UNIT 1 STREAMS AND FILES

Structure **Page Nos.**

1.0	Introduction	5
1.1	Objectives	5
1.2	C++ Streams and Stream Classes	6
1.3	Unformatted I/O	8
1.4	Formatting Outputs	12
1.5	File Stream Operations	19
1.6	File Pointers and Operations	23
1.7	Summary	26
1.8	Answers to Check Your Progress	26
1.9	Further Readings	27

1.0 INTRODUCTION

In the previous blocks, we have discussed about the basic approach of object oriented programming and its important themes such as classes, inheritance, overloading, polymorphism and virtual functions. You might have seen a number of programming examples where you used **cin** and **cout** with operators >> and << for the input and output operations. We have, however, not taken up the issue of input and output to C++ programs formally. This was deliberately deferred to this point since I/O functions in C++ makes use of features like classes, derived classes and virtual functions. As we have already covered these topics, this unit builds further upon the preliminary introduction of I/O statements in C++ programs.

Through this unit, we aim to present a systematic and formal description of input and output mechanisms employed by C++ programs. C++ provides a rich set I/O functions and operations through the concept of streams and stream classes to implement I/O operations: both console and disk. The C++ I/O system contains a hierarchy of classes (known as stream classes) that are used for reading and writing by programs. Like many popular high level languages C++ supports two types of I/O: unformatted and formatted. While unformatted I/O uses functions like put(), get(), getline(), write(); formatted I/O makes use of manipulators and user-defined functions in addition to ios class functions and flags. We have also described the file stream operations and use of buffer and pointers for manipulating files.

1.1 OBJECTIVES

At the end of the unit, you should be able to:

- understand the basic mechanism of I/O in C++ through streams;
- distinguish between unformatted and formatted I/O operations in C++;
- use functions for unformatted I/O;
- design manipulators and user-defined functions for formatted I/O;
- understand stream classes for file manipulation;
- open and close files for various I/O operations;
- manage buffer and pointers for I/O from files; and
- obtain a thorough understanding and practice of using I/O functions in C++.

1.2 C++ STREAMS AND STREAM CLASSES

In C++ I/O occurs in streams, which are sequence of bytes. A stream acts as an interface between the program and the I/O device and can acts either as a source from which the input data can be obtained or as a destination to which the output data may be sent. In input operations, thy bytes flow from a device (e.g. a keyboard, a disk drive, a network connection, etc.) to main memory. In output operations bytes flow from main memory to a device (e.g., display monitor, a printer, a disk drive, a network connection, etc.). The stream that acts as a source is called input stream whereas the stream acting as destination is called output stream. The figure below illustrates the flow of data while using streams"

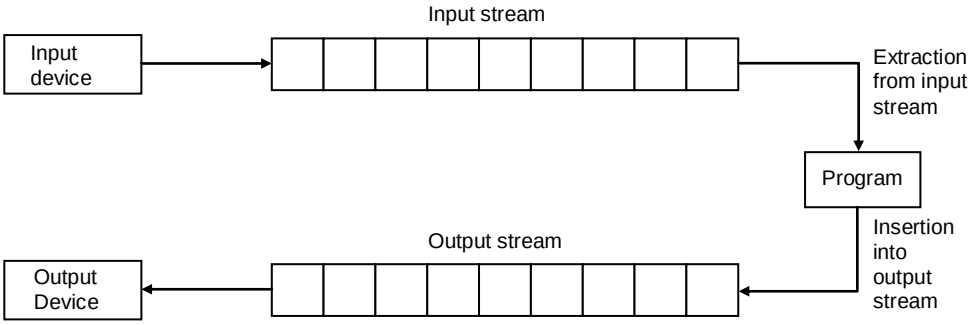

Figure 1.1 Data Streams

Though a program can take input and write output to a variety of devices (each of which may be quite different), the C++ streams provides a common interface for input and output operations irrespective of the device used. The bytes in the stream could represent characters, raw data, an image, video, speech or any other information that an application may require. It is the application (user program) that associates meaning with bytes.

In the past, the C++ used streams (often termed as classic streams) to enable input and output of characters. Since a character usually occupies one byte, it can represent only a limited set of characters (such as those in ASCII character set). Many languages however use alphabets that contain more alphabets than a single-byte character can represent. Hence C++ now includes the standard stream library that allows performing I/O operations with new character encoding systems such as Unicode. As we will shortly see, C++ now contains several pre-defined streams (e.g., cin, cout, cerr, etc.) that are automatically opened when a program begins its execution.

1.2.1 C++ Stream Classes

The C++ I/O system contains a hierarchy of classes which are used to define various streams to manage both console and disk I/O operations. These classes are called stream classes. The figure 1.2 given below shows the hierarchy of the stream classes used for input and output operations with the console. The stream class hierarchy for disk I/O is described in section 1.5

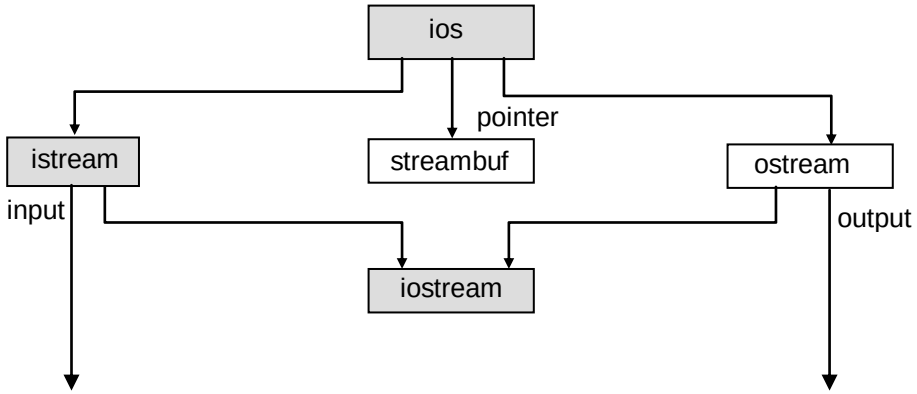

Figure 1.2 Stream Classes for Console I/O operations

The ios class is the base class for iostream (input stream) and ostream (output stream) which are in turn base classes for iostream class (input/ output stream). The ios class is declared as the virtual base class so that only one copy of its member are inherited by the iostream. The ios class thus provides the basic support for formatted and unformatted I/O operations. The class istream provides facilities for formatted and unformatted input while the class ostream provides facilities for formatted output. The class iostream provides facilities for handling both input and output operations. All these classes are declared in the header file iostream. Most of the C++ programs include the <iostream> header file, which declares all the basic services required for all stream-I/O operations. Relevant classes for formatted I/O and disk I/O are declared in <iomanip> and <fstream> header files respectively (discussed in later part of the chapter).

The standard streams- cin, cou, cerr and clog- are also defined in <iostream> header file.

1.2.2 Standard Stream Objects: cin, cout, cerr and clog

The cin, cout, cerr and clog objects correspond to the standard input stream, the standard output stream, the unbuffered standard error stream and the buffered standard error stream, respectively. These are predefined objects which are by default connected to certain devices. The extraction (>>) and insertion (<<) operators define the direction of data flow. For example, the first statement below is used to read a value from keyboard and pass it to the variable 'num'; whereas the second statement the value of variable 'avg' is written to the monitor (the standard output device).

```
cin >> num;  // reading the value of num from keyboard, data flow denotes input
cout << avg;  // displaying the value of avg on monitor, data flow denotes output
```

The number of characters read is determined by the type of variable. For example, if we type 3456X as the value for number, only '3456' is assigned to the variable 'num' and 'X' remains in the stream. Hence the operator >> reads the data character by character and assigns it to the indicated variable, unless a whitespace or a character that does not match with the destination variable type is encountered. At this point, extraction from stream is terminated and the remaining characters are consumed by subsequent cin statements. The nature of extraction (>>) and insertion (<<) operators therefore allows use of following kinds of cin and cout statements.

```
cin >> var1 >> var2 >> var3 >> ….>> varN
cout << var1 << var2 << var3 <<….<< varN
```

The variables in first statement may be variables of any type. The variables in second statement may be variables or constants of any type.

The predefined object **cerr** is an ostream instance and is connected by default to the standard error device (usually the monitor). Outputs to object **cerr** are *unbuffered*, implying that each stream insertion to **cerr** causes its output to appear immediately. (This is appropriate since it allows notifying a user promptly about errors). The predefined object **clog** is an instance of the ostream class also connected to the standard error device. However, outputs to **clog** are *buffered*. This means that each insertion to **clog** causes its ouput to be held in a buffer until the buffer is filled or it is flushed.

1.2.3 Types of I/O

The **ios** class provides the basic support for two kinds of input and output: formatted and unformatted. The class **istream** provides the facilities for formatted and unformatted input operations whereas the **ostream** class provides the facilities for formatted and unformatted output. The **istream** class declares input functions such as **get()**, **getline()**, **read()** etc., in addition to inheriting the properties of **ios** class. It also contains overloaded extraction operator >>. The **ostream** class declares output functions such as **put()** and **write()**, in addition to inheriting properties of **ios** class. It also contains overloaded insertion operator <<. The **iostream** class inherits the properties of **ios**, **istream** and **ostream** classes through multiple inheritance and hence contains all input and output functions. This is the reason why we include the iostream class only in our programs.

1.3 UNFORMATTED I/O

We have already seen use of cin and cout objects along with overloaded operators >> and << for input and output operations, in previous blocks. C++ provides many other functions for input and output operations. We will therefore briefly review their use through some examples. Functions put(), get(), getline(), putline(), read() and write() are other commonly used functions for unformatted I/O. The following sections describe, with appropriate examples, the use of these functions for unformatted I/O operations.

1.3.1 Overloaded Operators >> and <<

As stated in section 1.2.2 earlier, the overloaded **extraction (>>)** and **insertion (<<)** operators are used with **cin** and **cout** objects for stream input and output operations, respectively. The general syntax of its usage is as:

```
cin >> variable 1 >> variable 2 >> ……. >> variable N
cout << item 1 << item 2 << ……. << item N
```

where, variable 1 to N are any valid variable types in C++ and items 1 to N are any valid variable or constants in C++. C++ takes care of reading only appropriate number and type of variables (corresponding to the type of input variable) from the stream. The insertion operator puts the unformatted output into the output stream which is then displayed on the monitor. The use of these operators with **cin** and **cout** is further illustrated through following programming example:

```
# include <iostream.h>
int main()
{
        float num1, num2, num3, sum, avg;
        cout << "Enter the three numbers:";
        cin >> num1 >> num2 >> num3;
```

```
        sum = num1 + num2 +num3;
        avg = sum / 3;
        cout << "Sum of the numbers =" << sum;
        cout << "Average of the numbers =" << avg;
        return (0);
}
```

This program first displays a prompt for entering three numbers by using cout and then reads three numbers entered from the keyboard by using cin. The program then computes sum and average of the numbers and their values are displayed on the monitor through cout statements.

1.3.2 Using member functions get (), put (), getline (), ignore (), putback () and peek ()

Unformatted input and output operations can also be carried out using various member functions of cin and cout objects provided by the istream and ostream classes. The functions get() and put() can be used to handle single character input and output operations, respectively. The general usage syntax of get is as follows:

```
get (char *)
get (void)
```

The get (char *) function assigns the character read from the keyboard to its argument. Unlike extraction operator >>, it can read blanks spaces and newline characters. [Note that >> operator simply skips the blank spaces and newline character while extracting characters from the input stream.] The get (void) simply returns the character read from the keyboard without assigning it to any variable.

The put () member function can be used to write one character to the monitor. It may take a character constant or variable as argument. The put () function can also take an integer value as input (for ex. 65), however rather than displaying the integer value it displays its ASCII equivalent character, "A" for 65. It can be put in a loop to output a line of text character by character.

```
put ('char constant')
put (char variable)
```

The program below illustrates use of get () and put () functions. The program reads an input text and displays it on the output screen. It also counts the number of characters and displays it on output screen. For example, if the user enters "I love Programming", the output screen displays the entered text "I love Programming" and "Number of characters = 18" on separate lines on the output screen.

```
# include <iostream.h>
int main()
{
        int count = 0;
        char c;
        cout << "Enter some text:" ;
        cin.get(c);
        while (c!= '\n')
        {
                cout.put(c);
                count++;
                cin.get(c);
```

```
        }
        cout << "\n Number of characters = " << count << "\n";
        return (0);
}
```

The functions get() and put() can handle a single character at a time. Many practical situations however require us to read and display more than a single character, for example a line of text. The getline () member function can be used for this purpose. The general syntax of getline () function is as follows:

cin.getline (variable, size)

This function reads the character input into the variable. The reading is terminated as soon as size-1 characters are read or '\n' is encountered. The delimiter character '\n' however is discarded and not saved in the variable. For example, for the code segment:

```
char name [20];
cin.getline (name, 20)
```

if the text entered from the keyboard is "IGNOU <press RETURN>", it stores the line as "IGNOU" in the variable **name**. However, if we enter "Object Oriented Programming <press RETURN>", it stores only first 19 characters in the variable **name** as "Object Oriented Pro". Similar to get (), getline () can also read blank spaces.

The **ignore ()** member function reads and discards a designated number of characters (default = 1) or terminates upon encountering delimiter EOF. The **putback ()** member function places the character just read by get () back into the stream. The **peek ()** member function returns the next character from an input stream but does not remove the character from the stream.

1.3.3 Using member functions read (), write () and gcount ()

The other member functions used to perform unformatted I/O include read (), write () and gcount (). The member function read () inputs bytes to a character array in memory; member function write () write output bytes from character array; and member function gcount () reports the number of characters read by the last input operation. The general syntax of read () and write () functions are as follows:

```
cin.read (variable, size);
cout.write (variable, size)
```

The member function read () inputs a designated number of characters (max. = size) into a character array variable. It is different from getline () in terms of usage of array variable. The member function write () outputs the characters in the character array variable on the output screen. It however does not stops at delimiter (null char) boundary and continues to display characters even beyond the line (when size > length of array). The gcount () member function is used to report the number of characters actually read by the last read () operation. The program below demonstrates the use of read () and write () member functions. If the string entered from the keyboard is "I love Programming", it stores only "I love" in the input array variable. Function gcount () returns the value as 6.

```
# include <iostream.h>
int main()
{
        char buffer [80];
        cout << "Enter a line of text \n" ;
```

```
        cin.read(buffer, 20);
        cout.write (buffer, cin.gcount());
        return (0);
}
```

☞ Check Your Progress 1

1) Fill in the blanks:
   a) Input/ output in C++ occurs as ……of bytes.
   b) Most C++ program that do I/O should include the …… header file that contains the required for all stream I/O operations.
   c) The symbol for stream extraction operator is …….. .
   d) The four objects that correspond to the standard devices on the system include ……., ……., ……, and …… .
   e) Unformatted output member functions are provided in the class ……..

2) State whether following are *True* or *False*.
   a) The cin stream normally is connected to the display screen.
   b) Input with stream extraction operator >> always skips leading blank spaces in the input stream, by default.
   c) The ostream member function put () outputs the specified number of characters.

3) For each of the following, write a single statement that performs the indicated task:
   a) Output the string "Enter your name".

   ……

   ……

   ……

   ……

   ……

   b) Use istream member function read to input 50 characters into char array line.

   ……

   ……

   ……

   ……

   ……

   c) Get the value of next character to input without extracting it from the stream.

   ……

   ……

   ……

   ……

   ……

d) Use the istream member function gcount to determine the number of characters input into character array line by the last call to read and output that number of characters using write.

……

……

……

……

……

1.4 FORMATTING OUTPUTS

C++ provides a number of features that can be used display the outputs in a specified manner (formatted output). These features can be broadly categorized into following three types:

- **ios** class functions and flags
- Stream manipulators
- User-defined output functions

The **ios** class contains a large number of member functions that are used to format outputs in variety of ways. Most of the stream manipulators provide roughly the same features as that of **ios** class formatting functions, though they are at times convenient to use than their counterpart **ios** class functions. C++ allows users to design their own stream manipulators as well.

1.4.1 ios class format functions and flags

The **ios** class contains functions for defining field width, setting precision, filling and padding, displaying sign of numerical values etc. The Table 1.1 shows the lists of important **ios** class functions for formatting I/O:

Table1.1: List of important ios class functions for formatting I/O

Function	Purpose
width ()	To specify field size for displaying an output value
precision ()	To specify the number of digits to be displayed after the decimal sign
fill ()	To specify a character that fills the unused portion of an output data filed
setf()	To specify format flags such as left-justify, right-justify etc.
unsetf ()	To clear/ reset defined flags

Setting field width

The **width ()** function is used to define the width of a field required for displaying an output item. It is invoked by **cout** object as follows:

cout.width (w)

where **w** is the number of columns (field width). The output value is printed in a field of **w** characters wide at right end. This can specify field width for displaying only one

output value (the one that immediately follows it). The following statements demonstrate the use of width ():

```
cout.width (5);
cout << 123 << "\n";
cout.width (5);
cout << 35;
```

The output of the above statements would be displayed in a field width of 5 as follows:

		1	2	3
			3	5

However, C++ never truncates the display value if the specified width is smaller than required. In turn it expands field width to accommodate the output value. This feature can be used to print large number of numerical values in a predetermined manner (a situation that is often encountered in accounting and other commercial applications).

Setting Precision

In C++ the floating point numbers are by default printed with six digits after the decimal point. We can however change the number of digits to be displayed after the decimal sign by using precision () function. The syntax is as follows:

cout.precision (d)

where d is the number of digits to be displayed to the right of decimal point. For example, the statements:

```
cout.precision (3);
cout << sqrt (2) << "\n";
cout << 3.14159 << "\n";
cout << 1.50009 << "\n";
```

will produce the following output:

```
1.141   (truncated)
3.142   (rounded to nearest cent)
1.5     (no trailing zeros)
```

The **precision ()** function is different from **width ()** in its effect, as the effect of precision once set continues in all subsequent statements until it is reset. The **precision ()** function can be used with **width ()** function as illustrated in following example:

```
cout.precision(2);
cout.width(5);
cout << 5.6705;
```

produces following output:

	5		6	7

The statements print the output value with a precision of 2 digits after the decimal place within a field width of 5.

Filling and Padding

The **fill ()** function is used to fill unused portion of a display field with a desired character rather than the blank spaces printed by default. The syntax is:

cout.fill (ch);

where, ch is the character to be used to fill the unused portions of a display field. For example, the following statement:

```
cout.fill ('*');
cout.width(10);
cout << 1234 << "\n";
```

produces following output:

*	*	*	*	*	*	1	2	3	4

This kind of padding is very useful for institutions like banks which use it in their financial instruments (demand drafts etc) so that no one can change the figures. The **fill ()** function also stays in effect till it is reset.

Formatting Flags, Bit-fields and setf ()

The **setf ()** is another important **ios** class function that is commonly used for formatting outputs. The general syntax of **setf ()** is as follows:

cout.setf (arg1, arg2)

or

cout.setf (arg)

Here the **arg1** is one of the formatting flags defined in **ios** class and **arg2** is an **ios** constant that specifies the group to which the formatting flag belongs. These flags and bit-fields can be used for changing the alignment of display (left, right and center justify), displaying numbers in desired notation (scientific or fixed point), displaying numbers in different base systems (decimal, octal and hexadecimal). The setf () function can be used with single argument as well. In that case only flags are used without bit-fields.

The flags, bit-fields and their format actions are described in Table 1.2

Table1.2: Flags, bit-fields, formats

Format Required	Flag (arg1)	Bit-field (arg2)
Left justified output	ios::left	ios::adjustfield
Right justified output	ios::right	ios::adjustfield
Padding after sign or base indicator	ios::internal	ios::adjustfield
Scientific notation	ios::scientific	ios::floatfield
Fixed point notation	ios::fixed	ios::floatfield
Decimal base	ios::dec	ios::basefield
Octal base	ios::oct	ios::basefield
Hexadecimal base	ios::hex	ios::basefield

The **ios** flags that do not have bit-fields are described in Table 1.3:

Table 1.3: ios Flags

Flag	Purpose
ios::showbase	Use base indicator on output
ios::showpos	Print '+' before positive numbers
ios::showpoint	Show trailing decimal point and zeros
ios::uppercase	Use uppercase letters for hex output
ios::skipus	Skip blank space on input
ios::unitbuf	Flush all streams after insertion
ios::stdio	Flush stdout and stderr after insertion

These flags are used with **setf ()** to produce desired outputs. The following example code segments demonstrate use of some of these flags:

Example Code Segment 1:

```
cout.fill ('*');
cout.setf (ios::left, ios::adjustfield);
cout.width (10);
cout << "OUTPUT 1" << "\n";
```

will produce the following output:

O	U	T	P	U	T		1	*	*

Example Code Segment 2:

```
cout.fill ('#');
cout.precision (2);
cout.setf (ios::internal, ios::adjustfield);
cout.setf (ios::scientific, ios::floatfield);
cout.width (12);
cout << -32.234 << "\n";
```

will produce the following output: (Sign is left justified and value is right-justified)

-	#	#	3	2	.	2	3	e	+	0	1

Example Code Segment 3:

```
cout.setf (ios::showpoint);
cout.setf (ios::showpos);
cout.precision (3);
cout.setf (ios::fixed, ios::floatfield);
cout.setf (ios::internal, ios::adjustfield);
cout.width (10);
cout << 123.4 << "\n";
```

will produce the following output: (output with sign and trailing zeros)

+			1	2	3	.	4	0	0

1.4.2 Stream Manipulators

C++ defines a number of functions called manipulators in header file **iomanip** that can be used to manipulate the output formats. Most of these manipulators are quite similar in function to the **ios** class functions and flags, though at times they are more convenient to use. The best thing is that two or more manipulators can be used in a single statement as:

cout << manip1 << manip2 << manip3 << manip4 << item;

Some of the commonly used stream manipulators and their effect is listed in Table 1.4.

Table1.4: List of Used Stream Manipulators

Manipulator	Purpose
setw (int w) setprecision (int d)	Set the field width to w. Set the floating point precision to d.
setfill (int c)	Set the fill character to c.
setiosflags (long f)	Set the format flag f.
resetiosflags (long f)	Clear the format flag f.
endif	Insert a new line and flush stream.

You can easily notice that in terms of functionality, **setw ()** has similar effect to ios function **width ()**, **setprecision ()** to ios function **precision ()**, **setfill ()** to ios function **fill ()**, **setiosflag ()** to ios function **setf ()**, **resetiosflags ()** to ios function **unsetf ()** and **endif** to **"\n"**. There is however a major difference between implementation of **ios** functions and stream manipulators. The **ios** member functions return the previous format state which can be used later but the manipulator does not return to previous format state.

Some example code segments illustrating use of various stream manipulators and their outputs are given below:

Example Code Segment 1:

```
cout << setw (10) << 12345;
```

will produce the following output: (12345 is displayed right-justified in a field width of 10)

					1	2	3	4	5

Example Code Segment 2:

```
cout << setw (5) << setprecision (2) << 1.2345;
cout <<  setw (10) << setprecision (4) << 3.1415435;
```

will produce the following output: (Sign is left justified and value is right-justified)

	1	.	2	3

				3	.	1	4	1	5

1.4.3 User Defined Stream Manipulators

C++ allows users to design their own customized stream manipulators. These user-defined manipulators may be non-parameterized or parameterized ones. The general syntax for creating a user-defined stream manipulator without any argument is as follows:

```
Ostream & manipulator (ostream & output)
{
        ……
        …… Statements for formatting
        ……
        return output;
}
```

Here, the manipulator is the name of user-defined manipulator to be created. For example, we can create the user defined manipulator unit that displays "INR" after each numerical value displayed.

```
ostream & unit (ostream & output)
  {
    output << "INR";
    return output;
  }
```

A more complex user-defined manipulator show can be defined as follows:

```
ostream & show (ostream & output)
{
output.setf (ios::showpoint);
output.setf (ios::showpos);
output << setw (10);
return output;
}
```

The following program presents a full blown example of designing user-defined stream manipulators. This program creates two user-defined manipulators currency and form, which are used to display output values in a specific manner.

```
# include <iostream.h>
#include <iomanip.h>
ostream & currency(ostream & output)
{
        output << "INR";
        return (output);
}
ostream & form(ostream & output)
{
        output.setf (ios::showpos);
        output.setf (ios::showpoint);
        output.fill ('*');
        output.precision (2);
        output << setiosflags (ios::fixed) << setw (10);
        return (output);
}
```

```
int main()
{
        cout << currency << form << 5465.4;
        return (0);
}
```

The output of the program would be INR **+5465.40.

Another example code which makes use of stream manipulators and other formatting techniques is given below. This program creates two user-defined manipulators area and volume, which are used to display output values in a specific manner. The value of area is displayed with SQ.MTS unit and the volume is displayed with CUBIC MTS unit.

```
# include <iostream.h>
#include <iomanip.h>
ostream & area(ostream & output)
{
        output << "SQ.MTS";
        return (output);
}
ostream & volume(ostream & output)
{
        output.precision (4);
        output << setiosflags (ios::fixed) << setw (15);
        output << "CUBIC MTS"
        return (output);
}
int main()
{
        cout << area << 3000;
        cout<< volume << 9000;
        return (0);
}
```

☞ Check Your Progress 2

1) Fill in the blanks:
   a) Member function ……. can be used to set and reset format state.
   b) The stream manipulators that format justification are ……., ……., and …….. .
   c) The ……. stream manipulator causes positive number to be displayed with a plus sign.
   d) The functionally equivalent ios function for stream manipulator setw () is …….

2) State whether True or False:
   a) The stream manipulators dec, oct and hex affect only the next integer output operations.
   b) The stream member function flags with a long argument sets the flags state variable to its argument and returns its previous value.
   c) By default, memory addresses are displayed in hexadecimal format.

3) For each of the following, show the output:
   a) cout << setw (10) << setfill ('$') << 10000;

   ……

   ……

   ……

   ……

   b) cout << showbase << oct << 99 << endl << hex << 99;

   ……

   ……

   ……

   ……

   c) cout << 10000 << endl << showpos << 10000;

   ……

   ……

   ……

   ……

4) For each of the following, write a single C++ statement that performs the desired task.
   a) Use a stream manipulator such that, when integer values are output, the integer base for octal and hexadecimal values is displayed.

   ……

   ……

   ……

   b) Print the current precision setting, using a member function of object cout.

   ……

   ……

   ……

   c) Print 1234 right justified in a 10- digit field.

   ……

   ……

   ……

   d) Print 1.92, 1.925 and 1.9258 separated by tabs and with 3 digits of precision, using a stream manipulator.

   ……

   ……

   ……

5) Create a user-defined manipulator *showcurr* that prints '#' sign before every numerical value displayed.

……

……

……

1.5 FILE STREAM OPERATIONS

Many real world applications require reading and writing large number of data items. Usually large volumes of data are stored as files on disk. Like many high level programming languages, C++ also provides mechanism to read and write data items from files. A C++ program can thus take input from a disk file and also write data to it. C++ file handling mechanism is quite similar to console input-output operations. It uses streams (called file streams) as an interface between programs and files. The Figure 1.3 illustrates the use of file streams for input and output:

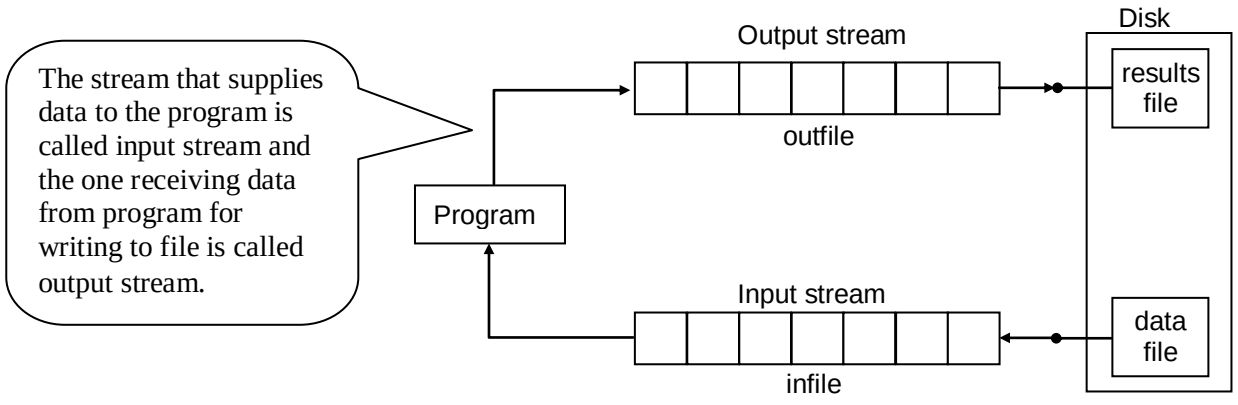

Figure 1.3 File Input and Output Steams

The Figure 1.3 shows a C++ program that reads data from one file and writes the output to another file (named results). The input stream is connected to data file used for input and output stream is connected to results file used for output. Programs can do reading and writing on the same file as well.

The I/O system of C++ contains a set of classes that provide methods for reading and writing from files. These classes are ifstream, ofstream and fstream. All these classes are derived from fstream base class and also inherits features from iostream class. The stream classes and their inheritance is shown in the Figure 1.4:

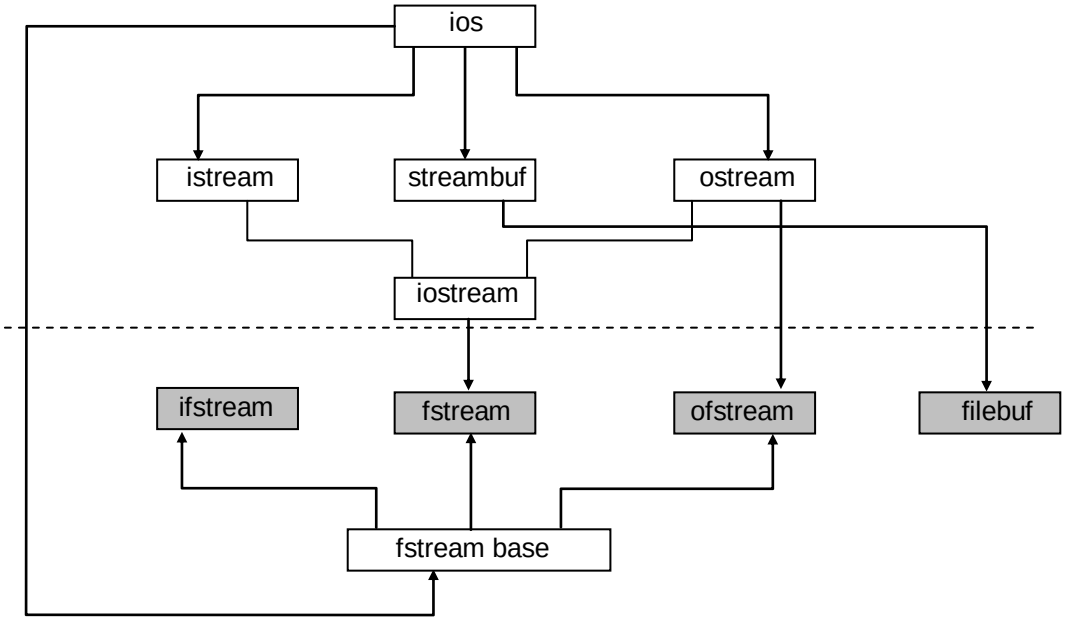

Figure 1.4 File Steam Classes

All these classes are declared in **fstream** and that is why this header file is included in all programs doing file processing. The **fstream** base class provides operations common to the file streams and contains **open()** and **close()** functions. It also serves as base class for the other three file stream classes. The **ifstream** class provides functions for input operations. This includes functions like **get()**, **getline()**, **read()**, **seekg()** etc. The **ofstream** class provides functions for output operations and include functions like **put()**, **write()** etc. The **fstream** class provides support for simultaneous input and output operations. It also inherits all functions from **istream** and **ostream** classes through **iostream**. The **filebuf** is used for setting the file buffers for read and write operations. This is required since volume of data is read and written to files.

Opening and Closing Files

Every disk file has a name (usually a string of valid characters). In order to read data from a file or to write data into it, we first need to open the file. This opened file has to be then connected to input-output stream so that the corresponding program becomes able read and write from it. A file stream can be defined using any of the three classes ifstream, ofstream or fstream, depending upon the purpose of the file.

A file can be opened either by using the **constructor** function of the class or by invoking member function **open ()** of the class. In order to open file using constructor function, we may use statements of the form:

```
ifstream infile (“data”); and
ofstream outfile (“results”);
```

Here, first statement opens a file called data and attaches it to the stream infile. This file can then be used for input operations. The second statement opens a file called results and attaches it to stream outfile. This file can only be used for output operations. Once the infile and outfile streams are created and connected to corresponding files, statements of the form:

```
outfile << "Sum";
outfile << "avg";
infile >> num;
infile >> name;
```

can be used to write data items to the output file results and to read data from input file data. After completing the input-output operations, the file may be closed by closing the corresponding stream. For example:

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

Even if we fail to explicitly close a file, it gets closed automatically when the program terminates (and hence the corresponding stream expires). Nevertheless it is a good practice to close the open files once they are no longer required.

A file can also be opened using **open ()**. The general syntax of open () is as follows:

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

For example, to open a file data for input, we may use the statements:

```
ifstream infile;
infile.open("data");
```

Now, the stream infile can be used for reading data form the file "data" in a similar manner as that of console I/O. After the file's use is complete, it should be closed by closing the corresponding (by invoking close() function).

C++ allows the files to be opened in different modes. Hence, the open() function can specify the mode of opening as well. The statement invoking open can be written as:

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

The mode argument specifies the purpose for which the file is opened. The file mode parameter is defined in ios class and can take any of the following values:

Mode Parameter	Effect
ios::app	Append to end-of-file.
ios::ate	Go to end-of-file on opening.
ios::binary	Open a binary file.
os::in	Open file for input only.
ios::nocreate	Open fails if file does not exist.
ios::noreplace	Open already existing file.
ios::out	Open file for output only.
ios::trunk	Delete the contents of the file if it exists.

The function open() contains default values for these modes and hence even if the mode is not specified, the file is opened with default mode.

One must also be careful in reading from an input file so as to ensure that no read attempt is made once the file end is reached. We will see the use of EOF() member function of ios class for this purpose in next section.

An example program demonstrating how we can read from a disk file and write the results to the disk file is given below:

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
int main()
{
        const int SIZE = 80;
        char line [SIZE];
        ifstream infile;
        ofstream outfile;
        infile.open("namelist");
        outfile.open("results");
        while(infile.eof()!=0)
        {
                infile.getline(line, SIZE);
                outfile << line;
        }
        return (0);
}
```

This program opens a file "namelist" for reading and "results" for writing. Then it continues to read the entire contents of file "namelist" and copies it to the file "results". The getline() stream member function is used for reading and the output content is simply directed to the stream "outfile" for writing it to the file "results". The reading (and subsequent copying) continues till the end of file "namelist" is reached. If one needs to display the contents while copying, a statement cout << line; may be added. This program can be made meaningful, if we copy the sorted namelist to the "results" file. This can be done by first sorting the contents of "namelist" file (temporarily storing it in an array of strings) and then writing the sorted values.

1.6 FILE POINTERS AND OPERATIONS

However, if we open a file in append mode, the output pointer is set to the end of file.

We can further control the reading and writing operations from file by manipulating the file pointers. Every file in C++ is marked by two pointers, one for input (called get pointer) and one used for output (called put pointer). The get pointer can be set to a specified location in order to reach from that desired location in the file. These pointers are automatically incremented every time after every read and write operation. You may be wondering how we have been able to read and write in our earlier programs without setting these pointers. Actually every time we open a file for input, the input pointer is automatically set to the beginning of the file.

We can manipulate these file pointers by invoking member functions of the file stream class. The commonly used functions for this purpose include **seekg()**, **seekp()**, **tellg()**, **tellp()** etc. The **seekg()** function moves the input (get) pointer to a specified location. The **seekp()** function moves to output (put) pointer to a specified location. The **tellg()** and **tellp()** functions tell the current position of get and put pointers, respectively. The general syntax for using **seekg()** and **seekp()** is given below:

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

Where, **offset** represents the number of bytes, the file pointer is to be moved from the location specified by the parameter refposition. The **refposition** may take one of the three positions defined in the **ios** class:

ios::beg – start of the file
ios::cur – current position of the pointer
ios::end – end of file.

The seekg() function moves the get pointer whereas the seekp() function moves the put pointer as per the parameters specified in the statement.

Functions to read from and write to files

Reading and writing operations in files are made simpler by C++ by providing some member functions in the file stream class. The functions **put()** and **get()** are used for handling a single character at a time. Functions **read()** and **write()** are designed to handle blocks of binary data. We also have functions like **getline().**

Two programming examples demonstrating use of **put()** – **get()** and **read() - write()** are given below:

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>
int main()
{
        char name [80];
        cin >> name;
        int len = strlen(name);
        fstream file;
        file.open("text", ios::in I ios:out);
        for (int i =0; I < len; i++)
                file.put(name[i]);
        file.seekg(0);
        char c;
        while (file)
        {
                file.get(ch);
                cout << ch;
        }
        return (0);
}
```

The program above reads a string and puts it character by character in a file called "text". This file is opened for both reading and writing. Then the program goes to read the contents written in the file one character at a time by using get() function. Before doing that, the file pointer is set to the beginning of the file.

```
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
const char * filename = "BINARY";
int main()
{
        int num[5] = {12, 23, 34, 45, 56};
        ofstream outfile;
        outfile.open(filename);
        outfile.write((char *) & num, sizeof(num));
        outfile.close();
        for (int i=0; i<5; i++)
```

```
                num[i]= 0;
        ifstream infile;
        infile.open(filename);
        infile.read((char *) & num, sizeof(num));
        for (i=0; i<5; i++)
        {
                cout.setf(ios::showpos);
                cout << setw(5) << name[i];
        }
        infile.close();
        return (0);
}
```

The program above reads an integer array and then writes its contents to a binary file using write(). The file is then opened again this time for reading and the contents written in the file are read and displayed on the monitor. The read() function is used for reading.

☞ Check Your Progress 3

1) Fill in the blanks:

   a) Header file …… contains the declarations required for file processing.
   b) The ios mode …….. only opens a file that already exists, otherwise it fails.
   c) The function……. can be used to position the input file pointer at a desired location.

2) State whether the following statements are True or False.

   a) The file opening mode ios::ate positions the file pointer at end-of-file on opening.
   b) The function tellp() can be used to detect end of a file.
   c) A C++ program can do reading and writing on the same file.

3) For each of the following, write a single statement that performs the desired task:

   a) To open a file "text" in append mode and connect a stream infile to it.

   ……

   ……

   ……

   b) To read a single character in a file connected to stream file.

   ……

   ……

   ……

c) To read an entire line of text of *size* characters from a file connected to f1 file stream into a variable *line*.

……

……

……

d) To detect the end of a file connected to stream file.

……

……

……

1.7 SUMMARY

This unit has introduced you to the concept of streams and how they are used for console and disk I/O operations by a C++ program. The stream concept provides in C++ the flexibility of using the same mechanism for input-output operations from different devices. A stream is a sequence of bytes and it serves as both source and destination for an I/O data. C++ provides a hierarchy of stream classes that support different functions for managing I/O devices from console and disk. There are certain streams which are by default connected to standard devices. The console I/O operations may be unformatted or formatted. We use separate functions for both purposes. Output can be produced in a desired manner (formatted) by using one of the three mechanisms: ios class functions and flags, stream manipulators and user-defined stream manipulator functions. C++ also provides a large number of functions that can be used to read and write data from disk files. Files need to be opened and closed for I/O operations. Files may be opened in different modes depending upon the kind of operation to be performed. C++ provide different functions for manipulating file pointers and hence deciding where to read and write the data.

1.8 ANSWERS TO CHECK YOR PROGRESS

Check Your Progress 1

1. a) streams b) <iostream> c) >> d) cin, cout, cerr, clog
   e) ostream
2. a) False, It is connected to standard input, which is keyboard.
   b) True
   (b) False, It outputs its single character argument.
3. a) cout << “Enter your name”;
   b) cin.read (line, 50);
   c) cin.peek ()
   d) cout.write (line, cin.gcount());

Check Your Progress 2

1. a) flags b) left, right and internal
   c) showpos d) width ()

2. a) False, They have effect till it is reset or the program terminates.
   b) False, The stream member function flags with a fmtflags argument sets the flags state variable to its argument and returns the prior state setting.

c) True

3. a) $$$$$10000
   b) o143
      oX63
   c) 10000
      +10000

4. a) cout << showbase;
   b) cout << cout.precision ();
   c) cout << setw (10) <<1234;
   d) cout << setprecision (3) << 1.92 << '\t' << 1.925 << '\t' <<1.9258

5.
```
ostream & showcurr (ostream & output)
{
output << "#";
return output;
}
```

Check Your Progress 3

1. a) <fstream> b) ios::noreplace c) seekg()

2. a) True
   b) False, It is actually eof() function.
   c) True
3. a) ifstream infile; infile.open("text", "ios::app");
   b) file.get(ch);
   c) f1.getline(line, size);
   d) file.eof()

1.9 FURTHER READINGS

1) E. Balaguruswamy, *Object Oriented Programming with C++*, Tata McGraw Hill, 2010.
2) P. Deitel and H. Deitel, *C++: How to Program, PHI,* 7th edition, 2010.
3) B. Strousstrup, *Programming – Principles and Practices using C++*, Addison Wesley, 2009.

UNIT 2 TEMPLATES AND STANDARD TEMPLATE LIBRARY

Structure **Page Nos.**

2.0	Introduction	28
2.1	Objectives	28
2.2	Class Templates	29
2.3	Function Templates	31
2.4	Use of Templates	34
2.5	The Standard Template Library	35
2.6	Summary	43
2.7	Answers to Check Your Progress	44
2.8	Further Readings	45

2.0 INTRODUCTION

In the previous units, we have discussed about different features and functionalities of C++ programming language for object oriented programming. C++ being an object oriented language also supports reuse. Templates are one of the most prominent examples of reuse concept in action. This feature has been added to the C++ design recently. It supports the idea of generic programming by providing facility for defining generic classes and functions. Thus a template class provides a broad architecture which can be used to create a number of new classes. Similarly, a template function can be used to write various versions of the function. This chapter describes the template mechanism in C++ based on the paper titled 'Parameterized types for C++' by Bjarne Stroustup (creator of C++) published in Proceedings of the USENIX C++ Conference held in Denver, Colorado in October 1988.

This unit aims to explain the basic idea and motivation for templates. The unit explains in detail (with appropriate examples) the design and use of both Class and Function Templates. It then proceeds further to explain the use of templates and how they are related to the concepts of overloading and inheritance. We then discuss the structure and utility of the Standard Template Library in C++. Further, the unit describes the three core components of the Standard Template Library: containers, algorithms and iterators. These features are used in many real world application designs. The unit concludes with a summary of the Template and Standard template Library framework followed by model answers to assist you in check your progress exercises.

2.1 OBJECTIVES

At the end of the unit, you should be able to:

- identify the concept of Templates in C++;
- understand the definition and usage of class and function templates;
- write your own class and function templates;
- understand the design of class and function templates with and without parameters;
- visualize the idea behind design of the Standard Template Library;
- describe the components of the Standard Template Library and understand their use; and

- appreciate the use of the Standard template Library in real world application designs.

2.2 CLASS TEMPLATES

You might have got at least an idea from the introduction that templates are like stencils out of which we trace shapes. Function-template specializations and class-template specializations are like the separate tracings that all have the same shape, but could, for example, be drawn in different colours. In other words, a template may be considered as a kind of macro. When the actual object of that type is to be defined, the template definition is substituted with required data type. For example, if we define a template Array of elements, then this same generic definition may be used to create Array of integers or of characters or float quantities. We need not make a new class definition every time. We define a generic class with a parameter that s replaced by a particular data type at the time of actual use of that class. This is the reason template classes are also known as parameterized classes.

Designing a template class thus involves a simple process of creating a generic class with an anonymous type. The general syntax for defining a class template is:

```
template <class T>
class classname
{
……..
…….class specification with anonymous type T
……..
};
```

For example, consider the following template definition for a **Vector** class:

```
template<class T>
class vector
{
  T * v;  // the vector is of type T
  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 &x)
  {
   T sum = 0;
   for(int i=0; i<size; i++)
       sum += this->v[i] *x- v[i];
   return (sum);
  }
};
```

This definition creates a template class *vector* of type T with variables, constructors and '*' operator. This class definition is similar to an ordinary class definition except that of the use of **template<class T>** and use of type **T** inside the class definition. The template<class T> tells the compiler that it is a template class with parameter T instead of a normal class definition. The declaration thus creates a parameterized class with T as the parameter, which can be substituted with any valid data type. This can be done by a statement of the following form:

- classname <type> objectname(argument list);

For example, following statements create classes of 20 element integer and float vectors, respectively.

- vector <int> v(20);
- vector <float> v(20);

This task of creating an actual object from a template class is instantiation. Here we have written only one class definition for class *vector* but we have been able to create more than one actual instantiations of the template class *vector*.

A class template can also be created by using multiple generic data types as arguments. The general syntax for such definition would be as below:

```
template <class T1, class T2, …….>
class classname
{
……..
…….class specification with anonymous type T
……..
};
```

A simple program demonstrating the declaration and use of class templates is given below. Please note that this program instantiates two objects from the class template Example. The program first declares a template class Example with two arguments and then declares a constructor to instantiate the class. The main function creates two objects test1 and test2 of different types and their values are then displayed using the function show.

```
#include <iostream>

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

```
int main()
{
  Example <float, int> test1 (3.45, 345);
  Example <int, char> test2 (100, 'm');
  test1.show();
  test2.show();
return(0);
};
```

The program creates two template classes test1 and test2 using the template class Example. The test1 class has two parameter values "3.45" and "345", whereas test2 class has two parameter values "100" and character "m". For creating test1 object, arguments are float and integer respectively, whereas in case of test2 object they are integer and character. The values displayed in invocation of show() function from main will be "3.45 and 345" for test1 and "100 and m" for test2 object.

2.3 FUNCTION TEMPLATES

Function templates are similar to class templates in the sense that they create a generic function type. This generic function can then be used to create a family of functions that may take different arguments. A function template can be defined as follows:

```
template <class T>
return_type function_name (arguments of type T)
{
……..
…….body of function with argument of type T
……..
};
```

Function templates are another way of handling overloaded function requirements. If overloaded functions perform identical operations for different type of data then they can be more appropriately and conveniently declared as function templates. The following example demonstrates creation of a function template swap:

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

The swap() function can now be invoked like any ordinary function. Any call to swap() with input arguments will exchange the values contained in those arguments. Hence if a and b are integer variables and p and q are float variables; we may invoke swap() function as swap(a,b) and swap(p,q), respectively. The same function definition can be used to swap values of two variables of different types.

As we have discussed earlier, we can define class templates with multiple arguments, we can also define function templates with multiple arguments. This can be done through a declaration of the form:

```
template <class T1, class T2, ……..>
return_type function_name (arguments of type T1, T2,….)
{
……..
…….body of function
……..
};
```

The function and class templates can be used to write programs which work correctly on different types of data. For example, we can write the following program to sort a list in desired order using function templates swap() and bsort() as shown in the program below:

```
 #include <iostream>

template<class T>
void bsort(T a[], int n)
{
  for (int i=0; i<n-1; i++)
    for (int j=n-1; i<j; j--)
        if (a[j] < a[j-1])
          swap(a[j], a[j-1]);
 }
template <class X>
void swap( X &a, X &b)
{
  X temp =a;
  a = b;
  b = temp;
}

int main()
{
  int x[5] = {10,50,30,60,40};
  float y[5] = {3.2, 71.5, 17.3, 45.9, 92.7};

bsort(x,5);
bsort(y,5);

cout << "Sorted X-Array:";
for (int i=0; i<5; i++)
   cout << x[i] << " ";
cout << endl;

cout << "Sorted Y-Array:";
for (int j=0; j<5; j++)
   cout << y[j] << " ";
cout << endl;
return(0);
};
```

This program uses two function templates swap() and bsort(). The function template swap() is invoked within the bsort() function and is hence said to be nested in it. This program can be used to sort different types of lists without the need of modifying the program. The program will produce following output:
Sorted X-Array: 10 30 40 50 60

Sorted Y-Array: 3.2 17.3 45.9 71.5 92.7

A template function may also be overloaded in a manner similar to other functions. In fact, function templates and overloading are intimately related. All the function-template specializations generated from a function template have the same name, so the compiler uses overloading resolution to invoke the proper function. A function template can be overloaded in several ways:

a) functions having same name but different parameters
b) providing a non-template functions with the same function name but different arguments

Whenever, the compiler has to perform the matching process to determine what function to call, it achieves the overloading resolution as follows:

a) call an ordinary function that has an exact match
b) call a template function that could be created with an exact match
c) try normal overloading resolution to ordinary functions and call the one that matches.

In case no match is found, the compiler generates an error. In case there are multiple matches for the function call, the compiler considers the call to be ambiguous and the compiler generates an error message. It would also be worth mentioning that no automatic conversions are applied to arguments on the template functions.

☞ Check Your Progress 1

1) Fill in the blanks:
   a) Templates enable us to specify, with a single code segment, an entire range of related functions called ……..……, or an entire range of related classes called…….…… .
   b) All function template definitions begin with the keyword…… followed by a list of template parameters to the function template enclosed in…….. ..
   c) Class templates are also called ……..…… types.
   d) The …….. operator is used with a class template name to tie each member function definition to the class template's scope.

2) State whether following are *True* or *False*.
   a) A function template can be overloaded by another function template with the same function name.
   b) Template parameter names along template definitions must be unique.
   c) Each member function definition outside a class template must begin with a template header.
   d) Keywords *typename* and *class* as used with a template type parameter specifically mean "any user-defined class type."

3) Write appropriate statements to create a function template *printarray* that can display the values contained in array passed as parameter to the function. The function must be able to accept integer, float and character arrays as arguments.

   ……

   ……

   ……

4) Write appropriate statements to create a template class item that can instantiate objects of at least following types: item name: shirt, measure: size (expressed as characters 'S', 'M', 'L' and 'X') and item name: trouser, measure: size (expressed as waist size of integers). The template class must also have a template function to display the details of the items.

……

……

……

2.4 USE OF TEMPLATES

The concept of class templates and function templates derives its motivation from the principle of reuse. We have seen in earlier sections how templates allow us to create generic data types and functions that can fit into various situations. Rather than defining multiple classes and functions, we define a generic type and depending on the kind of input data it may customize itself. Templates in this sense serve as a blueprint for defining classes and functions. This not only eliminates code duplication for handling different data types but also makes the program development easier and more manageable. In the previous section, we saw an example of program for sorting that can a sort lists of various types. We present below another example demonstrating use of templates for solving a quadratic equation:

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

template<class T>
void roots(T a, T b, T c)
{
  T d = b*b - 4*a*c;
  if (d==0)
   {
     cout << "R1 = R2 ="  << -b/(2*a) << endl;
   }
  else if (d > 0)
   {
      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
   {
      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 <<endl;
    }
 }

int main()
```

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

  return(0);
};
```

The program will generate following output:

```
Integer coefficients
Roots are real
R1 = 3 and R2 = 2
Float coefficients
Roots are complex
Real part = -1.2
Imaginary part = 1.375985
```

As we can see, the program above can be used to compute roots of a quadratic equation having different kinds of coefficients. It calculates roots for an equation having integer coefficient and for another equation having float coefficients. The templates have become so popular that now we have a rich library of predefined templates in C++, known as the Standard Template Library. We will discuss the contents and use of the standard template library in next section.

2.5 THE STANDARD TEMPLATE LIBRARY

The STL was developed by Alexander Stepanov and Meng Lee at Hewlett-Packard while pursuing their research in generic programming, with significant contributions from David Musser.

Recognizing that many data structures and algorithms are commonly used, the C++ standards committee added the Standard Template Library (STL) to the C++ standard library. The STL defines powerful, template-based, reusable components that implement many common data structures, and algorithms used to process those data structures. STL is a large collection of generic classes and functions. This large collection can be grouped at its core into three categories:

- Containers
- Algorithms, and
- Iterators.

These components work in conjunction with each other to provide solution to complex programming problems. A statement summarising their relationship could be "*Algorithms employ iterators to perform operations stored in containers*". The figure 2.1 further elaborates this relationship:

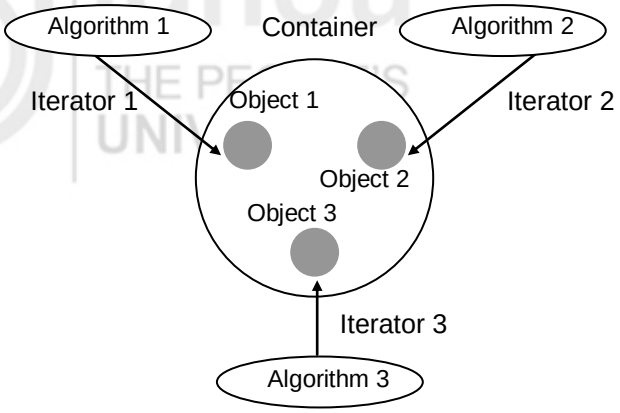

Figure 2.1 : Relationship between the three STL components

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 hence can be easily customized to hold different types of data. An *algorithm* is a procedure that is used to process the data stored in the containers. The STL include many different kinds of algorithms such as searching, sorting, copying, merging etc. Algorithms are implemented by template functions. An *iterator* is an object that works like a pointer. It is used to point to elements in a container. Iterator value may be incremented or decremented just like pointers. They play a key role in accessing and manipulating various data structures.

Containers

Containers are objects that hold data. These container classes are defined as class templates that can be customized to hold different kinds of data. The Figure 2.2 illustrates the three main types of container classes. The container classes contain definitions for commonly used data structures such as vector, list, queue, stack, set, map etc. Each container class also defines a set of functions that can be used to manipulate its contents. The STL defines a number of containers which can be grouped into mainly three types: sequence containers, associative containers and derived containers. The derived containers are sometime also referred to as container adapters.

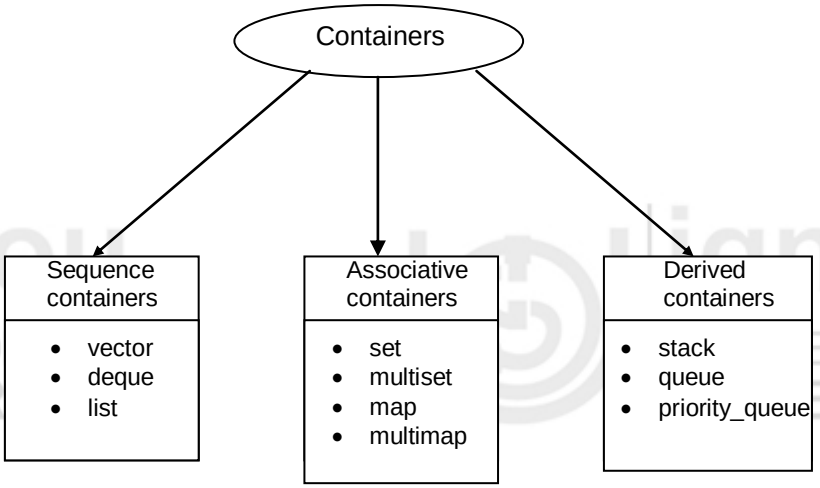

Figure 2.2 : Main Container types

The Table 2.1 lists some commonly used container classes available in the STL.

Table 2.1: List of commonly used Container

Container	Description	Header File	Iterator
vector	A dynamic array. Allows insertion and deletions at rear. Permits direct access.	<vector>	Random access
list	A bidirectional linear list. Allows insertion and deletions anywhere.	<list>	Bidirectional
stack	A standard stack, Last in First out operation.	<stack>	No iterator
queue	A standard Queue, First in First out operation.	<queue>	No iterator
priority queue	A priority queue, highest priority element as first out.	<queue>	No iterator
deque	A double ended queue, allows	<deque>	Random

	insertions and deletions at both ends.		access
set	An associative container for storing unique sets. Allows fast lookup.	<set>	Bidirectional
multiset	An associative container for storing non-unique sets.	<set>	Bidirectional
map	An associative container for storing unique key-value pairs.	<map>	Bidirectional
multimap	An associative container for storing key-value pairs that may use one-to-many mapping.	<map>	Bidirectional

The sequence containers represent linear data structures such as vectors and linked lists. The associative containers are non-linear container that typically store elements in a key-value pair fashion and support fast lookup. The sequence containers and associative containers are collectively referred to as *First Class containers*. Stacks and Queues are actually constrained versions of these first class containers and that is why often referred to as derived containers or container adapters. They enable a program to view a sequential container in a constrained manner. Sometimes we also hear about "near conatiners", which are similar to first class containers but do not support all functionalities of first class containers. Bitsets is one such example.

Hence, out of the various container classes listed in the table, vector, list and deque are sequential containers. Set, multiset, map and multimap are associative containers. Stack, queue and priority queue are derived containers.

Most STL containers provide similar functionality. Many generic operations, such as member function size, therefore apply to all containers. A good number of operations apply on subsets of container classes. Some of the common member functions that apply to most of container classes are listed in the Table 2.2:

Table 2.2: Some common member functions of Container Classes

Member Functions
default constructor, copy constructor, destructor, empty, insert, size, operator=, operator>, operator<, operator<=, operator>=, operator==, operator!=, swap
Functions found only in First class containers
max_size, begin, end, rbegin, rend, erase, clear.

Algorithms

Algorithms are functions that are used across a variety of container classes for processing their contents. As we have just learnt that each container class provides member functions for its basic operations, but STL further extends this by providing some standard algorithms for manipulating different containers. The STL contains approximately seventy standard algorithms to support more extended or complex operations. Standard algorithms have another advantage that they allow working with two different types of containers at the same time, unlike container member functions. The STL implement these algorithms as standalone function templates that can be customized to work with different kind of containers. Inserting, deleting, searching and sorting are some of the examples.

Unlike member functions, the algorithms operate on containers indirectly through the use of iterators. Many algorithms operate on sequences of elements defined by pair of iterators- one pointing to the first element of the sequence and other pointing to one

element past the last element. It is also possible to create new algorithms that operate in a similar fashion to that of STL algorithms. Algorithms often return iterators that indicate the results of algorithms (for example algorithm find). STL algorithms, based on the nature of operations they perform, may be categorized into following groups:

- Retrieve or non-modifying sequence algorithms
- Mutating-sequence algorithms
- Sorting Algorithms
- Set Algorithms
- Relational Algorithms

A list of some of these algorithms along with a description of their purpose is given in the Table 2.3:

Table 2.3: Mutating sequence Algorithm

Mutating-sequence algorithms	
copy()	Copies a sequence
copy_backward()	Copies a sequence from the end
fill()	Fills a sequence with a specified value
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 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

Non-modifying sequence algorithms	
adjacent_find()	Finds adjacent pair of objects that are equal
count()	Counts occurrence of a value in a sequence
count_if()	Counts number of elements that matches a predicate
equal()	True if two ranges are the same
find()	Finds first occurrence of a value in a sequence
find_end()	Finds last occurrence of a value in a sequence
find_first_of()	Finds a value from one sequence in another
find_if()	Finds first match of a predicate in a sequence
for_each()	Apply an operation to each element
mismatch()	Finds first elements for which two sequences differ
search()	Finds a subsequence within a sequence
search_n()	Finds a sequence of a specified number of similar elements

Sorting algorithms	
binary_search()	Conducts a binary search on an ordered sequence
equal_range()	Finds a sub range 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

Set algorithms	
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

Relational algorithms	
equal()	Finds whether two sequences are the same
lexicographical_compare()	Compares alphabetically one sequence with other
max()	Gives minimum 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 sequence

Iterators

Iterators are used to access container class elements. They are called iterators because of their use in traversing the elements (from one to another) of a container class. In this sense they are quite similar to pointers. Iterators hold state information sensitive to the particular containers on which they operate, thus, iterators are implemented appropriately for each type of container. Certain iterator operations are uniform across containers. For example, ++ operation on an iterator moves it to the next element of the container. If iterator i points to a particular element, then, i++ points to the "next" element and *I refers to the element pointed by i.

There are five broad types of iterators supported by the STL. These are listed in Table 2.4:

Table 2.4: Types of Iterators

Iterator	Access method	Movement	I/O Capability
Input	Linear	Forward only	Read only
Output	Linear	Forward only	Write only
Forward	Linear	Forward only	Read/ Write
Bidirectional	Linear	Forward & Backward	Read/ Write
Random	Random	Forward & Backward	Read/ Write

Each type of iterator is used for performing a particular set of functions. The input and output iterators are used to traverse a container and have functionality limited to this use. The forward operator also supports input and output and at the same time retaining its position in the container. The bidirectional iterator provides ability to move backwards in addition to forward movements. The random access iterator allows random jumps to a particular location in addition to bidirectional operations.

Examples of use of List and Map containers

Now that we had a look at the organization and different components of the STL, we will go through some illustrative examples of use of the STL components. Since the STL is quite big and we can not cover examples on the entire STL, we will see here one example each of the use of List and Map container classes.

List is a commonly used container class that implements a standard bidirectional linked list. It supports insertion and deletion operations and can be accessed only in a sequential manner. The STL class list provides appropriate set of member functions to manipulate lists. We will see here an example of use of list container class for creating and processing a list:

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

void display(list<int> &lst)
{
  list<int> :: iterator p;
  for (p=lst.begin(); p!=lst.end(); ++p)
        cout << *p << " " ;
  cout << "\n";
}

int main()
{
 list<int> list1; // empty list of zero length
 list<int> list2(5); //empty list of 5 elements

 for (int i=0; i<3; i++)
    list1.push_back(rand()/100);

 list<int> :: iterator p;
 for (p=list2.begin(); p!=list2.end(); ++p)
   *p=rand()/100;
 cout << "list1 \n";
 display(list1);
```

```
 cout << "list2 \n";
 display(list2);

//Add elements at both the ends of list1
  list1.push_front(100);
  list1.push_back(200);

 //Remove an element at front of list2
  list2.pop_front();

  cout << "now list1 \n";
  display(list1);
  cout << "now list2 \n";
  display(list2);

  list<int> listA, listB;
  listA=list1;
  listB=list2;

  //Merging two lists
  list1.merge(list2);
  cout << "Merged unsorted list \n";
  display(list1);

  //Sorting and Merging
  listA.sort();
  listB.sort();
  listA.merge(listB);
  cout << "Merged sorted list \n";
  display(listA);

  return(0);
};
```

The program above creates various lists and performs different operations on them. It uses member functions like begin(), end(), push_back(), push_front() etc. It also sorts and merges two lists. The user-defined function display() makes use of iterator to display the elements of the lists.

Another example that we would consider is that of container class map. As we discussed earlier a map is a sequence of key:value pairs, where a single value is associated with each unique key. Its an associative container class. The entries in a map are specified as:

phone ["ashok"] = 123456;

Here, phone is a map object and the statement associates number 123456 with the key value "ashok". The map entries can be manipulated using various member functions and algorithms. The key operations in a map include operations like add, delete, modify, sort the entries in map etc. We will have a look at an example of use of map in the program below:

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

```
typedef map<string, int> phonemap;

int main()
{
 string name;
 int number;
 phonemap phone;

// Entering key:value pairs in map
 cout << "Enter three sets of name and numbers \n";
 for (int i=0, i<3; i++)
  {
    cin >> name;
    cin >> number;
    phone[name] = number;
  }

// inserting a new entry
phone["Ramesh"]= 621345;

 //inserting using insert() function
 phone.insert(pair<string, int> ("ajay", 234432));
 int n = phone.size();
 cout << "\n size of map:"  << n;

 // reading the entries in the map using iterator
 cout << "\n List of telephone numbers \n";
 phonemap::iterator p;
 for (p=phone.begin(); p!=phone.end(); p++)
    cout << (*p).first << "  " << (*p).second << "\n";

 cout << "\n";

 //looking up for an entry
 cout << "Enter name:";
 cin >> name;
 number = phone[name];
 cout << "Number:" << number << "\n";

 return(0);
};
```

This program first creates a map (phone) with three entries read from the keyboard. Then it moves to insert two new entries using two ways. It then prints the entire map using iterator. Finally it looks up the value of a given key stored in the map.

Similar to list and map container classes, the other container classes can also be used as the situation desires. We can use vector container class to create and manipulate various arrays; stack for handling LIFO memory operations; queue and priority queues for managing queue operations etc. We can manipulate these data structures not only by the member functions they contain, but also by using various algorithms that apply to them. Iterators help and guide the processing of elements contained in the class. The STL provides a rich set of template declarations along with different algorithms that are very useful in designing and implementing programming solutions for different real world problems. It supports and promotes reuse, a key theme of object oriented programming.

☞ Check Your Progress 2

1) Fill in the blanks:
   a) The two types of first-class STL containers are sequence containers and …….. containers.
   b) The five main iterator types are ……., ……..., ……., ……., and ……. .
   c) The three STL container adapters are ……., ……., and ……. .
   d) STL algorithms operate on container elements indirectly, using …… .
   e) The sort algorithm requires a (n) …… iterator.

2) State whether following are *True* or *False*.
   a) The STL makes abundant use of inheritance and virtual functions.
   b) An iterator acts like a pointer to an element.
   c) STL algorithms can operate on C-like pointer-based arrays.
   d) STL algorithms are encapsulated as member functions within each container class.
   e) Container member function end yields the position of the container's last element.

3) For each of the following, write a single statement that performs the indicated task:
   a) Name the member functions that are used to refer to beginning and end of the list class.

      ……

      ……

      ……

   b) Name the different type names used to categorize the algorithms.

      ……

      ……

      ……

   c) What is the purpose of push_back(), push_front(), pop_back() and pop_front() functions of a list.

      ……

      ……

      ……

2.6 SUMMARY

This unit has introduced the basic idea of template classes and functions. Templates are mechanisms supported by C++ for generic programming. Templates allow us to generate a family of classes or a family of functions to handle different data types. Template classes and functions promote reuse and avoid code duplication. The member functions of a class template are also defined using the parameters of the class templates.

C++ now contains a rich set of template classes and functions packaged as a library, known as the standard template library. The STL consists of three main components: containers, algorithms, and iterators. Containers are objects that hold data of some type and are usually grouped into three types: sequential, associative and derived. Container classes contain a large number of member functions that make manipulating them simple. In addition to member functions, we also have a large number of algorithms (such as sorting, searching, copying, and merging) that are used to manipulate the container classes and perform various operations on them. Iterators, which are similar to pointers allow manipulation of elements of container classes indirectly by algorithms.

2.7 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

1. (a) function-template specialization, class-template specialization
   (b) template<…….>
   (c) parameterized
   (d) binary scope resolution

2. (a) True
   (b) False, it need not be unique.
   (c) True
   (d) False, This also allows for a type parameter of a fundamental type

3.
```
template<typename T>
        void printarray(T a[], int n)
        {
         for (int i=0; i<n-1; i++)
            cout << a[i] << " ";
         }
```

4.
```
template<class T1, class T2>
        class item
        {
          T1 x;
          T2 y;
        Public:
         item(T1 a, T2 b)
         {
           x = a;
           y = b;
         }
        template<typename T>
        void display(T a, T b)
        {
           cout << "item name" << a;
          cout << "item measure" << b ;
        }
```

Check Your Progress 2

1. (a) Associative
   (b) input, output, forward, bidirectional and random access
   (c) stack, queue, priority queue
   (d) iterators
   (e) random access

2. (a) False, These are avoided for performance reasons.
   (b) True
   (c) True
   (d) False, STL algorithms are not member functions. They operate indirectly on container classes through iterators.
   (e) False, it actually yields the position just after the end of the container.

3. (a) begin() and end()
   (b) non-modifying, mutating, sort, set and relational
   (c) push_bac() – is used to insert an element at the back of a list
   push_front() – is used to insert an element at the front of the list
   pop_back() – deleting an element from the back of the list
   pop_front() – deleting an element from the front of the list

2.8 FURTHER READINGS

1) E. Balaguruswamy, *Object Oriented Programming with C++*, Tata McGraw Hill, 2010.
2) P. Deitel and H. Deitel, C++: *How to Program, PHI*, 7th ed, 2010.
3) B. Strousstrup, *Programming – Principles and Practices using C++*, Addison Wesley, 2009.
4) B. Stroustrup, *"Parameterized types for C++" Proceedings of the USENIX C++* Conference, Denver, Colorado, October 1988.

UNIT 3 EXCEPTION HANDLING

Structure **Page Nos.**

3.0	Introduction	46
3.1	Objectives	46
3.2	Exceptions in C++ Programs	46
3.3	Try, Throw and Catch Expressions	48
3.4	Specifying Exception Types	53
3.5	Summary	58
3.6	Answers to Check Your Progress	58
3.7	Further Readings	58

3.0 INTRODUCTION

In the units covered so far, we have talked about various features provided by C++ to design different programs. The programs which are written correctly produce the desired output when input data is supplied to them. However, in some cases programs may behave in an unexpected manner. One of the reasons that may lead to this situation is when we provide inappropriate input data (something that the programmer never expected or anticipated). These situations are unexpected and that is why they are termed exceptions. Exceptions are different from syntactical and logical errors but they also cause the program to misbehave. Exceptions are usually encountered at run time. In order to ensure that programs work correctly under all conditions, we have to incorporate mechanisms to identify and handle different exceptions that may occur in a program.

This unit introduces the nature and type of exceptions that may occur in a C++ program. It then describes the steps in exception handling. Then the syntax and use of try, throw and catch expressions are explained with appropriate examples. This unit tries to present a comprehensive picture of the exception handling mechanisms in C++ and their use in designing correct and robust programs.

3.1 OBJECTIVES

At the end of the unit, you should be able to:

- know what is exceptions and how to handle them;
- use try, catch and throw expressions to identify and handle exceptions;
- describe the standard exception hierarchy;
- appreciate the usefulness of exception handling mechanisms in C++; and
- write robust and fault tolerant C++ programs.

3.2 EXCEPTIONS IN C++ PROGRAMS

The name ‘exception’ implies that the problem is one which occurs infrequently- if the “rule” is that a statement normally executes correctly, then the “exception to the rule” is that a problem occurs.

The term exception itself implies an unusual condition. Exceptions are anomalies that may occur during execution of a program. Exceptions are not errors (syntactical or logical) but they still cause the program to misbehave. They are unusual conditions which are generally not expected by the programmer to occur. An exception in this sense is an indication of a problem that occurs during a programs’s execution. The

typical exceptions may include conditions like divide by zero, access to an array outside its range, running out of memory etc.

For example: Consider the following code segment:

```
# include <iostream>
                    int main()
                    {
                      int x,y;
                      cout << "enter values of x & y \n";
                      cin >> x;
                      cin >> y;
                      cout << "result of x divided by y is:" <<
x/y;
                     }
```

This program reads values of variables x and y from standard input and produces output of x divided by y, which is then displayed on the standard output. However, please note that if the value of y read from the keyboard is '0', then the program witnesses a divide by zero situation. In this case, the result will be infinite and program terminates.

To deal with these unexpected conditions, C++ provides an exception handling mechanism that detects and handles exceptions in a predefined way. Exception handling mechanism was not part of the original C++ specifications. It was added later and now almost all compilers support this feature. C++ exception handling mechanism provide with a scheme to identify and handle predictable exceptions that may occur during program execution. It may kindly be noted that exception handling mechanism needs to be incorporated explicitly in the program to handle the exceptions and that it may be able to handle only those exceptions that are provisioned in exception handling statements.

Exceptions can be of two kinds: *asynchronous and synchronous*. The exceptions that are caused by events beyond the control of the program (such as keyboard interrupts) are called asynchronous exceptions. The other unusual conditions (such as overflow, out of range array index) that are part of the program are called synchronous exceptions. The C++ exception handling mechanism is designed to handle synchronous type of exceptions only. It provides a means to detect, report and act on an unusual condition. The exception handling mechanism C++ deals with exceptions by performing following tasks:

a. Identify the problem (**hit** the exception)
b. Inform that an exception has occurred (**throw** the exception)
c. Exception handler catches the exception information (**catch** the exception)
d. Exception handler takes corrective actions (**handle** the exception)

These tasks are incorporated in C++ exception handling mechanism in two segments, one to detect (**try**) and inform (**throw)** about the exception, and the other to catch the exception and take appropriate actions to handle it.

As indicated earlier, the C++ exception handling mechanism is built upon following three expressions:

- Try
- Throw
- Catch

The expression **try** is used to preface (enclose in braces) that block which may generate exceptions. This block is often termed as try block. The **throw** expression is invoked when an exception is detected. It informs the catch block that an exception has occurred. The exception handling code is enclosed in **catch** block. The catch block catches the exception thrown by the throw expression and handles it in a predefined manner. The Figure 3.1 further shows the use of these three expressions and their relationship:

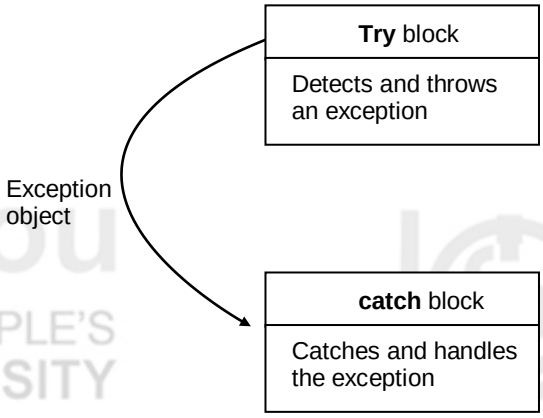

Figure 3.1 : Exception Handling

3.3 TRY, THROW AND CATCH EXPRESSIONS

The keyword 'try' is used to enclose the statements that may generate an exception. These statements begin with a try keyword and are enclosed in braces. When the try block encounters an exception, it uses the throw keyword to pass the information to the exception handling block. The exception handling block is enclosed within a block preceding the catch keyword. This catch block should immediately follow the try block that throws the exception. The general syntax of these statements and blocks is as follows:

```
……..
……..
try
{
  ……
  ……
  throw exception;
  ……
  ……
 }
catch (type arg)
{
  ……..
  ……..
  ……..
 }
……
……
```

As some statement within the try block of the program generates an exception, it is thrown using the throw statement. At this time, the program control transfers to the

catch block. The throw contains an argument named exception which is passed to the catch block as an argument. However, the catch block handles this exception only if the arguments passed from throw matches with the argument specified in the catch block. In case the exceptions that may be generated could be of different types, then multiple catch blocks will be required. This can be done by writing these catch blocks one after another. When an exception is thrown, the exception object is compared with the argument in the catch blocks written in succession. The first catch block matching the exception object is executed. We will see use of multiple catch blocks in the next part of the chapter. If the exception object thrown does not match with the argument specified in catch block, then the program gets terminated by automatic invocation of abort() function. Sometimes exceptions are thrown from functions that are invoked within the try block. The point at which this throw is executed is called throw point. After an exception is thrown to the catch block, control cannot return back to the throw point. The Figure 3.2 demonstrates this situation when a function invoked from within the try block throws an exception.

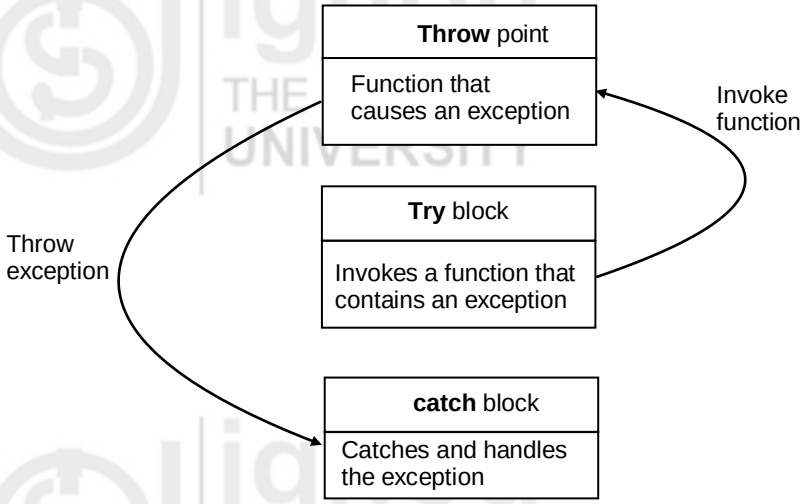

Figure 3.2 Throw, try and catch blocks for exception handling

We present following two code segments that demonstrate use of try, throw and catch expressions for exception handling. The first program presents a simple example of use of a single try and catch block. The second program presents an example where the exception is thrown from a function invoked from within the try block.

```
# include <iostream>
int main()
{
        int x,y;
        cout << "enter values of x & y \n";
        cin >> x;
        cin >> y;
        int a = x-y;
        try
        {
         if (a!=0)
         {
          cout << "Result (x/a) =" << x/a;
         }
         else
         {
          throw (a);
         }
        }
```

```
        catch (int i)
        {
          cout << "Exception caught a=" << a;
        }
        return(0);
        }
```

```
# include <iostream>
void divide (int x, int y, int z)
{
        cout << "inside function \n";
        if (x-y)!=0)
        {
          int r = z/ (x-y);
          cout << "result =" << r << "\n";
        }
        else
        {
          throw (x-y);
        }
 }

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

You would see that in the first program, if input values are 20 and 15 then the program runs correctly without any exception with result being 4. On the other hand, if input given is 10 and 10, the program would detect an exception and display it as "0" value. Similarly, in the second program for the first invocation of divide (), the result will be -3, whereas the second invocation will result into an exception.

The throw statement can take multiple forms:

```
throw (exception);
throw exception;
throw;
```

The first two throw statements pass the exception object to the catch handler, whereas the third throw statement without any exception object is used to rethrow an exception from within a catch block.

A catch handler may rethrow the exception, without handling it, to the next catch block. This is equivalent to passing the exception to the next enclosing catch block within the scope of try/catch sequence. Please note that a rethrown exception is not

caught by the same catch handler or any other catch handler in that group, but by an appropriate catch handler in the outer try/catch sequence only. This also applies to those exceptions that may be detected within a catch handler block itself.

Multiple Catch Statements

We have seen earlier that a catch block looks like a function definition. It has a type argument which specifies the type of exception that this catch block may handle followed by statements (enclosed within braces) to process the exception. After executing these statements, the control goes to the statement immediately following the catch block. It is also possible that a program may have more than one exception condition. In such cases multiple catch statements are needed to handle the different types of exceptions. Now when an exception is thrown, the exception handlers are searched in order for an appropriate match. The first handler that matches the thrown exception object type is executed. Rest of the catch blocks are then skipped and the control goes to first statement following the last catch block. This indicates that in case of multiple catch blocks matching the thrown exception, only the first matching catch block is executed. The following program demonstrates the use of multiple catch blocks:

```
#include<iostream>
void test(int x)
{
        try
        {
          if (x==1) throw x;
          else
            if (x==0) throw 'x';
          else
            if(x==-1) throw 1.0;
        }
        catch (char c)
        {
          cout << "caught a character \n";
        }
        catch(int m)
        {
          cout << "caught an integer \n";
        }

        catch(double d)
        {
          cout << "caught a double \n";
        }
}

int main()
{
        cout << "testing multiple catches \n";
        cout << "x==1 \n";
        test(1);
        cout << "x==0 \n";
        test(0);
        cout << "x==-1 \n";
        test(-1);
        cout << "x==2 \n";
        test(2);
```

```
        return(0);
}
```

As you can see this program has multiple catch blocks, each handling exception objects of different types. One integer, other character and the third one a double exception argument. The last invocation of test with argument 2 does not throw any exception and hence no catch block is invoked.

Catch all exceptions

There are many situations where it may be very difficult to anticipate in advance all possible types of exceptions that may occur in a program. C++ provides with a mechanism to cope with this problem in form of a catch (…) expression. This type of catch statement catches all exceptions, unlike only one exception being caught. The general syntax of use of this kind of catch expression is as follows:

```
catch(…)
{
//statements for processing
//all exceptions
}
```

The following example presents a case of use of this kind of catch expression:

```
# include <iostream>
void test (int x)
{
        try
        {
          if (x==0) throw x;
          if (x==-1) throw 'x';
          if (x==1) throw 1.0;
        }
        catch (…)
        {
         cout << "caught an exception \n";
        }
}
int main()
{
        cout << "testing generic catch \n";
        test (-1);
        test (0);
        test(1);
        return(0);
}
```

You may see that this program prints the line "caught an exception" three times as follows:

Output:

caught an exception
caught an exception
caught an exception

This is because all the exceptions (x getting values -1, 0 and 1) are caught by the same catch handler block. This is the reason why this kind of catch block is termed as *catch all expression.* This property can make this kind of catch all expression to be placed as default catch handler. This default handler may then be used to catch all those exceptions which are not handled explicitly. However, one must be careful to place this catch all expression in the last place of catch handlers.

☞ Check Your Progress 1

1) List five common examples of exceptions.

……

……

……

2) If no exceptions are thrown in a try block, where does control proceed to after the try block completes the execution?

……

……

……

3) What happens if an exception is thrown outside a try block?

……

……

……

4) What does the statement throw do?

……

……

……

5) What happens if several handlers match the type of thrown object?

……

……

……

3.4 SPECIFYING EXCEPTION TYPES

We have seen earlier that the exception type is reported by the throw statement to the catch handler, which then takes appropriate action on it. It is also possible to restrict a function to throw only certain specified exceptions. This can be done by adding a throw list clause to the function definition. The general syntax of doing this exception type specification is as follows:

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

The type-list after throw specifies the type of exceptions that may be thrown. An attempt to throw another type of exception will cause abnormal program termination. In case we want to prevent a function from throwing any exception at all, we may do so by making the type-list empty, i.e., simply writing throw ( ) in the function header line. However, these specifications operate only when the function is called back from a try block and it reports back the exceptions to that try block. It does not apply if exception is thrown within the function code itself. An example to demonstrate this scenario is as follows:

```
#include <iostream>
void test(int x) throw (int, double)
{
        if (x==0) throw 'x';
        else
          if (x==1) throw x;
          else
            if (x==-1) throw 1.0;
}
int main()
{
        try
        {
          cout << :testing throw specifications \n";
          cout << "x==0 \n";
          test(0);
          cout << "x==1 \n";
          test(1);
          cout << "x==-1 \n";
          test(-1);
          cout << "x==2 \n";
          test(2);
        }
        catch(cahr c)
        {
           cout << "caught a character \n";
        }
        catch (int m)
        {
          cout << "caught an integer \n";
        }
        catch (double d)
        {
          cout << caught a double \n";
        }
        return (0);
}
```

You may note that this program tries to throw a char exception object in the very first invocation of test(). This results in an abnormal termination of the program, since the test can throw only exceptions of type int and double.

Program Output:

```
testing throw specifications
x==0
caught a character
```

Throwing an exception that has not been declared in a function's exception specification causes a call to function unexpected. The compiler will not generate a compilation error if a function contains a throw expression for an exception not listed in the function's specification. An error occurs only when that function attempts to throw that exception at execution time. To avoid surprises at execution time, its better to carefully check the code to ensure that functions do not throw exceptions not listed in their specifications.

The *function unexpected* calls, the function registered with the function set_unexpected (defined in header file <exception>). If no function has been registered in this manner, function terminate is called by default. The function set_terminate can specify the function to invoke when terminate is called. Otherwise, terminate calls abort, which terminates the program without calling the destructors of any remaining objects of automatic or static storage class. This could lead to resource leaks when a program terminates prematurely.

Exceptions and Stack Unwinding

When an exception is thrown but not caught in a particular scope, the function call stack is "unwound" and an attempt is made to catch the exception in the next outer try….catch block. Unwinding the function call stack means that the function in which the exception was not caught terminates, all local variables in that function are destroyed and control returns to the statement that originally invoked that function. If a try block encloses that statement, an attempt is made to catch the exception. If a try block does not enclose the statement, stack unwinding occurs again. If no catch handler ever catches this exception, function terminate is called to terminate the program. The following programming example demonstrates stack unwinding:

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

//function3 throws runtime error
void function3() throw (runtime_error)
{
  cout << "in function3" << endl;
// no try block, stack unwinding occurs, return control to function2
  throw runtime_error ("runtime_error in function3");
}

//function2 invokes function3
void function2() throw (runtime_error)
{
  cout << "function3 is called inside function2" << endl;
  function3(); // stack unwinding occurs, return control to function1
}

//function1 invokes function2
void function1() throw (runtime_error)
{
  cout << "function2 is called inside function1" << endl;
  function2(); //stack unwinding occurs, return control to main
}

int main()
{
  //invoke function1
```

```
  try
   {
     cout << "function1 is called inside main" << endl;
     function1();
    }
  catch (runtime_error & error)
   {
     cout << "exception occurred:"  << error.what() << endl;
     cout << " exception handled in main" << endl;
   }
}
```

The program will produce the following output:

function1 is called inside main
function2 is called inside function1
function3 is called inside function2
in function3
exception occurred: runtime_error in function3
exception handled in main

Exception handling and Constructors and Destructors

It is worth discussing that what happens if an exception is thrown while executing a constructor? Since the object is not yet fully constructed, its destructor would not be called once the program control goes out of the object's context. And, if the constructor had reserved some memory before the exception was raised, then there would be no mechanism to prevent such memory leak. Hence. Appropriate exception handling mechanism must be implemented pertaining to the constructor routine to handle exceptions that occur during object construction.
Where to catch the exception is another important issue here. Whether it should be done inside the constructor block or inside the main. If we allow the exception to be handled inside main then we would not be able to prevent the memory leak situation. Therefore, we must catch the exception within the constructor block so that we get chance to free up any reserved memory spaces. However, we must simultaneously rethhrow the exception to be appropriately handled inside the main block.

Exceptions and Inheritance

Like the normal inheritance concept, various exception classes can be derived from a common base class. If a ctach handler catches a pointer or reference to an exception objectof a base-class type, it also can catch a pointer or reference to all objects of class publicly derived from that base class- this allows for polymorphic processing of related errors. Using inheritance with exceptions enables an exception handler to catch related errors with concise notation. One approach is to catch each type of pointer or reference to a derived-class exception object individually, but a more concise approach is to catch pointers or references to base-class exception objects nstead. Also, catching pointers or references to derived-class exception object individually is error prone, especially if you forget to test explicitly for one or more of the derived-class pointer or reference types.

Standard Library Exception Hierarchy

As we know that exceptions fall nicely into a number of categories. The C++ standard library includes a hierarchy of exception classes. The hierarchy is headed by base-class exception (defined in header file <exception>), which contains virtual function what, which derived base classes can override to issue appropriate error messages. Immediate derived classes of base class exception include runtime_error and logic_error (both defined in header <stdexcept>), each of which has several derived classes. Also derived from exception are the exceptions thrown by C++ operators – for example, bad_alloc is thrown by new, bad_cast is thrown by dynamic_cast and

bad_typeid is thrown by typeid. Including bad_exception in the throw list of a function means that, if an unexpected exception occurs, function unexpected can throw bad_exception rather than terminating the program's execution or calling another function specified by set_unexpected. The class logic_error is the base class of several standard exception classes that indicate errors in program logic, whereas class runtime_error is the base class of several other standard exception classes that indicate execution-time errors. The Figure 3.3 below presents an overview of standard library exception classes.

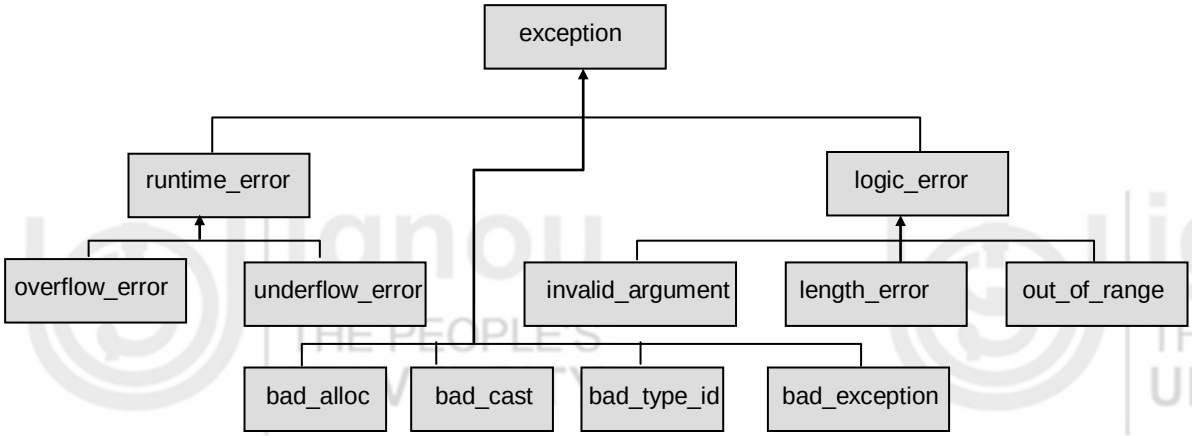

Figure 3.3 Some of the Standard Library Exception Classes

☞ Check Your Progress 2

1) What happens if no catch handler matches the type of a thrown object?

……

……

……

2) Must throwing an exception cause program termination?

……

……

……

3) What happens when a catch handler throws an exception?

……

……

……

4) How do you restrict the exception type that a function can throw?

……

……

……

5) What happens if a function throws an exception of a type not allowed by the exception specification for the function?

……

……

……

3.5 SUMMARY

Exceptions are a kind of problem that may occur when a program is executed. C++ provides an exception handling mechanism to handle synchronous exceptions. An exception is handled by a group of try, throw and catch expressions. The block that may cause a possible exception is enclosed within a try block. The exceptional situation occurring in the try block is reported by throw expression to an exception handling block, called catch block. When an exception is not caught the program is terminated. The throw expression passes the exception object to an appropriate catch block. A program may have more than one catch block to handle different kinds of exceptions. We can also have a catch all expression that can be used as a default handler. A catch block may also rethrow an exception without processing it. We may also restrict the type of exception objects that a function may throw. The try-throw-catch mechanism in C++ is quite useful to provide for dealing with unexpected situations that may occur at runtime in a program.

3.6 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

1) Insufficient memory to satisfy a new request, array subscript out of bounds, arithmetic overflow, division by zero, invalid function parameters.
2) The exception handlers (in the catch handlers) for that try block are skipped, and the program resumes execution after the last catch handler.
3) An exception thrown outside a try block causes a call to terminate.
4) It passes the exception object to the catch handler. If it occurs within a catch block, it rethrows the exception.
5) The first matching exception handler after the try block is executed.

Check Your Progress 2

1) This causes the search for a match to continue in the next enclosing try block if there is one. As this process continues, it might eventually be determined that there is no handler in the program that matches the type of thrown object. In this case, program is aborted.
2) No, but it does terminate the block in which the exception is thrown.
3) The exception will be processed by a catch handler (if one exists) associated with the try block (if one exists) enclosing the catch handler that caused the exception.
4) Provide an exception specification listing the exception type that the function can throw.
5) Function unexpected is called to terminate the program.

3.7 FURTHER READINGS

1) E. Balaguruswamy, *Object Oriented Programming with C++*, Tata McGraw Hill, 2010.
2) P. Deitel and H. Deitel, C++: *How to Program, PHI*, 7th edition, 2010.
3) B. Strousstrup, *Programming – Principles and Practices using C++*, Addison Wesley, 2009.

UNIT 4 A CASE STUDY

Structure **Page Nos.**

4.0 Introduction 59
4.1 Objectives 59
4.2 Designing a Transaction-processing System 60
4.3 Summary 74
4.4 Further Readings 74

4.0 INTRODUCTION

In the units covered so far, we have discussed various features of C++ language that help in designing and implementing useful programs to solve various real world problems. The bottom up approach adopted by C++ language provides a better way to capture the details of real world problems and design efficient and adaptable solutions. Whether it is the mechanism of constructors and destructors, inheritance, or polymorphism; all taken collectively provide suitable design methodology and tool for solving various real world problems through C++ programming formulations. In order to solve a real world problem, the first and foremost requirement is to identify the various objects in the system along with their general attributes. This is then followed by the more involved process of identifying the methods/ functions that can be applied on the different objects. Once this is done, the effort gets more focused towards design, which involves designing various classes and implementing different functions.

This unit presents a case study of designing and implementing a transaction-processing system for banking domain. Unlike a database design for bank accounts, we have adopted a file processing approach to emphasize the C++ features and capabilities. The design involves creating necessary data files to store accounts and customer information and then accessing them through suitable code for writing data, appending data, performing credit operations and displaying the results. Our focus in the case study is on demonstrating the design methodology and the steps required to design a small real world application. The steps involved in design and implementation of the transaction-processing system are described with relevant explanations at various places. The program design also makes use of certain features of C++ which are used for larger programs. After a careful study of the unit, you would be able to clearly identify the broad guidelines and general steps involved in solving a real world problem and using C++ for solving variety of problems.

4.1 OBJECTIVES

At the end of the unit, you should be able to:

- explain the steps involved in solving real world problems;
- describe the overall framework of such designs;
- appreciate the usefulness of various features of C++ for solving different real problems;
- design C++ programming solutions for many other problems; and
- design C++ programs involving multiple files.

4.2 DESIGNING A TRANSACTION-PROCESSING SYSTEM

A transaction processing system is one where certain activities are carried out as part of some bigger goal. The kind of transactions performed in a system depends on the domain of the problem. For example, in a ticket reservation domain the key transactions may be booking a ticket, realizing payment for a ticket, cancelling a ticket, modifying or amending a ticket, refund of a cancelled ticket etc. Similarly in a banking domain, the transactions could be opening a customer account, updating the account records, processing withdrawal from an account, deposit into an account, printing summary of accounts etc. Every transaction processing system, irrespective of the domain, has to complete certain activities (referred to as transactions). Nowadays when a large number of transaction processing activities are automated to be performed on a computer system, it is very important to know and understand how can we design such a system. Moreover, with the increased use of Internet and web-based services many of these transactions are initiated and realized at different physical machines. The data related to transactions is sent over communication lines. One common thing however in all kind of transaction processing systems is the need to store and manipulate associated data. The data of the system may be stored either in a database format or in terms of a collection of various files. The general practice in most of the real world transaction processing systems is to go for a database design approach. However, in the example described below we have used a file processing approach.

Most of the transaction processing systems are quite big and result into large programs. The large programs often comprise of various modules. In order to have better understanding and design convenience, these transaction processing systems are designed as a collection of multiple programs. In the previous units we have largely seen example programs consisting of a single program file. Whenever we have to design programming solution for a larger problem, it is often required to organize the large code into multiple files. Using a multiple file organization not only helps in clarity of the design but also in appropriate use of class libraries and better coordination of programmers working on the large project. In fact, large programs are usually divided into separate files, where different files have code for different functionalities, such as one file for mathematical analysis, another for graphics display and a separate one for I/O etc. Large applications sometimes also involve multiple designers who coordinate their effort to design the final solution. Most of the C++ IDEs provide a feature called PROJECT which helps in designing and organizing larger programs comprising multiple files.

We will now see how we can use the various features and capabilities of C++ that we have learned so far to design a transaction-processing system. The proposed system involves some fixed-length account records for a company having certain number of customers. Each record consists of an account number that acts as the record key, a last name, a first name and a balance. The transaction-processing program is to be designed in such a manner that it can provide overall management functions for the accounts and transactions. It should be able to perform functions like update an account, insert a new account, delete an account and insert all the account records into a formatted text file for printing.

The key components in this application design can be understood through following abstract diagram representation.

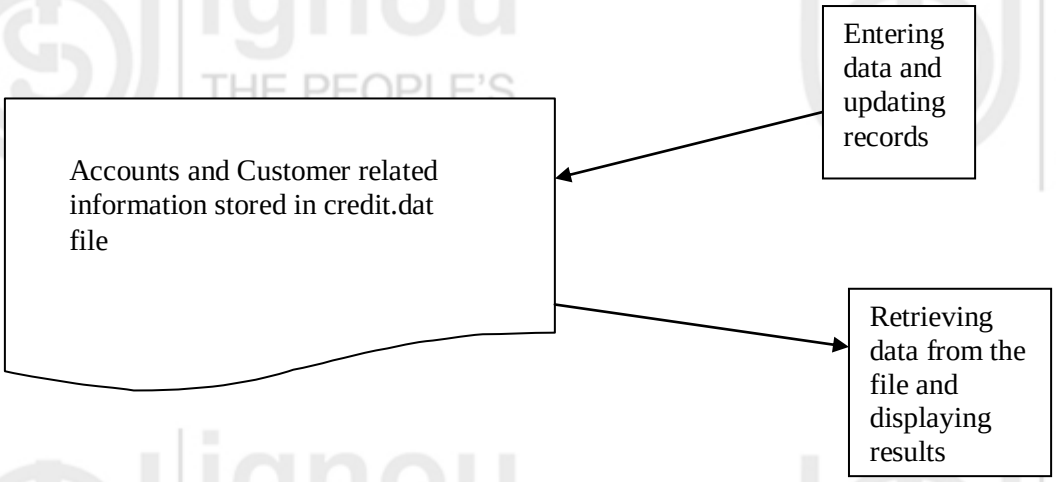

Figure 4.1 : Abstract Diagram of transaction processing system

As we can see, the entire data corresponding to accounts and customer information is stored in credit.dat file. We will design program to enter, add and update the data in this file. The data entered may also be retrieved and displayed as a formatted text output through the use of various stream manipulators. We first create a ClientData class header file that defines the format of the data and then define the constructor and certain basic methods. The following two program segments (Program 4.1 and 4.2) are written for this purpose.

```
 1  #ifndef CLIENTDATA_H
 2  #define CLIENTDATA_H
 3  #include <string>
 4  using namespace std;
 5  class ClientData
 6  {
 7  public:
 8  ClientData(int =0, string = " ", string = " ", double =
0.0);
 9
10  void setAccountNumber (int);
11  int getAccountNumber() const;
12  void setLastname(string);
13  string getLastName() const;
14  void setFirstName(string);
15  string getFirstName() const;
16  void setBalance(double);
17  double getBalance() const;
18
19  private:
20  int accountNumber;
21  char lastName[15];
22  char firstName[10];
23  double balance;
24  };
25  #endif
```

Program 4.1: ClientData class header file

The program 4.1 above defines client data header file which specifies the data format to be used in the application. The main data items are account number (an integer

value), last name of customer (a string), first name of customer (a string) and balance (a float value denoting the account balance). These data items are declared as private. The methods to access and modify this data are setAccountNumber(), getAccountNumber(), setLastName(), getLastName(), setFirstName(), getFirstName(), setBalance() and getBalance(). All these functions are used to define and retrieve values for various fields, and are defined in next program (Program 4.2). The client data represent a customer's credit information and the methods described in program 4.2 provide the code for manipulating the data. The exact code is described in in the program 4.2 given below:

```
 1  #include <string>
 2  #include "ClientData.h"
 3  using namespace std;
 4
 5  // default ClientData constructor
 6  ClientData::ClientData(int accountNumberValue, string
lastNameValue,
 7     string firstNameValue, double balanceValue)
 8 {
 9  setAccountNumber(accountNumberValue);
10  setLastName(lastNameValue);
11  setFirstName(firstNameValue);
12  setBalance(balanceValue);
13 } // end ClientData constructor
14  //get account number value
15  int ClientData::getAccountNumber() const
16 {
17  return accountNumber;
18 }
19  //set account number value
20  void ClientData::setAccountNumber(int accountNumberValue)
21 {
22  accountNumber=accountNumberValue;
23 }
24 // get last name value
25  string ClientData::getLastName() const
26  {
27   return lastName;
28  }
29  // set last name value
30  void ClientData::setLastName(string lastnameString)
31  {
32   int length=lastNameString.size();
33   length = (length<15? length:14); \\for copying at most 15
chars
34   lastNameString.copy(lastName, length);
35   lastName[length]='\0'; \\appending null character
36  }
37
38   //get first-name value
39   string ClientData::getfirstName() const
40   {
41    return firstName;
42   }
43
44   // set first-name value
45   void ClientData::setfirstName(string firstNameString)
46   {
47    // copy at most 10 chars
48    int length =firstNameString.size();
49    length = (length < 10? length:9);
```

```
50    firstNameString.copy(firstName, length);
51    firstName[length]='\0'; \\ appending null char
52   }
53
54    // get balance value
55    double ClientData::getBalance() const
56    {
57     return balance;
58    }
59
60    //set balance value
61    void ClientData::setBalance(double lalanceValue)
62    {
63     balance=balanceValue;
64    }
65
```

Program 4.2: ClientData class representing customer credit information

As you may easily notice, the program defines the ClientData constructor comprising of various functions, each of which have a specified code. The functions setLastName() and setFirstName() limit the number of characters read from input that are finally written to actual data.

We now look at the code (Program 4.3) for creating a file credit.dat with some data entries to be used in our transaction processing system. The program creates a binary file credit.dat for output. It then writes 100 blank records into the credit.dat data file. The next program (program 4.4) then uses various functions to actually write data into this file.

```
 1  // creating randomly accessible file credit.dat
 2  #include <iostream>
 3  #include <fstream>
 4  #include <cstdlib>
 5  #include "ClientData.h"
 6  using namespace std;
 7
 8  int main()
 9 {
10  ofstream outCredit ("credit.dat", ios::out | ios::binary);
11  if (!outCredit)
12  {
13    cerr << "File could not be opened." << endl;
14    exit(1);
15  }
16
17  ClientData blankClient; // constructor zeros out each data
member
18  // output 100 blank records to file
19  for (int i=0; i<100, i++)
20    outCredit.write(reinterpret_cast < const char * >
(&blankClient),
21     sizeof(ClientData));
22  }
```

Program 4.3: Creating the credit.dat file with 100 blank records

Once the file credit.dat is created we can use this file to store the desired data corresponding to the accounts and customers. After entering the basic data, actual

transaction-processing may be performed. The program below reads the data from user entered values through keyboard and then uses fstream functions to store data at desired locations in the file credit.dat. Note that the file is opened in out mode for writing. An example run of the program 4.4 is presented after the program code. The run shows how different data values can be entered into the data file. Note that line 19 includes the header file ClientData.h defined in Program 4.1 so the program can use ClientData objects.

```
 1
 2  // Writing to a random-access file.
 3  #include <iostream>
 4  using std::cerr;
 5  using std::cin;
 6  using std::cout;
 7  using std::endl;
 8  using std::ios;
 9
10  #include <iomanip>
11  using std::setw;
12
13  #include <fstream>
14  using std::fstream;
15
16  #include <cstdlib>
17  using std::exit; // exit function prototype
18
19  #include "ClientData.h" // ClientData class definition
20
21  int main()
22  {
23     int accountNumber;
24     char lastName[ 15 ];
25     char firstName[ 10 ];
26     double balance;
27
28     fstream outCredit( "credit.dat", ios::in | ios::out |
ios::binary );
29
30     // exit program if fstream cannot open file
31     if ( !outCredit )
32     {
33        cerr << "File could not be opened." << endl;
34        exit( 1 );
35     } // end if
36
37     cout << "Enter account number (1 to 100, 0 to end
input)\n? ";
38
39     // require user to specify account number
40     ClientData client;
41     cin >> accountNumber;
42
43     // user enters information, which is copied into file
44     while ( accountNumber > 0 && accountNumber <= 100 )
45     {
46        // user enters last name, first name and balance
47        cout << "Enter lastname, firstname, balance\n? ";
48        cin >> setw( 15 ) >> lastName;
49        cin >> setw( 10 ) >> firstName;
50        cin >> balance;
51
```

```
52         // set record accountNumber, lastName, firstName and
balance values
53         client.setAccountNumber( accountNumber );
54         client.setLastName( lastName );
55         client.setFirstName( firstName );
56         client.setBalance( balance );
57
58         // seek position in file of user-specified record
59         outCredit.seekp( ( client.getAccountNumber() - 1 ) *
60            sizeof( ClientData ) );
61
62         // write user-specified information in file
63         outCredit.write( reinterpret_cast< const char * >(
&client ),
64            sizeof( ClientData ) );
65
66         // enable user to enter another account
67         cout << "Enter account number\n? ";
68         cin >> accountNumber;
69      } // end while
70
71      return 0;
72   } // end main
```

Program 4.4: Writing data to credit.dat file

```
Enter account number (1 to 100, 0 to end input)
? 37
Enter lastname, firstname, balance
? Singh Shweta 0.00
Enter account number
? 29
Enter lastname, firstname, balance
? Tiwari Nisha -24.54
Enter account number
? 96
Enter lastname, firstname, balance
? Jolly Stellina 34.98
Enter account number
? 88
Enter lastname, firstname, balance
? Sen Ajay 258.34
Enter account number
? 33
Enter lastname, firstname, balance
? Ghosh Soumitra 314.33
Enter account number
? 0
```

The transaction processing system that we are designing can now be visualized in figure 4.2:

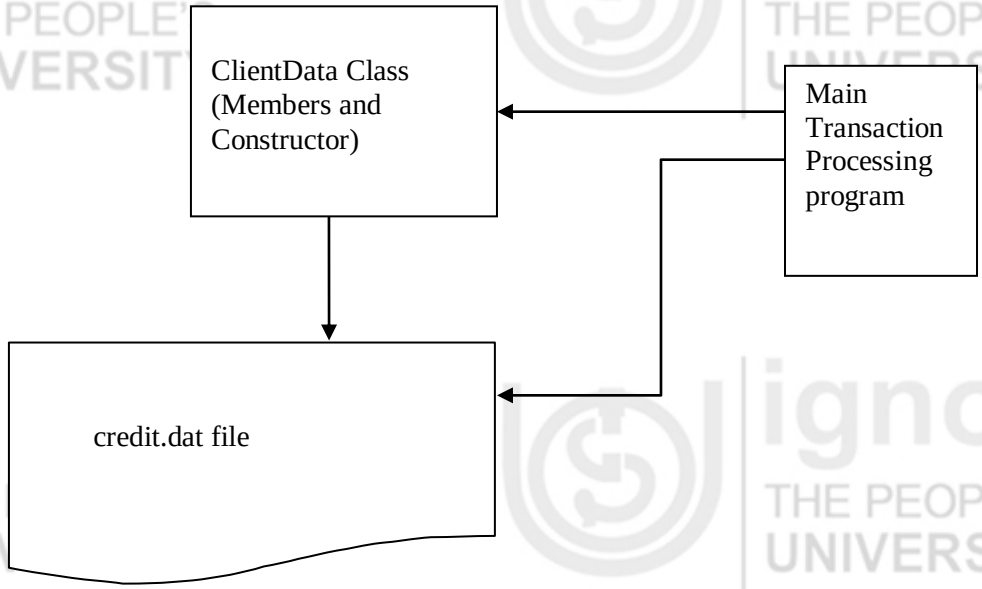

Figure 4.2: Transaction Processing System Program Structure

We now present our main transaction-processing program (Program 4.5) which uses the ClientData.h and credit.dat files to achieve "instant" -access processing. As we discussed earlier, the program manages a bank's account information. The program can perform all functions of accounts processing. It can update existing accounts, adds new accounts, deletes accounts and stores a formatted listing of all current accounts in a text file. We assume that the program 4.3 has been executed to create the file credit.dat and that the program of Program 4.4 has been executed to insert the initial data, before this program can be used for transaction-processing operations.

```
 1
 2   // This program reads a random-access file sequentially,
updates
 3   // data previously written to the file, creates data to be
placed
 4   // in the file, and deletes data previously stored in the
file.
 5   #include <iostream>
 6   using std::cerr;
 7   using std::cin;
 8   using std::cout;
 9   using std::endl;
10   using std::fixed;
11   using std::ios;
12   using std::left;
13   using std::right;
14   using std::showpoint;
15
16   #include <fstream>
17   using std::ofstream;
18   using std::ostream;
19   using std::fstream;
20
21   #include <iomanip>
22   using std::setw;
```

```
23   using std::setprecision;
24
25   #include <cstdlib>
26   using std::exit; // exit function prototype
27
28   #include "ClientData.h" // ClientData class definition
29
30   int enterChoice();
31   void createTextFile( fstream& );
32   void updateRecord( fstream& );
33   void newRecord( fstream& );
34   void deleteRecord( fstream& );
35   void outputLine( ostream&, const ClientData & );
36   int getAccount( const char * const );
37
38   enum Choices { PRINT = 1, UPDATE, NEW, DELETE, END };
39
40   int main()
41   {
42      // open file for reading and writing
43      fstream inOutCredit( "credit.dat", ios::in | ios::out |
ios::binary );
44
45      // exit program if fstream cannot open file
46      if ( !inOutCredit )
47      {
48         cerr << "File could not be opened." << endl;
49         exit ( 1 );
50      } // end if
51
52      int choice; // store user choice
53
54      // enable user to specify action
55      while ( ( choice = enterChoice() ) != END )
56      {
57         switch ( choice )
58         {
59            case PRINT: // create text file from record file
60               createTextFile( inOutCredit );
61               break;
62            case UPDATE: // update record
63               updateRecord( inOutCredit );
64               break;
65            case NEW: // create record
66               newRecord( inOutCredit );
67               break;
68            case DELETE: // delete existing record
69               deleteRecord( inOutCredit );
70               break;
71            default: // display error if user does not select
valid choice
72               cerr << "Incorrect choice" << endl;
73               break;
74         } // end switch
75
76         inOutCredit.clear(); // reset end-of-file indicator
77      } // end while
78
79      return 0;
80   } // end main
81
82   // enable user to input menu choice
83   int enterChoice()
```

```
84    {
85       // display available options
86       cout << "\nEnter your choice" << endl
87          << "1 - store a formatted text file of accounts" <<
endl
88          << " called \"print.txt\" for printing" << endl
89          << "2 - update an account" << endl
90          << "3 - add a new account" << endl
91          << "4 - delete an account" << endl
92          << "5 - end program\n? ";
93
94       int menuChoice;
95       cin >> menuChoice; // input menu selection from user
96       return menuChoice;
97    } // end function enterChoice
98
99    // create formatted text file for printing
100   void createTextFile( fstream &readFromFile )
101   {
102      // create text file
103      ofstream outPrintFile( "print.txt", ios::out );
104
105      // exit program if ofstream cannot create file
106      if ( !outPrintFile )
107      {
108         cerr << "File could not be created." << endl;
109         exit( 1 );
110      } // end if
111
112      outPrintFile << left << setw( 10 ) << "Account" <<
setw( 16 )
113         << "Last Name" << setw( 11 ) << "First Name" <<
right
114         << setw( 10 ) << "Balance" << endl;
115
116      // set file-position pointer to beginning of
readFromFile
117      readFromFile.seekg( 0 );
118
119      // read first record from record file
120      ClientData client;
121      readFromFile.read( reinterpret_cast< char * >( &client
),
122         sizeof( ClientData ) );
123
124      // copy all records from record file into text file
125      while ( !readFromFile.eof() )
126      {
127         // write single record to text file
128 if ( client.getAccountNumber() != 0 ) // skip empty records
129            outputLine( outPrintFile, client );
130
131         // read next record from record file
132    readFromFile.read( reinterpret_cast< char * >( &client ),
133            sizeof( ClientData ) );
134      } // end while
135   } // end function createTextFile
136
137   // update balance in record
138   void updateRecord( fstream &updateFile )
139   {
140      // obtain number of account to update
141      int accountNumber = getAccount( "Enter account to
```

```
update" );
142
143     // move file-position pointer to correct record in file
144     updateFile.seekg( ( accountNumber - 1 ) * sizeof(
ClientData ) );
145
146     // read first record from file
147     ClientData client;
148     updateFile.read( reinterpret_cast< char * >( &client ),
149        sizeof( ClientData ) );
150
151     // update record
152     if ( client.getAccountNumber() != 0 )
153     {
154        outputLine( cout, client ); // display the record
155
156        // request user to specify transaction
157        cout << "\nEnter charge (+) or payment (-): ";
158        double transaction; // charge or payment
159        cin >> transaction;
160
161        // update record balance
162        double oldBalance = client.getBalance();
163        client.setBalance( oldBalance + transaction );
164        outputLine( cout, client ); // display the record
165
166        // move file-position pointer to correct record in
file
167        updateFile.seekp( ( accountNumber - 1 ) * sizeof(
ClientData ) );
168
169        // write updated record over old record in file
170        updateFile.write( reinterpret_cast< const char * >(
&client ),
171           sizeof( ClientData ) );
172     } // end if
173     else // display error if account does not exist
174        cerr << "Account #" << accountNumber
175           << " has no information." << endl;
176  } // end function updateRecord
177
178  // create and insert record
179  void newRecord( fstream &insertInFile )
180  {
181     // obtain number of account to create
182     int accountNumber = getAccount( "Enter new account
number" );
183
184     // move file-position pointer to correct record in file
185     insertInFile.seekg( ( accountNumber - 1 ) * sizeof(
ClientData ) );
186
187     // read record from file
188     ClientData client;
189     insertInFile.read( reinterpret_cast< char * >( &client
),
190        sizeof( ClientData ) );
191
192     // create record, if record does not previously exist
193     if ( client.getAccountNumber() == 0 )
194     {
195        char lastName[ 15 ];
196        char firstName[ 10 ];
```

```
197         double balance;
198
199         // user enters last name, first name and balance
200         cout << "Enter lastname, firstname, balance\n? ";
201         cin >> setw( 15 ) >> lastName;
202         cin >> setw( 10 ) >> firstName;
203         cin >> balance;
204
205         // use values to populate account values
206         client.setLastName( lastName );
207         client.setFirstName( firstName );
208         client.setBalance( balance );
209         client.setAccountNumber( accountNumber );
210
211         // move file-position pointer to correct record in
file
212         insertInFile.seekp( ( accountNumber - 1 ) * sizeof(
ClientData ) );
213
214         // insert record in file
215         insertInFile.write( reinterpret_cast< const char *
>( &client ),
216            sizeof( ClientData ) );
217      } // end if
218      else // display error if account already exists
219         cerr << "Account #" << accountNumber
220            << " already contains information." << endl;
221   } // end function newRecord
222
223   // delete an existing record
224   void deleteRecord( fstream &deleteFromFile )
225   {
226      // obtain number of account to delete
227      int accountNumber = getAccount( "Enter account to
delete" );
228
229      // move file-position pointer to correct record in file
230      deleteFromFile.seekg( ( accountNumber - 1 ) * sizeof(
ClientData ) );
231
232      // read record from file
233      ClientData client;
234      deleteFromFile.read( reinterpret_cast< char * >(
&client ),
235         sizeof( ClientData ) );
236
237      // delete record, if record exists in file
238      if ( client.getAccountNumber() != 0 )
239      {
240         ClientData blankClient; // create blank record
241
242         // move file-position pointer to correct record in
file
243         deleteFromFile.seekp( ( accountNumber - 1 ) *
244            sizeof( ClientData ) );
245
246         // replace existing record with blank record
247         deleteFromFile.write(
248            reinterpret_cast< const char * >( &blankClient ),
249            sizeof( ClientData ) );
250
251         cout << "Account #" << accountNumber << "
deleted.\n";
```

```
252     } // end if
253     else // display error if record does not exist
254        cerr << "Account #" << accountNumber << " is
empty.\n";
255  } // end deleteRecord
256
257  // display single record
258  void outputLine( ostream &output, const ClientData &record
)
259  {
260     output << left << setw( 10 ) <<
record.getAccountNumber()
261        << setw( 16 ) << record.getLastName()
262        << setw( 11 ) << record.getFirstName()
263        << setw( 10 ) << setprecision( 2 ) << right << fixed
264        << showpoint << record.getBalance() << endl;
265  } // end function outputLine
266
267  // obtain account-number value from user
268  int getAccount( const char * const prompt )
269  {
270     int accountNumber;
271
272     // obtain account-number value
273     do
274     {
275        cout << prompt << " (1 - 100): ";
276        cin >> accountNumber;
277     } while ( accountNumber < 1 || accountNumber > 100 );
278
279     return accountNumber;
280  } // end function getAccount
```

Program 4.5: Main transaction-processing program

The program presents a menu driven interface to the user. The choices available to the user are 1-Print, 2-Update, 3-New account, 4- Delete account and 5-End processing. These menu choices are realized through following five options:

Option1: calls function createtextFile to store a formatted list of all account information in a text file called print.txt that may be printed. The function createTextFile takes an fstream object as an argument to be used to input data from the credit.dat file. It invokes istream member function read and uses sequential access to input data from credit.dat. The function outputLine is used to output the data to file print.txt. Note that the createTextFile uses istream member function seekg to ensure that the file-position pointer is at the beginning of the file.

```
Enter your choice
1 - store a formatted text file of accounts
2 - update an account
3 - add a new account
4 - delete an account
5 - end program

1

Account     Last Name      First Name    Balance
29          Tiwari         Nisha         -24.54
33          Ghosh          Soumitra      314.33
37          Singh          Shweta        0.0
88          Sen            Ajay          258.34
96          Jolly          Stellina      34.98
```

Option2 calls updateRecord to update an account. This function updates only an existing record, so the function first determines whether the specified record is empty. Lines 128-129 read data into object client, using istream member function read. Then line 132 compares the value returned by getAccountNumber of the client object to zero to determine whether the record contains information. If this value is zero, lines 154-155 print an error message indicating that the record is empty. If the record contains information, line 134 displays the record, using function outputLine, line 139 inputs the transaction amount and lines 142-151 calculate the new balance and rewrite the record to the file. A typical output for option 2 is:

```
Enter your choice
1 - store a formatted text file of accounts
2 - update an account
3 - add a new account
4 - delete an account
5 - end program

2

Enter account to update (1 - 100) : 37
37          Singh          Shweta        0.0
Enter charge (+) or payment (-): +87.9988
37          Singh          Shweta        87.99
```

Option3 calls function newrecord (lines 159-201) to add a new account to the file. If the user enters an account number for an existing account, newRecord displays an error message indicating that the account exists (lines 199-200). A typical output for option 3 is:

```
Enter your choice
1 - store a formatted text file of accounts
2 - update an account
3 - add a new account
4 - delete an account
5 - end program

3


Enter new account number (1 - 100) : 22

Enter lastname, firstname, balance

?          Popli        Sukanya      247.45
```

Option4 calls function deleteRecord (lines 204-235) to delete a record from the file. Line 207 prompts the user to enter the account number. Only an existing record may be deleted, so if the specified account is empty, line 234 displays an error message. If the account exists, lines 227-229 reinitialize that account by copying an empty record (blank-Client) to the file. Line 231 displays a message to inform the user that the record has been deleted. A typical output for option 4 is:

```
Enter your choice
1 - store a formatted text file of accounts
2 - update an account
3 - add a new account
4 - delete an account
5 - end program

4

Enter account to delete (1 - 100) : 29

Account #29 deleted.
```

Option5 terminating the program.

```
Enter your choice
1 - store a formatted text file of accounts
2 - update an account
3 - add a new account
4 - delete an account
5 - end program

5
```

This transaction-processing system presents an example of a large multi-file program. The program demonstrates use of various features of C++ ranging from classes and methods, constructors, polymorphism, templates and stream I/O capabilities. The program illustrates how C++ language features can be used to solve real world applications by designing and deploying C++ programs.

It would also be in order to discuss here the fact that the sequential files are inappropriate for instant-access applications, in which a particular record must be located immediately. Common instant-access applications are airline reservation systems, banking systems, point-of-sale systems, automated teller machines and other kinds of transaction-processing systems that require rapid access to specific data. A bank might have hundreds of thousands (or even millions) of other customers, yet, when a customer uses an automated teller machine, the program checks that customer's account in a few seconds or less for sufficient funds. This kind of instant access is made possible with random-access files. Individual records of a random-access file can be accessed directly (and quickly) without having to search other records. As we have said, C++ does not impose structure on a file. So the application that wants to use random-access files must create them. A variety of techniques can be used. Perhaps the easiest method is to require that all records in a file be of the same fixed length. Using same-size, fixed-length records makes it easy for a program to calculate (as a function of the record size and the record key) the exact location of any record relative to the beginning of the file. Using a database based approach is a common choice for many of these applications.

C++ provides rich set of features for designing various applications to solve various real world problems. A number of useful applications in different domain cab be designed using C++. Applications like passenger reservation systems, automated plant control, restaurant management, library management are some possible applications which can be implemented using C++ features.

4.5 SUMMARY

In the previous chapters, we have discussed various features of C++. C++ provides a rich set of features and capabilities that can be used to write useful programs to solve a number of real world problems. This unit presents a case study of designing and implementing a transaction- processing system in banking domain. The design involves creating necessary data files to store accounts and customer information and then accessing them through suitable code for writing data, appending data, performing credit operations and displaying the results. The program design makes use of various features of C++, including its capability to design multi-file programs. The file processing capability of C++ coupled with the rich I/O capability through stream classes can be used to design many interesting applications. This unit demonstrated use and application of various C++ features for solving one real world large scale problem. C++ can be used to solve many other simple and sophisticated real world problems.

4.6 FURTHER READINGS

1. E. Balaguruswamy, *Object Oriented Programming with C++*, Tata McGraw Hill, 2010.
2. P. Deitel and H. Deitel, *C++: How to Program*, PHI, 7th ed, 2010.
3. B. Strousstrup, *Programming – Principles and Practices using C++*, Addison Wesley, 2009.
4. R. Lafore, *Object Oriented Programming in TURBO C++*, Galgotia Publications, 1994.