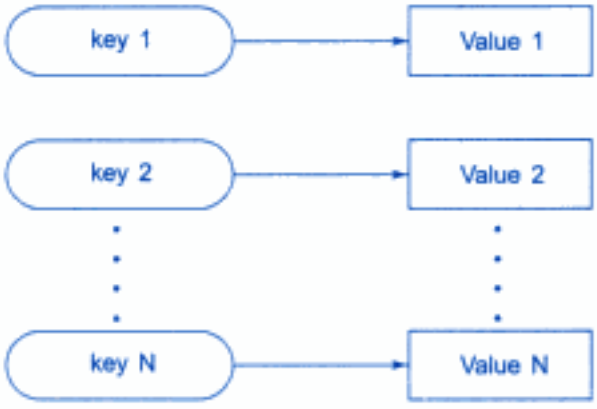


Fig. 14.5 ⇔ *The key-value pairs in a map*

A **map** is commonly called an *associative array*. The key is specified using the subscript operator [ ] as shown below:

```
phone[ "John" ] = 1111;
```

This creates an entry for "John" and associates(i.e. assigns) the value 1111 to it. **phone** is a **map** object. We can change the value, if necessary, as follows:

```
phone[ "John" ] = 9999;
```

This changes the value 1111 to 9999. We can also insert and delete pairs anywhere in the **map** using **insert( )** and **erase( )** functions. Important member functions of the **map** class are listed in Table 14.13.

Table 14.13 *Important member functions of the map class*

Function	*Task*
begin( )	Gives reference to the first element
clear( )	Deletes all elements from the map
empty( )	Decides whether the map is empty or not
end( )	Gives a reference to the end of the map
erase( )	Deletes the specified elements
find( )	Gives the location of the specified element
insert( )	Inserts elements as specified
size( )	Gives the size of the map
swap( )	Exchanges the elements of the given map with those of the invoking map

Program 14.13 shows a simple example of a **map** used as an associative array. Note that **<map>** header must be included.

Hidden page

Hidden page

Function objects are often used as arguments to certain containers and algorithms. For example, the statement

```
sort(array, array+5, greater<int>());
```

uses the function object **greater<int>( )** to sort the elements contained in **array** in descending order.

Besides comparisons, STL provides many other predefined function objects for performing arithmetical and logical operations as shown in Table 14.14. Note that there are function objects corresponding to all the major C++ operators. For using function objects, we must include **<functional>** header file.

Table 14.14 *STL function objects in <functional>*

Function object	*Type*	*Description*
divides<T>	arithmetic	x/y
equal_to<T>	relational	x == y
greater<T>	relational	x > y
greater_equal<T>	relational	x >= y
less<T>	relational	x < y
less_equal<T>	relational	x <= y
logical_and<T>	logical	x && y
logical_not<T>	logical	!x
logical_or<T>	logical	x || y
minus<T>	arithmetic	x - y
modulus<T>	arithmetic	x % y
negate<T>	arithmetic	- x
not_equal_to<T>	relational	x != y
plus<T>	arithmetic	x + y
multiplies<T>	arithmetic	x * y

Note: The variables x and y represent objects of class T passed to the function object as arguments.

Program 14.4 illustrates the use of the function object **greater<>( )** in **sort( )** algorithm.

USE OF FUNCTION OBJECTS IN ALGORITHMS

```
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;
int main()
{
        int x[] = {10,50,30,40,20};
        int y[] = {70,90,60,80};
```

(Contd)

```
        sort(x,x+5,greater<int>());
        sort(y,y+4);
        for(int i=0; i<5; i++)
                cout << x[i] << " ";
        cout << "\n";
        for(int j=0; j<4; j++)
                cout << y[j] << " ";
        cout << "\n";
        int z[9];
        merge(x,x+5,y,y+4,z);
        for(i=0; i<9; i++)
                cout << z[i] << " ";
        cout << "\n";
        return(0);
}
```

Program 14.4

Output of Program 14.4:

```
50 40 30 20 10
60 70 80 90
50 40 30 20 10 60 70 80 90
```

note

The program creates two arrays **x** and **y** and initializes them with specified values. The program then sorts both of them using the algorithm **sort( )**. Note that **x** is sorted using the function object **greater<int>( )** and **y** is sorted without it and therefore the elements in **x** are in descending order.

The program finally merges both the arrays and displays the content of the merged array. Note the form of **merge()** function and the results it produces.

SUMMARY

- A collection of generic classes and functions is called the Standard Template Library (STL). STL components are part of C++ standard library.
- The STL consists of three main components: containers, algorithms, and iterators.
- Containers are objects that hold data of same type. Containers are divided into three major categories: sequential, associative, and derived.

Hidden page

- sorting algorithms
- **stack**
- standard C++ library
- standard template library
- templates
- templatized classes
- tree
- using namespace
- values
- **vector**

Review Questions

14.1 *What is STL? How is it different from the C++ Standard Library? Why is it gaining importance among the programmers?*

14.2 *List the three types of containers.*

14.3 *What is the major difference between a sequence container and an associative container?*

14.4 *What are the best situations for the use of the sequence containers?*

14.5 *What are the best situations for the use of the associative containers?*

14.6 *What is an iterator? What are its characteristics?*

14.7 *What is an algorithm? How STL algorithms are different from the conventional algorithms?*

14.8 *How are the STL algorithms implemented?*

14.9 *Distinguish between the following:*

(a) *lists and vectors*
(b) *sets and maps*
(c) *maps and multimaps*
(d) *queue and deque*
(e) *arrays and vectors*

14.10 *Compare the performance characteristics of the three sequence containers.*

14.11 *Suggest appropriate containers for the following applications:*

(a) *Insertion at the back of a container.*
(b) *Frequent insertions and deletion at both the ends of a container.*
(c) *Frequent insertions and deletions in the middle of a container.*
(d) *Frequent random access of elements.*

14.12 *State whether the following statements are true or false.*

(a) *An iterator is a generalized form of pointer.*
(b) *One purpose of an iterator is to connect algorithms to containers.*
(c) *STL algorithms are member functions of containers.*
(d) *The size of a vector does not change when its elements are removed.*
(e) *STL algorithms can be used with c-like arrays.*
(f) *An iterator can always move forward or backward through a container.*

(g) *The member function* ***end()*** *returns a reference to the last element in the container.*

(h) *The member function* ***back()*** *removes the element at the back of the container.*

(i) *The* ***sort()*** *algorithm requires a random-access iterator.*

(j) *A map can have two or more elements with the same key value.*

Debugging Exercises

14.1 Identify the errror in the following program.

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

#define NAMESIZE 40

using namespace std;

class EmployeeMaster
{
private:
    char name[NAMESIZE];
    int id;

public:
    EmployeeMaster()
    {
        strcpy(name, "");
        id = 0;
    }

    EmployeeMaster(char name[NAMESIZE], int id)
        :id(id)
    {
        strcpy(this->name, name);
    }

    EmployeeMaster* getValuesFromUser()
    {
        EmployeeMaster *temp = new EmployeeMaster();
        cout << endl << "Enter user name : ";
        cin >> temp->name;
        cout << endl << "Enter user ID : ";
        cin >> temp->id;
        return temp;
    }
```

```
    void displayRecord()
    {
        cout << endl << "Name : " << name;
        cout << endl << "ID   : " << id << endl;
    }
};

void main()
{
        vector <EmployeeMaster*> emp;
        EmployeeMaster *temp = new EmployeeMaster();
        emp.push_back(getValuesFromUser());
        emp[0]->displayRecord();
        delete temp;

        temp = new EmployeeMaster("AlanKay", 3);
        emp.push_back(temp);
        emp[emp.capacity()]->displayRecord();
        emp[emp.size()]->displayRecord();
}
```

14.2 Identify the error in the following program.

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector <int> v1;
    v1.push_back(10);
    v1.push_back(30);

    vector <int> v2;
    v2.push_back(20);
        v2.push_back(40);

    if(v1==v2)
        cout<<"vectors are equal";
    else
        cout<<"vectors are unequal\t";
        v1.swap(20);
    for(int y=0; y<v1.size(); y++)
```

```
    {
        cout<<"V1="<<v1[y]<<" ";
                cout<<"V2="<<v2[y]<<" ";
    }
    return 0;
}
```

14.3 Identify the error in the following program.

```
#include<iostream>
#include<list>

void main()
{
    list <int> l1;

    l1.push_front(10);
    l1.push_back(20);
    l1.push_front(30);
    l1.push_front(40);
    l1.push_back(10);
    l1.pop_front(40);

    l1.reverse();
    l1.unique();

}
```

Programming Exercises

14.1 *Write a code segment that does the following:*
- (a) *Defines a vector* **v** *with a maximum size of 10*
- (b) *Sets the first element of* **v** *to 0*
- (c) *Sets the last element of* **v** *to 9*
- (d) *Sets the other elements to 1*
- (e) *Displays the contents of* **v**

14.2 *Write a program using the* ***find()*** *algorithm to locate the position of a specified value in a sequence container.*

14.3 *Write a program using the algorithm* ***count()*** *to count how many elements in a container have a specified value.*

14.4 *Create an array with even numbers and a list with odd numbers. Merge two sequences of numbers into a vector using the algorithm* ***merge()****. Display the vector.*

14.5 *Create a **student** class that includes a student's first name and his roll_number. Create five objects of this class and store them in a list thus creating a phone_lit. Write a program using this list to display the student name if the roll_number is given and vice-versa.*

14.6 *Redo the Exercise 14.17 using a set.*

14.7 *A table gives a list of car models and the number of units sold in each type in a specified period. Write a program to store this table in a suitable container, and to display interactively the total value of a particular model sold, given the unit-cost of that model.*

14.8 *Write a program that accepts a shopping list of five items from the keyboard and stores them in a vector. Extend the program to accomplish the following:*

(a) *To delete a specified item in the list*
(b) *To add an item at a specified location*
(c) *To add an item at the end*
(d) *To print the contents of the vector*

15

Manipulating Strings

Key Concepts

- C-strings
- The string class
- Creating string objects
- Manipulating strings
- Relational operations on strings
- Comparing strings
- String characteristics
- Swapping strings

15.1 Introduction

A string is a sequence of characters. We know that C++ does not support a built-in string type. We have used earlier null-terminated character arrays to store and manipulate strings. These strings are called *C-strings* or *C-style strings*. Operations on C-strings often become complex and inefficient. We can also define our own string classes with appropriate member functions to manipulate strings. This was illustrated in Program 7.4 (Mathematical Operation of Strings).

ANSI standard C++ now provides a new class called **string**. This class improves on the conventional C-strings in several ways. In many situations, the string objects may be used like any other built-in type data. Further, although it is not considered as a part of the STL, **string** is treated as another container class by C++ and therefore all the algorithms that are applicable for containers can be used with the **string** objects. For using the **string** class, we must include <**string**> in our program.

The **string** class is very large and includes many constructors, member functions and operators. We may use the constructors, member functions and operators to achieve the following:

- Creating string objects
- Reading string objects from keyboard
- Displaying string objects to the screen
- Finding a substring from a string
- Modifying string objects
- Comparing string objects
- Adding string objects
- Accessing characters in a string
- Obtaining the size of strings
- And many other operations

Table 15.1 gives prototypes of three most commonly used constructors and Table 15.2 gives a list of important member functions. Table 15.3 lists a number of operators that can be used on **string** objects.

Table 15.1 *Commonly used string constructors*

Constructor	*Usage*
String();	For creating an empty string
String(const chat *str);	For creating a string object from a null-terminated string
String(const string & str);	For creating a string object from other string object

Table 15.2 *Important functions supported by the string class*

Function	*Task*
append()	Appends a part of string to another string
Assign()	Assigns a partial string
at()	Obtains the character stored at a specified location
Begin()	Returns a reference to the start of a string
capacity()	Gives the total elements that can be stored.
compare()	Compares string against the invoking string
empty()	Returns true if the string is empty; Otherwise returns false
end()	Returns a reference to the end of a string
erase()	Removes characters as specified
find()	Searches for the occurrence of a specified substring
insert()	Inserts characters at a specified location
length()	Gives the number of elements in a string
max size()	Gives the maximum possible size of a string object in a give system
replace()	Replace specified characters with a given string
resize()	Changes the size of the string as specified
size()	Gives the number of characters in the string
swap()	Swaps the given string with the invoking string

Table 15.3 *Operators for string objects*

Operator	*Meaning*
=	Assignment
+	Concatenation
+=	Concatenation assignment
= =	Equality
!=	Inequality
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
[ ]	Subscription
<<	Output
>>	Input

15.2 Creating (string) Objects

We can create **string** objects in a number of ways as illustrated below:

```
string s1;              // Using constructor with no argument
string s2("xyz");       // Using one-argument constructor
s1 = s2;                // Assigning string objects
s3 = "abc" + s2         // Concatenating strings
cin >> s1;              // Reading through keyboard (one word)
getline(cin, s1);       // Reading through keyboard a line of text
```

The overloaded + operator concatenates two string objects. We can also use the operator += to append a string to the end of a string. Examples:

```
s3 += s1;        // s3 = s3 + s1
s3 += "abc";     // s3 = s3 + "abc"
```

The operators << and >> are overloaded to handle input and output of string objects. Examples:

```
cin >> s2;              // Input to string object (one word)
cout << s2;             // Displays the contents of s2
getline(cin, s2);       // Reads embedded blanks
```

note

Using **cin** and >> operator we can read only one word of a string while the **getline()** function permits us to read a line of text containing embedded blanks.

Program 15.1 demonstrates the several ways of creating string objects in a program.

CREATING STRING OBJECTS

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    // Creating string objects
    string s1;                      // Empty string object
    string s2(" New");              // Using string constant
    string s3(" Delhi");

    // Assigning value to string objects
    s1 = s2;                        // Using string object
    cout << "S1 = " << s1 << "\n";

    // Using a string constant
    s1 = "Standard C++";
    cout << "Now S1 = " << s1 << "\n";

    // Using another object
    string s4(s1);
    cout << "S4 = " << s4 << "\n\n";

    // Reading through keyboard
    cout << "ENTER A STRING \n";
    cin >> s4;                      // Delimited by blank space
    cout << "Now S4 = " << s4 << "\n\n";

    // Concatenating strings
    s1 = s2 + s3;
    cout << "S1 finally contains: " << s1 << "\n";

    return 0;
}
```

PROGRAM 15.1

The output of Program 15.1 would be:

```
S1 =  New
Now S1 = Standard C++
S4 = Standard C++
```

```
ENTER A STRING
COMPUTER CENTRE
Now S4 = COMPUTER

S1 finally contains:  New Delhi
```

15.3 Manipulating String Objects

We can modify contents of **string** objects in several ways, using the member functions such as **insert(), replace(), erase(),** and **append()**. Program 15.2 demonstrates the use of some of these functions.

MODIFYING STRING OBJECTS

```
#include <iostream
#include <string>

using namespace std;

int main()
{
  string s1("12345");
  string s2("abcde");

  cout << "Original Strings are: \n";
  cout << "S1: " << s1 << "\n";
  cout << "S2: " << s2 << "\n\n";

  // Inserting a string into another
  cout << "Place S2 inside S1 \n";
  s1.insert(4,s2);
  cout << "Modified S1: " << s1 << "\n\n";

  // Removing characters in a string
  cout << "Remove 5 Characters from S1 \n";
  s1.erase(4,5);
  cout << "Now S1: " << s1 << "\n\n";

  // Replacing characters in a string
  cout << "Replace Middle 3 Characters in S2 with S1 \n";
```

(Contd)

Hidden page

```
int main()
{
   string s1("ABC");
   string s2("XYZ");
   string s3 = s1 + s2;

   if(s1 != s2)
         cout << "s1 is not equal to s2 \n";
   if(s1 > s2)
         cout << "s1 greater than s2 \n";
   else
         cout << "s2 greater than s1 \n";
   if(s3 == s1 + s2)
         cout << "s3 is equal to s1+s2 \n\n";
   int x = s1.compare(s2);
   if(x == 0)
         cout << "s1 == s2 \n";
   else if(x > 0)
         cout << "s1 > s2 \n";
   else                              // x < 0
         cout << "s1 < s2 \n";

   return 0;
}
```

PROGRAM 15.3

Program 15.3 shows how these operators are used.

This program produces the following output:

```
s1 is not equal to s2
s2 greater than s1
s3 is equal to s1+s2

s1 < s2
```

15.5 String Characteristics

Class **string** supports many functions that could be used to obtain the characteristics of strings such as size, length, capacity, etc. The size or length denotes the number of elements

currently stored in a given string. The capacity indicates the total elements that can be stored in the given string. Another characteristic is the *maximum size* which is the largest possible size of a string object that the given system can support. Program 15.4 illustrates how these characteristics are obtained and used in an application.

OBTAINING STRING CHARACTERISTICS

```
#include <iostream>
#include <string>

using namespace std;

void display(string &str)
{
   cout << "Size = " << str.size() << "\n";
   cout << "Length = " << str.length() << "\n";
   cout << "Capacity = " << str.capacity() << "\n";
   cout << "Maximum Size = " << str.max_size() << "\n";
   cout << "Empty: " << (str.empty() ? "yes" : "no");
   cout << "\n\n";
}

int main()
{
   string str1;

   cout << "Initial status: \n";
   display(str1);

   cout << "Enter a string (one word) \n";
   cin >> str1;
   cout  << "Status now: \n";
   display(str1);

   str1.resize(15);
   cout << "Status after resizing: \n";
   display(str1);
   cout << "\n";

   return 0;
}
```

PROGRAM 15.4

Shown below is the output of Program 15.4:

```
Initial status:
Size = 0
```

Hidden page

```
using namespace std;

int main()
{
  string s("ONE TWO THREE FOUR");

  cout << "The string contains: \n";
  for(int i=0;i<s.length();i++)
        cout << s.at(i);              // Display one character
  cout << "\nString is shown again: \n";
  for(int j=0;j<s.length();j++)
        cout << s[j];

  int x1 = s.find("TWO");
  cout << "\n\nTWO is found at: " << x1 << "\n";

  int x2 = s.find_first_of('T');
  cout << "\nT is found first at: " << x2 << "\n";
  int x3 = s.find_last_of('R');
  cout << "\nR is last found at: " << x3 << "\n";

  cout << "\nRetrieve and print substring TWO \n";

  cout << s.substr(x1,3);
  cout << "\n";

  return 0;
}
```

PROGRAM 15.5

Shown below is the output of Program 15.5:

```
The string contains:
ONE TWO THREE FOUR
String is shown again:
ONE TWO THREE FOUR

TWO is found at: 4

T is found first at: 4

R is last fount at: 17

Retrieve and print substring TWO
TWO
```

We can access individual characters in a string using either the member function **at()** or the subscript operator [ ]. This is illustrated by the following statements:

```
cout << s.at(i);
cout << s[i];
```

The statement

```
int x1 = s.find("TWO");
```

locates the position of the first character of the substring "TWO". The statement

```
cout << s.substr(x1,3);
```

finds the substring "TWO". The first argument **x1** specifies the location of the first character of the required substring and the second argument gives the length of the substring.

15.7 Comparing and Swapping

The **string** supports functions for comparing and swapping strings. The **compare()** function can be used to compare either two strings or portions of two strings. The **swap()** function can be used for swapping the contents of two **string** objects. The capabilities of these functions are demonstrated in Program 15.6.

COMPARING AND SWAPPING STRINGS

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
   string s1("Road");
   string s2("Read");
   string s3("Red");
   cout << "s1 = " << s1 << "\n";
   cout << "s2 = " << s2 << "\n";
   cout << "s3 = " << s3 << "\n";

   int x = s1.compare(s2);
   if(x == 0)
```

(Contd)

```
        cout << "s1 == s2" << "\n";
    else if(x > 0)
        cout << "s1 > s2" << "\n";
    else
        cout << "s1 < s2" << "\n";

    int a = s1.compare(0,2,s2,0,2);
    int b = s2.compare(0,2,s1,0,2);
    int c = s2.compare(0,2,s3,0,2);
    int d = s2.compare(s2.size()-1,1,s3,s3.size()-1,1);

    cout << "a = " << a << "\n" << "b = " << b << "\n";
    cout << "c = " << c << "\n" << "d = " << d << "\n";

    cout << "\nBefore swap: \n";
    cout << "s1 = " << s1 << "\n";
    cout << "s2 = " << s2 << "\n";
    s1.swap(s2);
    cout << "\nAfter swap: \n";
    cout << "s1 = " << s1 << "\n";
    cout << "s2 = " << s2 << "\n";

    return 0;
}
```

PROGRAM 15.6

The output of Program 15.6:

```
s1 = Road
s2 = Read
s3 = Red
s1 > s2
a = 1
b = -1
c = 0
d = 0

Before swap:
s1 = Road
s2 = Read

After swap:
s1 = Read
s2 = Road
```

The statement

```
int x = s1.compare(s2);
```

compares the string **s1** against **s2** and **x** is assigned 0 if the strings are equal, a positive number if **s1** is *lexicographically* greater than **s2** or a negative number otherwise.

The statement

```
int a = s1.compare(0,2,s2,0,2);
```

compares portions of **s1** and **s2**. The first two arguments give the starting subscript and length of the portion of **s1** to compare to **s2**, that is supplied as the third argument. The fourth and fifth arguments specify the starting subscript and length of the portion of **s2** to be compared. The value assigned to **a** is 0, if they are equal, 1 if substring of **s1** is greater than the substring of **s2**, –1 otherwise.

The statement

```
s2.swap(s2);
```

exchanges the contents of the strings **s1** and **s2**.

SUMMARY

- Manipulation and use of C-style strings become complex and inefficient. ANSI C++ provides a new class called **string** to overcome the deficiencies of C-strings.
- The string class supports many constructors, member functions and operators for creating and manipulating string objects. We can perform the following operations on the strings:
  - Reading strings from keyboard
  - Assigning strings to one another
  - Finding substrings
  - Modifying strings
  - Comparing strings and substrings
  - Accessing characters in strings]
  - Obtaining size and capacity of strings
  - Swapping strings
  - Sorting strings

Key Terms

- **<string>**
- **append()**
- **assign()**
- **at()**
- **begin()**
- capacity
- **capacity()**
- **compare()**
- comparing strings
- C-strings
- C-style strings
- **empty()**
- **end()**
- **erase()**
- **find()**
- **find_first_of()**
- **find_last_of()**
- **getline()**
- **insert()**
- length
- **length()**
- lexicographical
- **max_size()**
- maximum size
- relational operators
- **replace()**
- size
- **size()**
- string
- string class
- string constructors
- string objects
- **substr()**
- substring
- **swap()**
- swapping strings

Review Questions

15.1 *State whether the following statements are TRUE or FALSE:*

(a) *For using **string** class, we must include the header <string>.*

(b) ***string** objects are null terminated.*

(c) *The elements of a **string** object are numbered from 0.*

(d) *Objects of **string** class can be copied using the assignment operator.*

(e) *Function **end()** returns an iterator to the invoking **string** object.*

15.2 *How does a **string** type string differ from a C-type string?*

15.3 *The following statements are available to read strings from the keyboard.*

```
(a)  cin >> s1;
(b)  getline(cin, s1);
```

*where **s1** is a **string** object. Distinguish their behaviour.*

15.4 *Consider the following segment of a program:*

```
string s1("man"), s2, s3;
s2.assign(s1);
s3 = s1;
string s4("wo" + s1);
s2 += "age";
s3.append("ager");
s1[0] = 'v';
```

State the contents of the objects ***s1***, ***s2***, ***s3*** *and* ***s4*** *when executed.*

15.5 *We can access string elements using*

(a) ***at()*** *function*

(b) *subscript operator [ ]*

Compare their behaviour.

15.6 *What does each of the following statements do?*

(a) `s.replace(n,1,"/");`

(b) `s.erase(10);`

(c) `s1.insert(10,s2);`

(d) `int x = s1.compare(0, s2.size(), s2);`

(e) `s2 = s1.substr(10, 5);`

15.7 *Distinguish between the following pair of functions.*

(a) *max_size() and capacity()*

(b) *find() and rfind()*

(c) *begin() and rbegin()*

Debugging Exercises

15.1 Identify the error in the following program.

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

using namespace std;

void main()
{
        string str1("ghi");
        string str2("abc" + "def");
        str2+=str1;
        cout << str2.c_str();
}
```

15.2 Identify the error in the following program.

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

```
#include <string>

using namespace std;
void main()
{
        string str1("ABCDEF");
        string str2("123");
        string str3;

        str1.insert(2, str2);
        str1.erase(2,2);
        str1.replace(2,str2);

        cout << str1.c_str();
        cout << endl;
}
```

15.3 Identify the error in the following program.

```
#include <iostream>
#include <string>

using namespace std;

class Product
{
        int iProductNumber;
        string strProductName;
public:
      Product()
      {
      }

      Product(const int &number, const string &name)
      {
            setProductNumber(number);
            setProductName(name);
      }

      void setProductNumber(int n)
      {
             iProductNumber = n;
```

```
        }

        void setProductName(const string str)
        {
                strProductName = str;
        }

        int getProductNumber()
        {
                return iProductNumber;
        }

        const string getProductName()
        {
                return strProductName ;
        }

        Product& operator = (Product &source)
        {
                setProductNumber(source.iProductNumber);
                string strTemp = source.strProductName;
                setProductName(strTemp);
                return *this;
        }

        void display()
        {
                cout << "ProductName : " << getProductName();
                cout << " " ;
                cout << "ProductNumber : " << getProductNumber();
                cout << endl;
        }
};

void main()
{
        Product p1(1, 5);
        Product p2(3, "Dates");
        Product p3;
        p3 = p2 = p1;
```

```
        p3.display();
        p2.display();
}
```

15.4 Find errors, if any, in the following segment of code.

```
int len = s1. length();
for (int i=0; i<len;++i)
        cout << s1.at[];
```

Programming Exercises

15.1 *Write a program that reads the name*

```
Martin Luther King
```

from the keyboard into three separate ***string*** *objects and then concatenates them into a new* ***string*** *object using*

(a) *+ operator and*

(b) ***append()*** *function.*

15.2 *Write a program using an iterator and* ***while()*** *construct to display the contents of a* **string** object.

15.3 *Write a program that reads several city names from the keyboard and displays only those names beginning with characters "B" or "C".*

15.4 *Write a program that will read a line of text containing more than three words and then replace all the blank spaces with an underscore(_).*

15.5 *Write a program that counts the number of occurrences of a particular character, say 'e', in a line of text.*

15.6 *Write a program that reads the following text and counts the number of times the word "It" appears in it.*

```
It is new. It is singular.
It is simple. It must succeed!
```

15.7 *Modify the program in Exercise 15.14 to count the number of words which start with the character 's'.*

15.8 *Write a program that reads a list of countries in random order and displays them in alphabetical order. Use comparison operators and functions.*

15.9 *Given a string*

```
string s("123456789");
```

Write a program that displays the following:

1
2 3 2
3 4 5 4 3
4 5 6 7 6 5 4
5 6 7 8 9 8 7 6 5

16

New Features of ANSI C++ Standard

Key Concepts

- Boolean type data
- Wide-character literals
- Constant casting
- Static casting
- Dynamic casting
- Reinterpret casting
- Runtime type information
- Explicit constructors
- Mutable member data
- Namespaces
- Nesting of namespaces
- Operator keywords
- Using new keywords
- New style for headers

16.1 Introduction

The ISO / ANSI C++ Standard adds several new features to the original C++ specifications. Some are added to provide better control in certain situations and others are added for providing conveniences to C++ programmers. It is therefore important to note that it is technically possible to write full-fledged programs without using any of the new features. Important features added are:

1. New data types
   - bool
   - wchar_t
2. New operators
   - const_cast
   - static_cast
   - dynamic_cast
   - reinterpret_cast
   - typeid
3. Class implementation
   - Explicit constructors
   - Mutable members
4. Namespace scope

5. Operator keywords
6. New keywords
7. New headers

We present here a brief overview of these features.

16.2 New Data Types

The ANSI C++ has added two new data types to enhance the range of data types available in C++. They are **bool** and **wchar_t**.

The bool Data Type

The data type **bool** has been added to hold a Boolean value, **true** or **false**. The values **true** and **false** have been added as keywords to the C++ language. The **bool** type variables can be declared as follows.

```
bool b1;                    // declare b1 as bool type
b1 = true;                  // assign true value to it
bool b2 = false;            // declare and initialize
```

The default numeric value of **true** is 1 and **false** is 0. Therefore, the statements

```
cout << "b1 = " << b1;      // b1 is true
cout << "b2 = " << b2;      // b2 is false
```

will display the following output:

```
b1 = 1
b2 = 0
```

We can use the **bool** type variables or the values **true** and **false** in mathematical expressions. For instance,

```
int x = false + 5*m - b1;
```

is valid and the expression on the right evaluates to 9 assuming **b1** is true and **m** is 2. Values of type **bool** are automatically elevated to integers when used in non-Boolean expressions.

It is possible to convert implicitly the data types pointers, integers or floating point values to **bool** type. For example, the statements

```
bool x  = 0;
bool y = 100;
bool z = 15.75;
```

assign **false** to **x** and **true** to **y** and **z**.

Program 16.1 demonstrates the features of **bool** type data.

USE OF bool TYPE DATA

```
#include <iostream>

using namespace std;

int main()
{
        int x1 = 10,x2 = 20,m = 2;
        bool b1, b2;

        b1 = x1 == x2;      // False
        b2 = x1 < x2;               // True

        cout << "b1 is " << b1 << "\n";
        cout << "b2 is " << b2 << "\n";

        bool b3  =  true;
        cout << "b3 is " << b3 << "\n";

        if(b3)
              cout << "Very Good" << "\n";
        else
              cout << "Very Bad" << "\n";

        int x3 = false + 5*m-b3;
        b1 = x3;
        b2 = 0;
        cout << "x3 = " << x3 << "\n";
        cout << "Now b1 = " << b1 << " and b2 = " << b2 << "\n";

        return 0;
}
```

PROGRAM 16.1

The output of Program 16.1 would be:

```
b1 is 0
b2 is 1
```

```
b3 is 1
Very Good
x3 = 9
Now b1 = 1 and b2 = 0
```

The wchar_t Data Type

The character type **wchar_t** has been defined in ANSI C++ to hold 16-bit wide characters. The 16-bit characters are used to represent the character sets of languages that have more than 255 characters, such as Japanese. This is important if we are writing programs for international distribution.

ANSI C++ also introduces a new character literal known as *wide_character* literal which uses two bytes of memory. Wide_character literals begin with the letter L, as follows:

```
L'xy'     // wide_character literal
```

16.3 New Operators

We have used cast operators (also known as *casts* or *type casts*) earlier in a number of programs. As we know, casts are used to convert a value from one type to another. This is necessary in situations where automatic conversions are not possible. We have used the following forms of casting:

```
double x = double(m);          // C++ type casting
double y = (double)n;          // C-type casting
```

Although these casts still work, ANSI C++ has added several new cast operators known as *static casts, dynamic casts, reinterpret casts* and *constant casts*. It also adds another operator known as **typeid** to verify the types of unknown objects.

The static_cast Operator

Like the conventional cast operators, the **static_cast** operator is used for any standard conversion of data types. It can also be used to cast a base class pointer into a derived class pointer. Its general form is:

```
static_cast<type>(object)
```

Here, *type* specifies the target type of the cast, and *object* is the object being cast into the new type. Examples:

```
int m = 10;
double x = static_cast<double> (m);
char ch = static_cast<char> (m);
```

Hidden page

The *type* must be a pointer or a reference to a defined class type. The *argument* object must be expression that resolves to a pointer or reference. The use of the operator **dynamic_cast()** is also called a *type-safe downcast*.

The typeid Operator

We can use the **typeid** operator to obtain the types of unknown objects, such as their class name at runtime. For example, the statement

```
char *objectType = typeid(object).name();
```

will assign the type of "object" to the character array **objectType** which can be printed out, if necessary. To do this, it uses the **name()** member function of the **type_info** class. The object may be of type **int**, **float**, etc. or of any class.

We must include **<typeinfo>** header file to use the operators **dynamic_cast** and **typeid** which provide run-time type information (RTTI).

16.4 Class Implementation

ANSI C++ Standard adds two unusual keywords, **explicit** and **mutable**, for use with class members.

The explicit Keyword

The **explicit** keyword is used to declare class constructors to be "explicit" constructors. We have seen earlier, while discussing constructors, that any constructor called with one argument performs *implicit conversion* in which the type received by the constructor is converted to an object of the class in which the constructor is defined. Since the conversion is automatic, we need not apply any casting. In case, we do not want such automatic conversion to take place, we may do so by declaring the one-argument constructor as explicit as shown below:

```
class ABC
{
      int m;
  public:
      explicit ABC (int i)      // constructor
      {
            m = i;
      }
      // ..............
      // ..............
};
```

Here, objects of ABC class can be created using only the following form:

```
ABC abc1(100);
```

The automatic conversion form

```
ABC abc1 = 100;
```

is not allowed and illegal. Remember, this form is permitted when the keyword **explicit** is not applied to the conversion.

The mutable Keyword

We know that a class object or a member function may be declared as **const** thus making their member data not modifiable. However, a situation may arise where we want to create a **const** object (or function) but we would like to modify a particular data item only. In such situations we can make that particular data item modifiable by declaring the item as **mutable**. Example:

```
mutable int m;
```

Although a function(or class) that contains **m** is declared **const**, the value of **m** may be modified. Program 16.2 demonstrates the use of a **mutable** member.

USE OF KEYWORD mutable

```
#include <iostream>
using namespace std;

class ABC
{
        private:
                mutable int m;   // mutable member
        public:
                explicit ABC(int x = 0)
                {
                        m = x;
                }
                void change() const     // const function
                {
                        m = m+10;
                }
                int display() const     // const function
                {
                        return m;
                }
};
```

(Contd)

```
int main()
{
        const ABC abc(100);              // const object
        cout << "abc contains: " << abc.display();

        abc.change();                    // changes mutable data
        cout << "\nabc now contains: " << abc.display();
        cout << "\n";

        return 0;
}
```

PROGRAM 16.2

The output of Program 16.2 would be:

```
abc contains: 100
abc now contains: 110
```

note

Although the function **change()** has been declared constant, the value of **m** has been modified. Try to execute the program after deleting the keyword **mutable** in the program.

16.5 Namespace Scope

We have been defining variables in different scopes in C++ programs, such as classes, functions, blocks, etc. ANSI C++ Standard has added a new keyword **namespace** to define a scope that could hold global identifiers. The best example of namespace scope is the C++ Standard Library. All classes, functions and templates are declared within the namespace named **std**. That is why we have been using the directive

```
using namespace std;
```

in our programs that uses the standard library. The **using namespace** statement specifies that the members defined in **std** namespace will be used frequently throughout the program.

Defining a Namespace

We can define our own namespaces in our programs. The syntax for defining a namespace is similar to the syntax for defining a class. The general form of namespace is:

```
namespace namespace_name
{
        // Declaration of
        // variables, functions, classes, etc.
}
```

note

There is one difference between a class definition and a namespace definition. The namespace is concluded with a closing brace but no terminating semicolon.

Example:

```
namespace TestSpace
{
      int m;
      void display(int n)
     {
           cout << n;
     }
}          // No semicolon here
```

Here, the variable **m** and the function **display** are inside the scope defined by the **TestSpace** namespace. If we want to assign a value to m, we must use the scope resolution operator as shown below.

```
TestSpace::m = 100;
```

Note that **m** is qualified using the namespace name.

This approach becomes cumbersome if the members of a namespace are frequently used. In such cases, we can use a **using** directive to simplify their access. This can be done in two ways:

```
using namespace namespace_name;        // using directive

using namespace_name::member_name;          // using declaration
```

In the first form, all the members declared within the specified namespace may be accessed without using qualification. In the second form, we can access only the specified member in the program. Example:

```
using namespace TestSpace;
m = 100;          // OK
display(200);     // OK

using TestSpace::m;
m = 100;          // OK
display(200);     // Not ok, display not visible
```

Nesting of Namespaces

A namespace can be nested within another namespace. Example:

```
namespace NS1
{
```

Hidden page

```
    }
}

namespace                          // Unnamed namespace
{
    int m = 200;
}

int main()
{
    cout << "x = " << Name1::x << "\n";          // x is qualified
    cout << "m = " << Name1::m << "\n";
    cout << "y = " << Name1::Name2::y << "\n";   // y is fully qualified
    cout << "m = " << m << "\n";                 // m is global

    return 0;
}
```

PROGRAM 16.3

The output of Program 16.3 is:

```
x = 4.56
m = 100
y = 1.23
m = 200
```

note

We have used the variable **m** in two different scopes.

Program 16.4 shows the application of both the **using** directive and **using** declaration.

ILLUSTRATING THE using KEYWORD

```
#include <iostream>

using namespace std;

// Defining a namespace
namespace Name1
{
    double x = 4.56;
    int m = 100;

    namespace Name2                    // Nesting namespaces
{
```

(Contd)

Hidden page

```
        int divide(int x,int y)    // definition
        {
            return(x/y);
        }

        int prod(int x,int y);     // declaration only
}

int Functions::prod(int x,int y)    // qualified
{
        return(x*y);
}

int main()
{
        using namespace Functions;

        cout << "Division: " << divide(20,10) << "\n";
        cout << "Multiplication: " << prod(20,10) << "\n";

        return 0;
}
```

PROGRAM 16.5

The output of Program 16.5 would be:

```
Division: 2
Multiplication: 200
```

note

When a function that is declared inside a namespace is defined outside, it should be qualified.

Program 16.6 demonstrates the use of classes inside a namespace.

USING CLASSES IN NAMESPACE SCOPE

```
include <iostream>

using namespace std;

namespace Classes
{
      class Test
{
```

(Contd)

```
        private:
            int m;

        public:

            Test(int a)
            {
                m = a;
            }

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

int main()
{
    // using scope resolution
    Classes::Test T1(200);
    T1.display();

    // using directive
    using namespace Classes;
    Test T2(400);
    T2.display();

    return 0;
}
```

PROGRAM 16.6

The output of Program 16.6 would be:

```
m = 200
m = 400
```

16.6 Operator Keywords

The ANSI C++ Standard proposes keywords for several C++ operators. These keywords, listed in Table 16.1, can be used in place of operator symbols in expressions. For example, the expression

```
x > y && m != 100
```

may be written as

```
x > y and m not_eq 100
```

Operator keywords not only enhance the readability of logical expressions but are also useful in situations where keyboards do not support certain special characters such as &, ^ and ~.

Table 16.1 *Operator keywords*

Operator	*Operator keyword*	*Description*
&&	and	logical AND
||	or	logical OR
!	not	logical NOT
!=	not_eq	inequality
&	bitand	bitwise AND
|	bitor	bitwise inclusive OR
^	xor	bitwise exclusive OR
~	compl	bitwise complement
&=	and_eq	bitwise AND assignment
|=	or_eq	bitwise inclusive OR assignment
^=	xor_eq	bitwise exclusive OR assignment

16.7 New Keywords

ANSI C++ has added several new keywords to support the new features. Now, C++ contains 64 keywords, including **main**. They are listed in Table 16.2. The new keywords are boldfaced.

Table 16.2 *ANSI C++ keywords*

asm	*else*	*namespace*	*template*
auto	enum	new	this
bool	**explicit**	operator	throw
break	**export**	private	**true**
case	extern	protected	try
catch	**false**	public	typedef
char	float	register	**typeid**
class	for	**reinterpret_cast**	**typename**
const	friend	return	union
const_cast	goto	short	unsigned
continue	if	signed	**using**
default	inline	sizeof	virtual
delete	int	static	void
do	long	**static-cast**	volatile
double	main	struct	**wchar_t**
dynamic_cast	**mutable**	switch	while

16.8 New Headers

The ANSI C++ Standard has defined a new way to specify header files. They do not use **.h** extension to filenames. Example:

```
#include <iostream>
#include <fstream>
```

However, the traditional style **<iostream.h>**, **<fstream.h>**, etc. is still fully supported. Some old header files are renamed as shown below:

Old style	New style
<assert.h>	<cassert>
<ctype.h>	<cctype>
<float.h>	<cfloat>
<limits.h>	<climits>
<math.h>	<cmath>
<stdio.h>	<cstdio>
<stdlib.h>	<cstdlib>
<string.h>	<cstring>
<time.h>	<ctime>

SUMMARY

- ANSI C++ Standard committee has added many new features to the original C++ language specifications.
- Two new data types **bool** and **wchar_t** have been added to enhance the range of data types available in C++.
- The **bool** type can hold Boolean values, **true** and **false**.
- The **wchar_t** type is meant to hold 16-bit character literals.
- Four new cast operators have been added: static_cast, const_cast, reinterpret_cast and dynamic_cast.
- The **static_cast** operator is used for any standard conversion of data types.
- The **const_cast** operator may be used to explicitly change the **const** or **volatile** attributes of objects.
- We can change the data type of an object into a fundamentally different type using the **reinterpret_cast** operator.
- Casting of an object at run time can be achieved by the **dynamic_cast** operator.
- Another new operator known as **typeid** can provide us run time type information about objects.
- A constructor may be declared **explicit** to make the conversion explicit.
- We can make a data item of a **const** object or function modifiable by declaring it **mutable**.

Hidden page

- RTTI
- source type
- standard library
- static casts
- **static_cast**
- **std** namespace
- target type
- **true** value
- type casts
- **type_info** class
- type_safe casting
- **typeid**
- **typeinfo** header
- unnamed namespaces
- **using** declaration
- **using** directive
- **using namespace**
- **volatile**
- **wchar_t**
- wide_character literal
- **xor**
- **xor_eq**

Review Questions

16.1 *List the two data types added by the ANSI C++ standard committee.*

16.2 *What is the application of* ***bool*** *type variables?*

16.3 *What is the need for* ***wchar_t*** *character type?*

16.4 *List the new operators added by the ANSI C++ standard committee.*

16.5 *What is the application of* ***const_cast*** *operator?*

16.6 *Why do we need the operator* ***static_cast*** *while the old style cast does the same job?*

16.7 *How does the* ***reinterpret_cast*** *differ from the* ***static_cast****?*

16.8 *What is dynamic casting?. How is it achieved in C++?*

16.9 *What is* ***typeid*** *operator?. When is it used?*

16.10 *What is explicit conversion?. How is it achieved?*

16.11 *When do we use the keyword* ***mutable****?*

16.12 *What is a namespace conflict? How is it handled in C++?*

16.13 *How do we access the variables declared in a named namespace?*

16.14 *What is the difference between using the* ***using namespace*** *directive and using the* ***using*** *declaration for accessing* ***namespace*** *members?*

16.15 What is wrong with the following code segment?

```
const int m = 100;
int *ptr = &m;
```

16.16 What is the problem with the following statements?

```
const int m = 100;
double *ptr = const_cast<double*>(&m);
```

16.17 What will be the output of the following program?

```
#include<iostream.h>
class Person
{
          // ......
}
int main()
{
          Person John;
          cout << " John is a ";
          cout << typeid(John).name() << "\n";
}
```

16.18 What is wrong with the following namespace definition?

```
namespace Main
{
       int main()
{
            // ......
       }
}
```

Debugging Exercises

16.1 Identify the error in the following program.

```
#include <iostream>

class A
{
public:
      A()
      {
      }

      A(int i)
      {
      }
```

```
};

class B
{
public:
        B()
        {
        }

        explicit B(int)
        {
        }
};

void main()
{
        A a1=12;
        A a2;
        A a3=a1;

        B b1 = 12;
}
```

16.2 Identify the error in the following program.

```
#include <iostream.h>

class A
{
protected:
        int i;
public:
        A()
        {
             i = 10;
        }

        int getI()
        {
```

```
            return i;
        }
};

class B: public A
{
public:
        B()
        {
        }

        int getI()
        {
            return i + i;
        }
};

void main()
{
        A *a = new A();
        B *b = static_cast<B*>(a);
        cout << b->getI();
}
```

16.3 Identify the error in the following program.

```
#include <iostream.h>

namespace A
{
            int i;
            void dispI()
            {
                    cout << i;
            }
}

void main()
{
            namespace Inside
```

```
    {
        int insideI;
        void dispInsideI()
        {
            cout << insideI;
        }
    }

    A::i = 10;
    cout << A::i;
    A::dispI();

    Inside::insideI = 20;
    cout << Inside::insideI;
    Inside::dispInsideI();
}
```

Programming Exercises

16.1 *Write a program to demonstrate the use of* ***reinterpret_cast*** *operator.*

16.2 *Define a namespace named* ***Constants*** *that contains declarations of some constants. Write a program that uses the constants defined in the namespace* ***Constants****.*

17

Object-Oriented Systems Development

Key Concepts

- Software development components
- Procedure-oriented development tools
- Object-oriented paradigm
- OOP notations and graphs
- Data flow diagrams
- Object-oriented design
- Top-down decomposition
- System implementation
- Procedure-oriented paradigm
- Classic software development life cycle
- Fountain model
- Object-oriented analysis
- Textual analysis
- Class hierarchies
- Structured design
- Prototyping paradigm

17.1 Introduction

Software engineers have been trying various *tools, methods,* and *procedures* to control the process of software development in order to build high-quality software with improved productivity. The methods provide "how to 's" for building the software while the tools provide automated or semi-automated support for the methods. They are used in all the stages of software development process, namely, planning, analysis, design, development and maintenance. The software development procedures integrate the methods and tools together and enable rational and timely development of software systems (Fig.17.1). They provide guideines as to how to apply the methods and tools, how to produce the deliverables at each stage, what controls to apply, and what milestones to use to assess the progress.

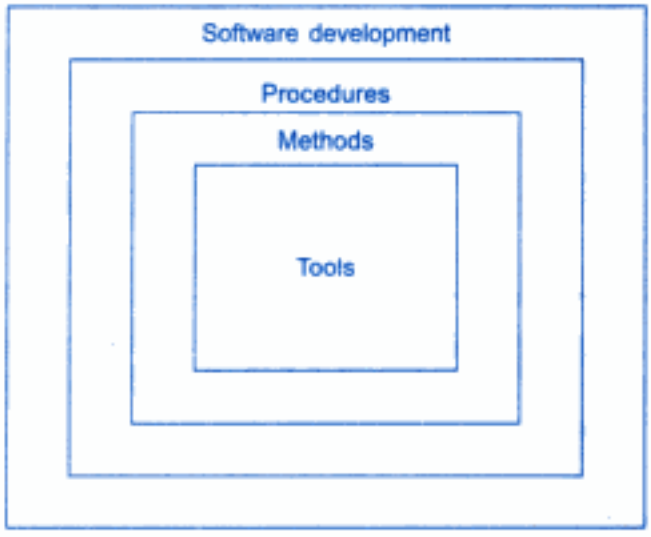


Fig. 17.1 ⇔ *Software development components*

There exist a number of software development paradigms, each using a different set of methods and tools. The selection of a particular paradigm depends on the nature of the application, the programming language used, and the controls and deliverables required. The development of a successful system depends not only on the use of the appropriate methods and techniques but also on the developer's commitment to the objectives of the system. A successful system must:

1. satisfy the user requirements,
2. be easy to understand by the users and operators,
3. be easy to operate,
4. have a good user interface,
5. be easy to modify,
6. be expandable,
7. have adequate security controls against misuse of data,
8. handle the errors and exceptions satisfactorily, and
9. be delivered on schedule within the budget.

In this chapter, we shall review some of the conventional approaches that are being widely used in software development and then discuss some of the current ideas that are applicable to the object-oriented software development.

17.2 Procedure-Oriented Paradigms

Software development is usually characterized by a series of stages depicting the various tasks involved in the development process. Figure 17.2 illustrates the classic software life cycle that is most widely used for the procedure-oriented development. The classic life cycle is based on an underlying model, commonly referred to as the "water-fall" model. This model attempts to break up the identifiable activities into series of actions, each of which must be

completed before the next begins. The activities include problem definition, requirement analysis, design, coding, testing, and maintenance. Further refinements to this model include iteration back to the previous stages in order to incorporate any changes or missing links.
Problem Definition: This activity requires a precise definition of the problem in user terms. A clear statement of the problem is crucial to the success of the software. It helps not only the developer but also the user to understand the problem better.

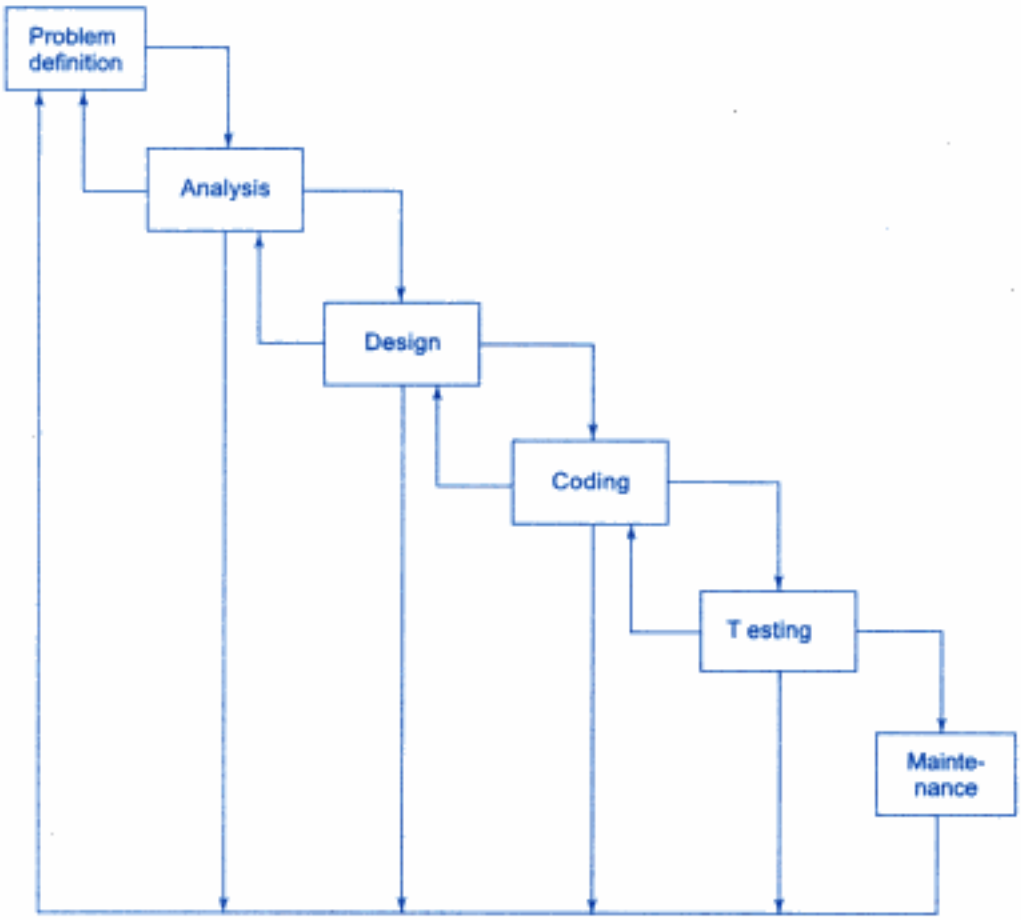


Fig. 17.2 ⇔ *Classic software development life cycle (Embedded 'water-fall' mode)*

Analysis: This covers a detailed study of the requirements of both the user and the software. This activity is basically concerned with what of the system such as

- what are the inputs to the system?
- what are the processes required?
- what are the outputs expected?
- what are the constraints?

Design: The design phase deals with various concepts of system design such as data structure, software architecture, and algorithms. This phase translates the requirements into a representation of the software. This stage answers the questions of *how*.

Coding: Coding refers to the translation of the design into machine-readable form. The more detailed the design, the easier is the coding, and better its reliability.

Testing: Once the code is written, it should be tested rigorously for correctness of the code and results. Testing may involve the individual units and the whole system. It requires a detailed plan as to what, when and how to test.

Maintenance: After the software has been installed, it may undergo some changes. This may occur due to a change in the user's requirement, a change in the operating environment, or an error in the software that has not been fixed during the testing. Maintenance ensures that these changes are incorporated wherever necessary.

Each phase of the life cycle has its own goals and outputs. The output of one phase acts as an input to the next phase. Table 17.1 shows typical outputs that could be generated for each phase of the life cycle.

Table 17.1 *Outputs of classic software life cycle*

Phase	***Output***
Problem definition (why)	• Problem statement sheet • Project request
Analysis (what)	• Requirements document • Feasibility report • Specifications document • Acceptance test criteria
Design (how)	• Design document • Test class design
Coding (how)	• Code document (program) • Test plan • User manual
Testing (what and how)	• Tested code • Test results • System manual
Maintenance	• Maintenance log sheets • Version documents

The software life cycle, as described above, is often implemented using the *functional decomposition technique*, popularly known as *top-down, modular* approach. The functional decomposition technique is based on the interpretation of the problem space and its translaion into the solution space as an inter-dependent set of functions. The functions are decomposed into a sequence of progressively simpler functions that are eventually implemented. The final system is seen as a set of functions that are organized in a top-down hierarchical structure.

There are several flaws in the top-down, functional decomposition approach. They include:

1. It does not allow evolutionary changes in the software.
2. The system is characterized by a single function at the top which is not always true. In fact many systems have no top.

3. Data is not given the importance that it deserves.
4. It does not encourage reusability of the code.

17.3 Procedure-Oriented Development Tools

A large number of tools are used in the analysis and design of the systems. It is important to note that the process of systems development has been undergoing changes over the years due to continuous changes in the computer technology. Consequently, there has been an evolution of new system development tools and techniques. These tools and techniques provide answers to the *how* questions of the system development.

The development tools available today may be classified as the *first generation, second generation*, and *third generation* tools. The first generation tools developed in the 1960's and 1970's are called the traditional tools. The second generation tools introduced in the late 1970's and early 1980's are meant for the structured systems analysis and design and therefore they are known as the structured tools. The recent tools are the third generation ones evolved since late 1980's to suit the *object-oriented* analysis and design.

Table 17.2 shows some of the popular tools used for various development processes under the three categories. Although this categorization is questionable, it gives a fair idea of the growth of the tools during the last three decades.

Table 17.2 *System development tools*

Process	*First generation*	*Second generation*	*Third generation*
Physical processes	System flowcharts	Context diagrams	Inheritance graphs Object-relationship charts
Data representation	Layout forms Grid charts	Data dictionary	Objects object dictionary
Logical processes	Playscript English narrative	Decision tables &trees Data flow diagrams	Inheritance graphs Data flow diagrams
Program representation	Program flowcharts I/O layouts	Structure charts Warnier /Orr diagrams	State change diagrams Ptech diagrams Coad/Yourdon charts

This section gives an overview of some of the most frequently used first and second generation tools. Object-oriented development tools will be discussed later in this chapter (as and when they are required).

System flowcharts: A graphical representation of the important inputs, outputs, and data flow among the key points in the system.
Program flowcharts: A graphical representation of the program logic.
Playscripts: A narrative description of executing a procedure.
Layout forms: A format designed for putting the input data or displaying results.
Grid charts: A chart showing the relationship between different modules of a system.
Context diagrams: A diagram showing the inputs and their sources and the outputs and their destinations. A context diagram basically outlines the system boundary.

Data flow diagrams: They describe the flow of data between the various components of a system. It is a network representation of the system which includes processes and data files.
Data dictionary: A structured repository of data about data. It contains a list of terms and their definitions for all the data items and data stores.
Structure chart: A graphical representation of the control logic of functions (modules) representing a system.
Decision table: A table of contingencies for defining a problem and the actions to be taken. It presents the logic that tells us what action to take when a given condition is true or otherwise.
Decision tree: A graphic representation of the conditions and outcomes that resemble the branches of a tree.
Warnier/Orr diagrams: A horizontal hierarchy chart using nested sets of braces, psuedo-codes, and logic symbols to indicate the program structure.

17.4 Object-Oriented Paradigm

The object-oriented paradigm draws heavily on the general systems theory as a conceptual background. A system can be viewed as a collection of *entities* that interact together to accomplish certain objectives (Fig. 17.3). Entities may represent physical objects such as equipment and people, and abstract concepts such as data files and functions. In object-oriented analysis, the entities are called *objects*.

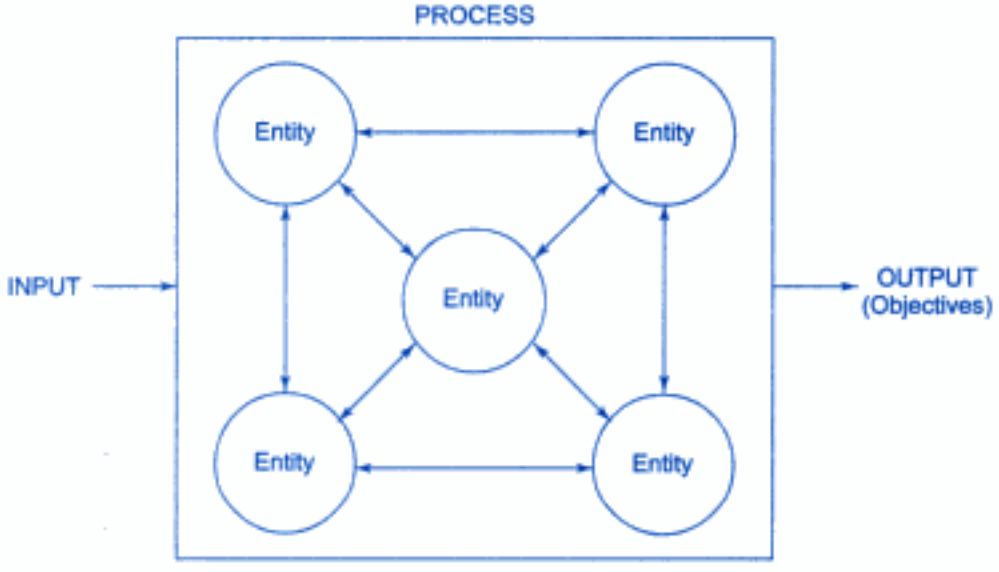


Fig. 17.3 ⇔ *A system showing inter-relationship of entities*

As the name indicates, the object-oriented paradigm places greater emphasis on the objects that encapsulate data and procedures. They play the central role in all the stages of the software development and, therefore, there exists a high degree of overlap and iteration between the stages. The entire development process becomes evolutionary in nature. Any

Hidden page

Hidden page

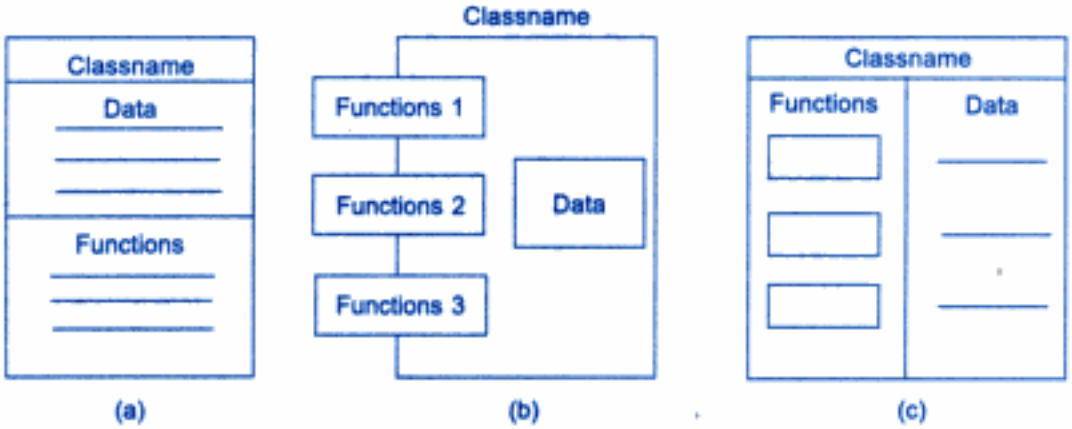


Fig. 17.6 ⇔ *Various forms of representation of classes/objects*

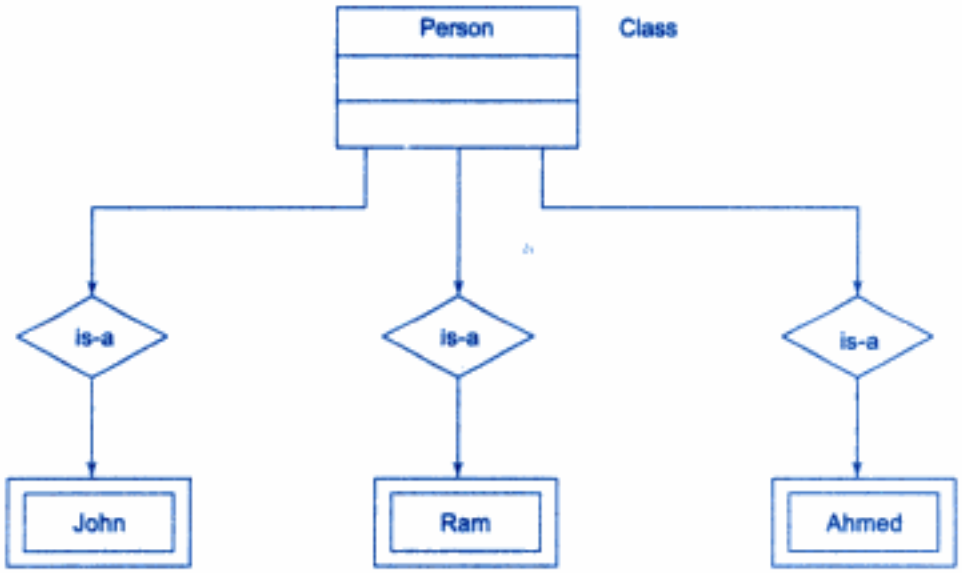


Fig. 17.7 ⇔ *Instances of objects*

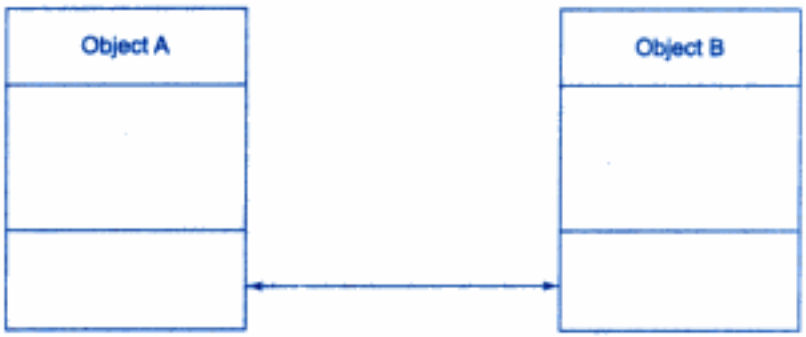


Fig. 17.8 ⇔ *Message communication between objects*

Hidden page

Hidden page

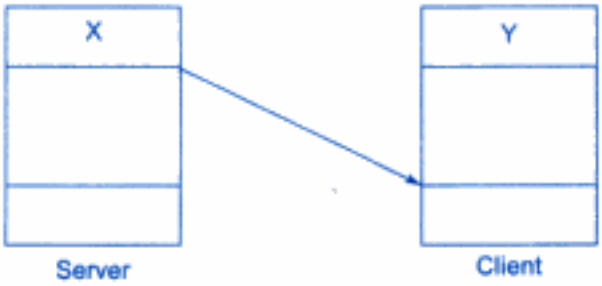


Fig. 17.13 ⇔ *Client-server relationship*

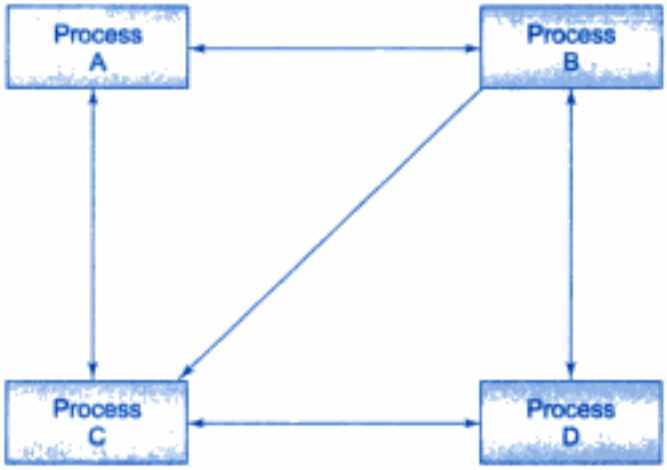


Fig. 17.14 ⇔ *Process layering (A process may have typically five to seven objects)*

17.6 Steps in Object-Oriented Analysis

Object-oriented analysis provides us with a simple, yet powerful, mechanism for identifying objects, the building block of the software to be developed. The analysis is basically concerned with the decomposition of a problem into its component parts and establishing a logical model to describe the system functions.

The object-oriented analysis (OOA) approach consists of the following steps:

1. Understanding the problem.
2. Drawing the specifications of requirements of the user and the software.
3. Identifying the objects and their attributes.
4. Identifying the services that each object is expected to provide (interface).
5. Establishing inter-connections (collaborations) between the objects in terms of services required and services rendered.

Although we have shown the above tasks as a series of discrete steps, the last three activities are carried out inter-dependently as shown in Fig. 17.15.

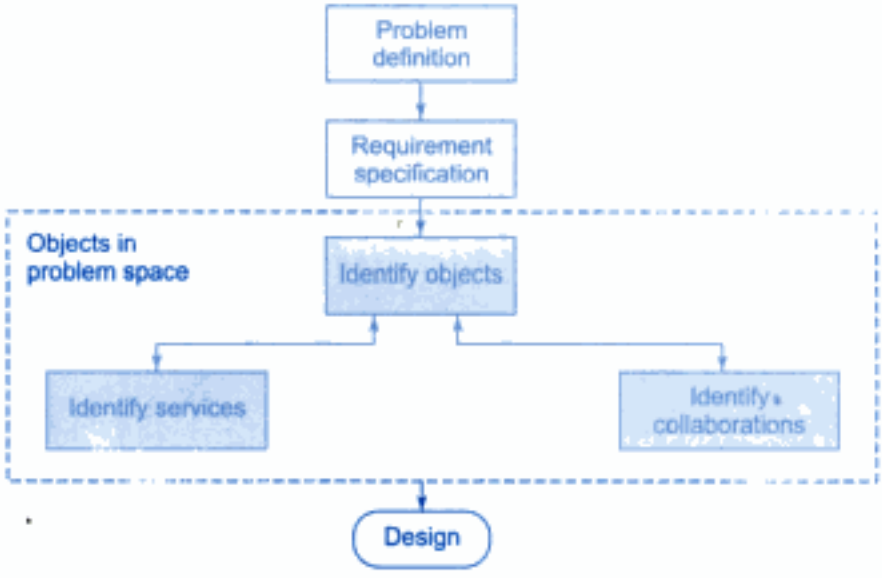


Fig. 17.15 ⇔ *Activities of object-oriented analysis*

Problem Understanding

The first step in the analysis process is to understand the problem of the user. The problem statement should be refined and redefined in terms of computer system engineering that could suggest a computer-based solution. The problem statement should be stated, as far as possible, in a single, grammatically correct sentence. This will enable the software engineers to have a highly focussed attention on the solution of the problem. The problem statement provides the basis for drawing the requirements specification of both the user and the software.

Requirements Specification

Once the problem is clearly defined, the next step is to understand what the proposed system is required to do. It is important at this stage to generate a list of user requirements. A clear understanding should exist between the user and the developer of what is required. Based on the user requirements, the specifications for the software should be drawn. The developer should state clearly

- What outputs are required.
- What processes are involved to produce these outputs.
- What inputs are necessary.
- What resources are required.

These specifications often serve as a reference to test the final product for its performance of the intended tasks.

Identification of Objects

Objects can often be identified in terms of the real-world objects as well as the abstract objects. Therefore, the best place to look for objects is the application itself. The application may be analyzed by using one of the following two approaches:

1. Data flow diagrams (DFD)
2. Textual analysis (TA)

Data Flow Diagram

The application can be represented in the form of a data flow diagram indicating how the data moves from one point to another in the system. The boxes and *data stores* in the data flow diagram are good candidates for the objects. The process *bubbles* correspond to the procedures. Figure 17.16 illustrates a typical data flow diagram. It is also known as a *data flow graph or a bubble chart*.

A DFD can be used to represent a system at any level of abstraction. For example, the DFD shown in Fig. 17.16 may be expanded to include more information (such as payment details) or condensed as illustrated in Fig. 17.17 to show only one bubble.

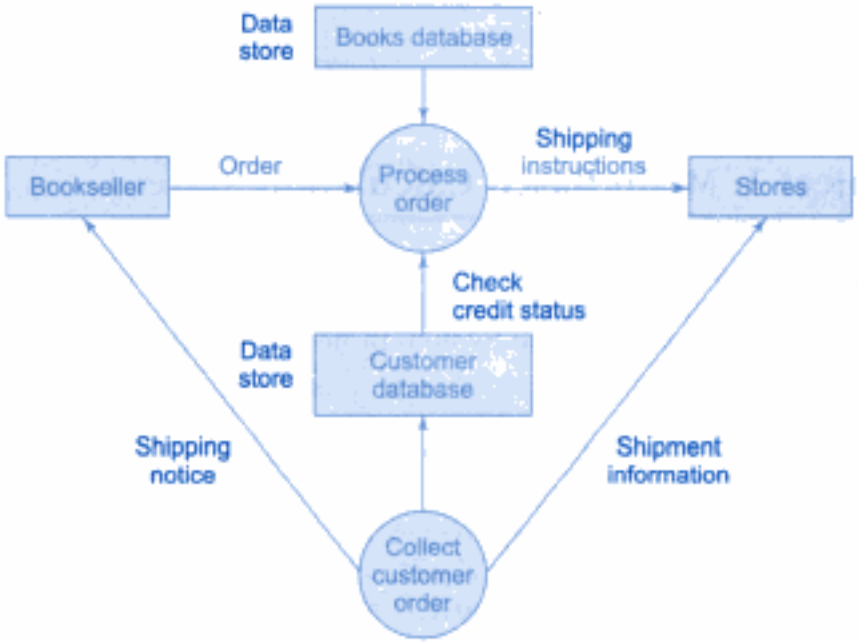


Fig. 17.16 ⇔ *Data flow diagram for order processing and shipping for a publishing company*

Fig. 17.17 ⇔ *Fundamental data flow diagram*

Textual Analysis

This approach is based on the textual description of the problem or proposed solution. The description may be of one or two sentences or one or two paragraphs depending on the type and complexity of the problem. The nouns are good indicators of the objects. The names can further be classified as *proper nouns, common nouns*, and *mass or abstract nouns*. Table 17.3 shows the various types of nouns and their meaning.

Table 17.3 *Types of nouns*

Type of noun	*Meaning*	*Example*
Common noun	Describe classes of things (entites)	Vehicle, customer income, deduction
Proper noun	Names of specific things	Maruti car, John, ABC company
Mass or abstract noun	Describe a quality, Quantity or an activity associated with a noun	Salary-income house-loan, feet, traffic

It is important to note that the context and semantics must be used to determine the noun categories. A particular word may mean a common noun in one context and a mass or abstract noun in another.

These approaches are only a guide and not the ultimate tools. Creative perception and intuition of the experienced developers play an important role in identifying the objects.

Using one of the above approaches, prepare a list of objects for the application problem. This might include the following tasks:

1. Prepare an object table.
2. Identify the objects that belong to the solution space and those which belong to the problem space only. The problem space objects are outside the software boundary.
3. Identify the attributes of the solution space objects.

Remember that not all the nouns will be of interest to the final realization of the solution. Consider the following requirement statements of a system:

Identification of Services

Once the objects in the solution space have been identified, the next step is to identify a set of services that each object should offer. Services are identified by examining all the verbs and verb phrases in the problem description statement. Verbs which can note actions or occurrences may be classified as shown in Table 17.4.

Doing verbs and *compare verbs* usually give rise to services (which we call as functions in C++). *Being verbs* indicate the existence of the classification structure while *having* verbs give rise to the composition structures.

Hidden page

Hidden page

The knowledge of such relationships is important to the design of a program.

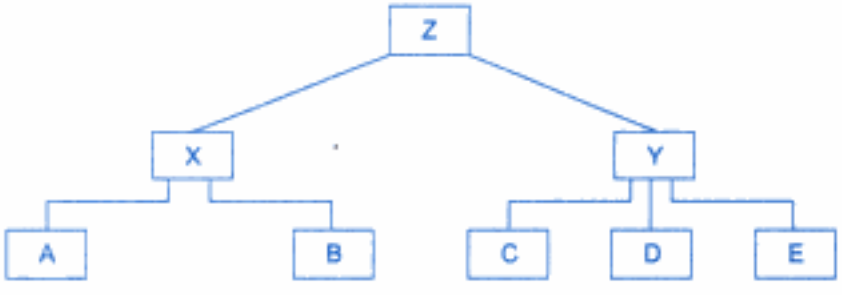
Organization of Class Hierarchies

In the previous step, we examined the inheritance relationships. We must re-examine them and create a class hierarchy so that we can reuse as much data and/or functions that have been designed already. Organization of the class hierarchies involves identification of common attributes and functions among a group of related classes and then combining them to form a new class. The new class will serve as the super class and the others as subordinate classes (which derive attributes from the super class). The new class may or may not have the meaning of an object by itself. If the object is created purely to combine the common attributes, it is called an *abstract class*.

This process may be repeated at different levels of abstraction with the sole objective of extending the classes. As hierarchy structure becomes progressively higher, the amount of specification and implementation inherited by the lower level classes increases. We may repeat the process until we are sure that no new class can be formed. Figure 17.18 illustrates a two-level iteration process.

(a) Objects in solution space

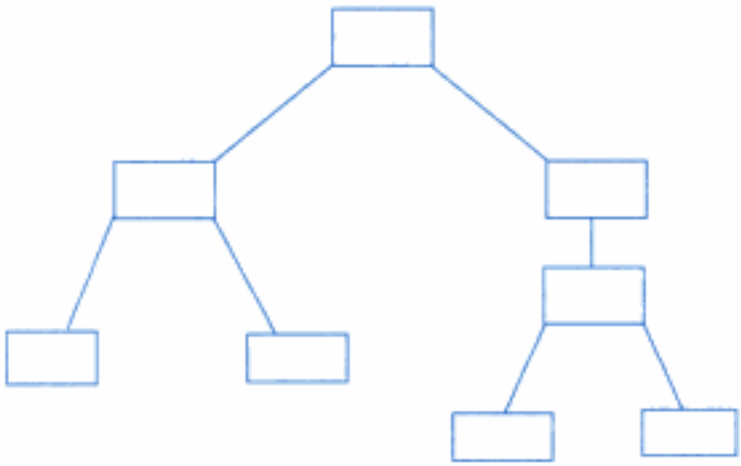
(b) First level of hierarchy



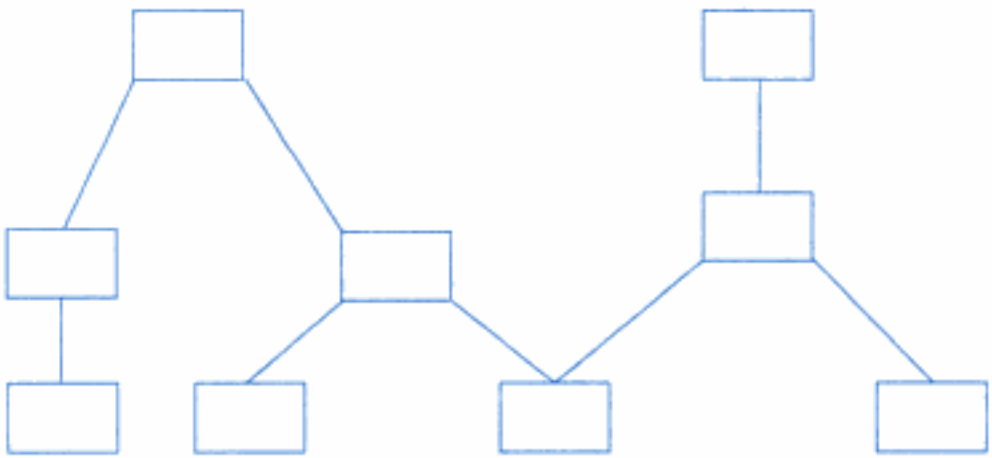
(c) Second level of hierarchy

Fig. 17.18 ⇔ *Level of class hierarchies*

The process of a class organization may finally result in a *single-tree model* as shown in Fig. 17.19(a) or *forest model* as shown in Fig. 17.19(b).



(a) Single-tree model



(b) Forest model

Fig. 17.19 ⇔ *Organisation of classes*

Design of Classes

We have identified classes, their attributes, and *minimal* set of operations required by the concept a class is representing. Now we must look at the complete details that each class represents. The important issue is to decide what functions are to be provided. For a class to be useful, it must contain the following functions, in addition to the service functions:

Hidden page

Hidden page

Hidden page

The driver program is the gateway to the users. Therefore, the design of user-system interface (USI) should be given due consideration in the design of the driver program. The system should be designed to be user-friendly so that users can operate in a natural and comfortable way.

17.8 Implementation

Implementation includes coding and testing. Coding includes writing codes for classes, member functions and the **main** program that acts as a driver in the program. Coding becomes easy once a detailed design has been done with care.

No program works correctly the first time. So testing the program before using is an essential part of the software development process. A detailed test plan should be drawn as to what, when and how to test. The class interfaces and class dependencies are important aspects for testing. The final goal of testing is to see that the system performs its intended job satisfactorily.

17.9 Prototyping Paradigm

Most often the real-world application problems are complex in nature and therefore the structure of the system becomes too large to work out the precise requirements at the beginning. Some particulars become known and clear only when we build and test the system. After a large system is completed, incorporation of any feature that has been identified as "missing" at the testing or application stage might be too expensive and time consuming. One way of understanding the system design and its ramifications before a complete system is built is to build and test a working model of the proposed system. The model system is popularly known as a *prototype*, and the process is called *prototyping*. Since the object-oriented analysis and design approach is evolutionary, it is best suited for *prototyping paradigm* which is illustrated in Fig. 17.22.

A prototype is a scaled down version of the system and may not have stringent performance criteria and resource requirements. Developer and customer agree upon certain "outline specifications" of the system and a prototype design is proposed with the outline requirements and available resources. The prototype is built and evaluated. The major interest is not in the prototype itself but in its performance which is used to refine the requirement specifications. Prototypes provide an opportunity to experiment and analyze various aspects of the system such as system structure, internal design, hardware requirements and the final system requirements. The benefits of using the prototype approach are:

- We can produce understandable specifications which are correct and complete as far as possible.
- The user can understand what is being offered.
- Maintenance changes that are required when a system is installed, are minimized.
- Development engineers can work from a set of specifications which have been tested and approved.

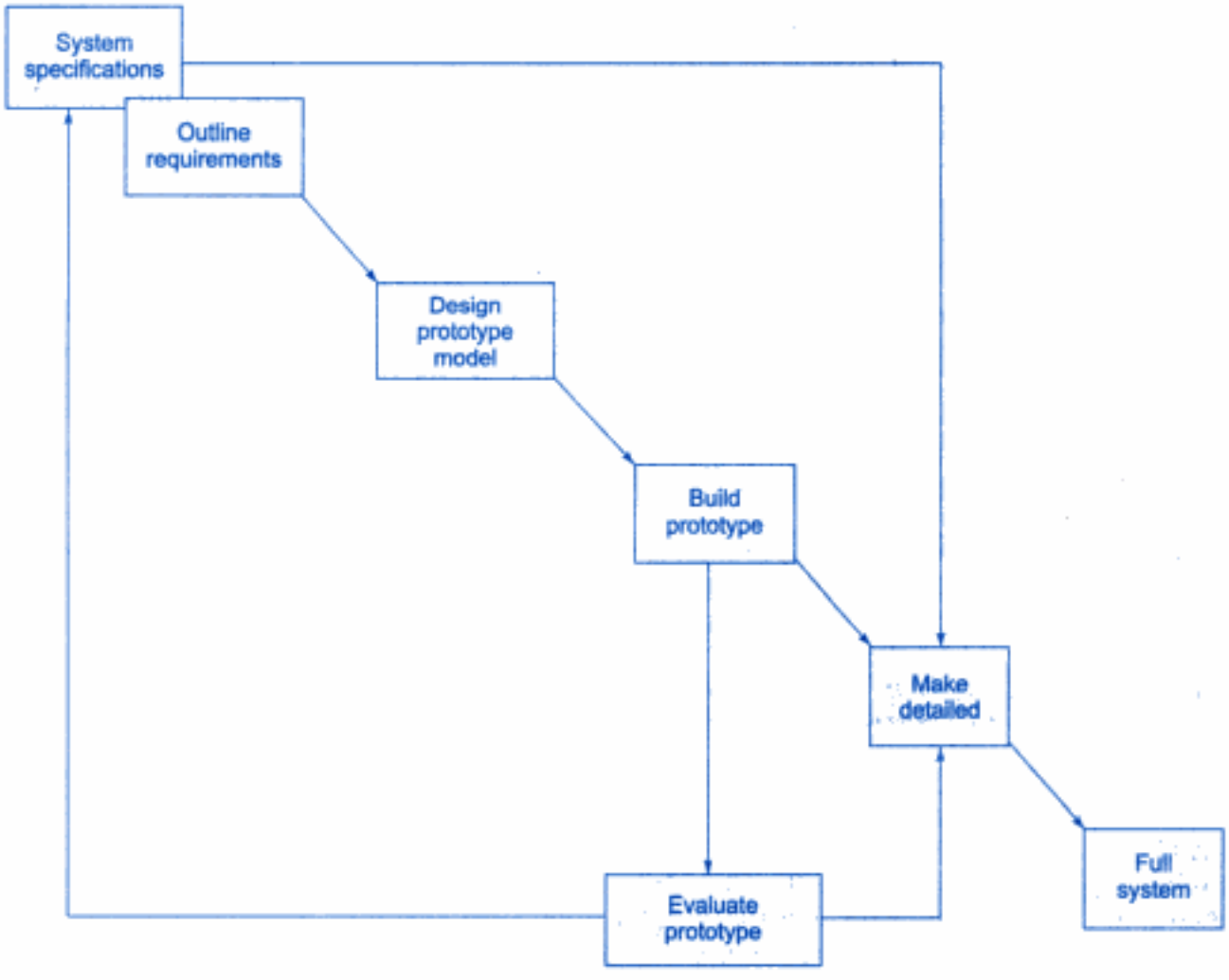


Fig. 17.22 ⇔ *Prototype paradigm*

Prototype is meant for experimenting. Most often it cannot be tuned into a product. However, occasionally, it may be possible to tune a prototype into a final product if proper care is taken in redesigning the prototype. The best approach is to throw away the prototype after use.

17.10 Wrapping Up

We have discussed various aspects of the object-oriented analysis and design. Remember, there is no one approach that is always right. You must consider the ideas presented here as only guidelines and use your experience, innovation and creativity wherever possible.

Following are some points for your thought and innovation:

1. Set clear goals and tangible objectives.
2. Try to use existing systems as examples or models to analyze your system.

3. Use classes to represent concepts.
4. Keep in mind that the proposed system must be flexible, portable, and extendable.
5. Keep a clear documentation of everything that goes into the system.
6. Try to reuse the existing functions and classes.
7. Keep functions strongly typed wherever possible.
8. Use prototypes wherever possible.
9. Match design and programming style.
10. Keep the system clean, simple, small and efficient as far as possible.

SUMMARY

- The classic system development life cycle most widely used for procedure oriented development consists of following steps.
  - Problem definition
  - Analysis
  - Design
  - Coding
  - Testing
  - Maintenance
- In object oriented paradigm, a system can be viewed as a collection of entities that interact together to accomplish certain objectives.
- In object oriented analysis, the entities are called objects. Object oriented analysis (OOA) refers to the methods of specifying requirements of the software in terms of real world objects, their behaviour and their interactions with each other.
- Object oriented design (OOD) translates the software requirements into specifications for objects, and derives class hierarchies from which the objects can be created.
- Object oriented programming (OOP) refers to the implementation of the program using objects, with the help of object oriented programming language such as C++.
- The object oriented analysis (OOA) approach consists of the following steps:
  - Defining the problem.
  - Estimating requirements of the user and the software.
  - Identifying the objects and their attributes.
  - Identifying the interface services that each object is supposed to provide.
  - Establishing interconnections between the objects in terms of services required and services rendered.
- The object oriented design (OOD) approach involves the following steps:
  - Review of objects created in the analysis phase.
  - Specification of class dependencies.

Hidden page

- data dictionary
- data flow diagrams
- decision table
- decision tree
- design
- development tools
- doing verbs
- driver program
- entities
- entity relationship diagram
- entity-relationship
- fist generation
- flowcharts
- forest model
- fountain model
- functional decomposition
- grid charts
- has-a relationship
- having verbs
- hierarchical chart
- information flow diagram
- inheritance relationship
- instances of objects
- is-a relationship
- layout forms
- proper nouns
- prototype
- prototyping
- prototyping paradigm
- second generation
- selection
- sequence
- single-tree model
- software life cycle
- solution space
- stative verbs
- structure chart
- structured design
- structured tools
- system flowcharts
- testing
- textual analysis
- third generation
- tools
- top-down approach
- traditional tools
- use relationship
- Warnier diagrams
- water-fall model

Review Questions

17.1 *List five most important features, in your opinion, that a software developer should keep in mind while designing a system.*

17.2 *Describe why the testing of software is important.*

17.3 *What do you mean by maintenance of software? How and when is it done?*

17.4 *Who are the major players in each stage of the systems development life cycle?*

17.5 *Is it necessary to study the existing system during the analysis stage? If yes, why? If no, why not?*

17.6 *What are the limitations of the classic software development life cycle?*

17.7 *"Software development process is an iterative process". Discuss.*

Hidden page