Unit –IV

Working with files

Managing Console I/O Operations – Working with Files – Templates – Exception Handling

A file is a collection of related data stored in a particular area on the disk. Programs can be designed to perform the read and write operations on these files.

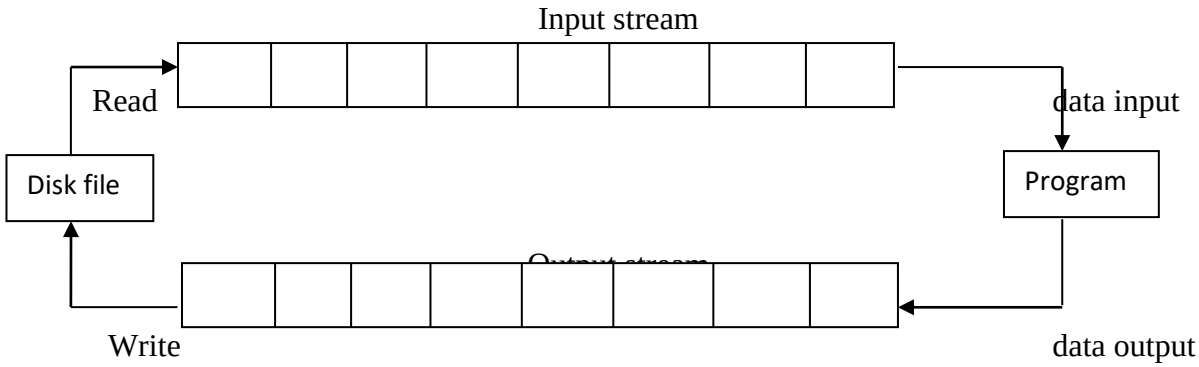
A file program involves the following kinds of communication

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

The I/O system of C++ handles file operations using **file streams as an interface** between the programs and files. Two types of streams are

1. Input Stream
2. Output Stream

Input stream extracts data from the file and the output stream inserts data to the file.



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

The I/O of C++ contains a set of classes that define the file handling methods. These classes are derived from **fstreambase** and from the corresponding **iostream** class.

Details of file stream classes

Class	Description
Fstreambase	• Provides operations common to the file streams • Base for fstream, ifstream and ofstream class • Contains open() and close() function
Ifstream	• Provides input operation • Contains open() with default input mode • Inherits function get(),getline(),read(),seekg() and tellg() from istream
Ofstream	• Provides output operation • Contains open() with default input mode • Inherits function put(),seekp(), tellp() and write() function from ostream
Fstream	• Provides support for simultaneous input and output operations • Contains open() with default input mode • Inherits functions from istream and ostream classes through iostream
Filebuf	• Set the file buffers to read and write • Contain close() and open() as members

Opening and closing a file

To use a file, suitable name for the file should be specified. **File name** is a string of characters that make up a valid file name for operating system. It contains two parts a primary name and an optional period with extension.

Ex: Test.doc, input.data

For opening a file, a file stream is created and it is linked to the filename. A file stream can be defined using the classes ifstream, ofstream and fstream that are contained in the header file fstream. The class to be used depends upon the purpose whether to read data from the file or write data to it.
A file can be opened in two ways

- Using the constructor function of the class
    - It is useful when only one file is used in the stream
- Using the member function open() of the class.
    - It is useful to manage multiple files using one stream.

Opening files using constructor

1. Create a file stream object to manage the stream using the appropriate class. The class ofstream is used to create the output stream and the class ifstream to create the input stream.

2. Initialize the file object with the desired filename.

   Example 1: Statement opens a file named “results” for output.

   ofstream outfile(“results”);

   Example 2: Statement declares infile as an ifstream object and attaches it to the data for reading.

   ifstream infile(“data”)

Although there is a single program two file stream objects, outfile (to put data to the file) and infile (to get data from the file)

Note: when a file is opened for writing only, a new file is created if there is no file of that name. If a file by that name already exists then its contents are deleted and the file is presented as a clean file.

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
void main()
{
clrscr();
int mk1,mk2,mk3;
ofstream outf("student.txt");
cout<<"Enter mark1,mark2,mark3\n";
cin>>mk1;
outf<<mk1<<"\n";
cin>>mk2;
outf<<mk2<<"\n";
cin>>mk3;
outf<<mk3<<"\n";
outf.close();
ifstream inf("student.txt");
inf>>mk1;
inf>>mk2;
inf>>mk3;
cout<<"MARK1=\n"<<mk1<<endl;
cout<<"MARK2=\n"<<mk2<<endl;
cout<<"MARK3=\n"<<mk3<<endl;
inf.close();
getch();
}
```

Output

```
Enter mark1, mark2, mark3
25
```

22
23

Opening files using open( )

The function open() can he used to open multiple files that use the same stream object. For example a user may want to process a set of files sequentially. In such cases user may create a single stream object and use it to open each file in turn. This is done as follows.

file-stream-class stream-object;
stream-object.open(“filename”);

Example

```
ofstream outfile;
outfile.open(“DATA1”);
.....
....
outfile.close();
outfile.open(“DATA2”);
.....
....
outfile.close();
```

Note: First file is closed before opening the second one. This is because a stream can be connected to only one file at a time.

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
void main()
{
clrscr();
int im1,im2,im3;
int em1,em2,em3;
ofstream outf;
outf.open("internal.txt");
cout<<"Enter internal marks:\n mark1,mark2,mark3\n";
cin>>im1;
outf<<im1<<"\n";
cin>>im2;
outf<<im2<<"\n";
cin>>im3;
outf<<im3<<"\n";
outf.open("external.txt");
cout<<"Enter External marks:\n mark1,mark2,mark3\n";
cin>>em1;
outf<<em1<<"\n";
cin>>em2;
outf<<em2<<"\n";
```

```
cin>>em3;
outf<<em3<<"\n";
outf.close();
ifstream inf;
inf.open("internal.txt");
inf>>im1;
inf>>im2;
inf>>im3;

inf.open("external.txt");
inf>>em1;
inf>>em2;
inf>>em3;

cout<<"Total marks are\n";
cout<<"MARK1=\n"<<em1<<endl;
cout<<"MARK2=\n"<<em2<<endl;
cout<<"MARK3=\n"<<em3<<endl;

inf.close();
getch();
}
```

Output

```
Enter internal marks
Enter mark1, mark2, mark3
15
22
23
Enter External marks
Enter mark1, mark2, mark3
60
55
50
Total marks are
MARK1=75
MARK2=77
MARK3=73
```

Open (): File modes

While opening files using constructor or open ( ) function, file opening mode can also be specified along with file name.

Syntax

Stream-object.open(“filename”,mode)

Mode – specifies the purpose for which the file is opened. If the mode is absence in syntax then default values are assumed.

The default values are

ios::in for ifstream functions meaning open for reading only

ios::out for ofstream functions meaning open for writing only

Several file mode constants are defined in ios class of C++ I/O System.

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

Note

1. Opening a file in ios::out mode also opens it in ios::trunc mode
2. Both ios::app and ios::ate take to the end of the file when it is opened. But ios::app add data to the end of the file only. ios::ate add data or modify existing data anywhere in the file.
3. When fstream class is used, mode should be specified explicitly.
4. The mode can combine two or more parameters using bitwise OR operator.
   Ex: This opens the file in the append mode but fails to open the file if it does not exist.

File pointers and their manipulations

Each file has two associated pointers known as the file pointers. These pointers can be used to move through files while reading and writing. They are

1. input pointer or get pointer – reading the contents of a given file location
2. output pointer or put pointer – writing to a given file location

Each time an input or output operation takes place the appropriate pointer is automatically advanced

Default pointer position

- When a file is opened in read-only mode the input pointer is automatically set at the beginning.
- When a file is opened in write-only mode the existing contents are deleted and the output pointer is set at the beginning
- When an existing file is opened in append mode it moves the output pointer to the end of the file

Functions for manipulation of file pointers

To move the file pointers to desired position, file stream classes support a set of functions. They are

Function	Description
seekg()	Moves get pointer to a specified location
seekp()	Moves put pointer to a specified location
tellg()	Gives the current position of the get pointer
tellp()	Gives the current position of the put pointer

For example infile.seekg(10) moves the file pointer to the byte number 10.

Seek functions seekg() and seekp() can also use two arguments as follows

seekg(offset,refposition)

seekp(offset,refposition)

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

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

Examples

fout.seekg(0,ios::end)	- go to the end of file
fout.seekg(m,ios::cur)	- go forward by m bytes from the current position
fout.seekg(-m,ios::end)	- go backward by m bytes from the end

Sequential input and output operations

The file stream classes support a number of member functions for performing input and output operations on files. They are

1. Put( ) and get( ) - read singe character at a time
2. Write( ) and read( ) - write and read blocks of binary data

Get and Put functions

Put( )- writes a single character to its associated file stream

Get( ) – reads a single character from its associated file stream.

Example: Program to display a file content on the screen.

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
#include<string.h>
void main()
{
char str[50];int i;char ch;
clrscr();
cout<<"enter any string\n";
cin>>str;
fstream file;
file.open("get.txt",ios::in|ios::out);
for(i=0;i<strlen(str);i++)
{
file.put(str[i]);
}
file.seekg(0);
cout<<"Output string is\n";
while(file)
{
file.get(ch);
cout<<ch;
}
getch();
}
```

Write and Read functions program

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
#include<string.h>
void main()
{
int a[5]={10,20,30,40,50};int i;
clrscr();
fstream file;
file.open("mark.txt",ios::in|ios::out);
for(i=0;i<5;i++)
{
file.write((char *) &a,sizeof(a));
}
file.seekg(0);
cout<<"Entered marks are\n";
while(file)
{
for(i=0;i<5;i++)
{
file.read((char *) &a,sizeof(a));
cout<<a[i];
}
```

```
}
file.close();
getch();
}
```

Write and Read functions program – using class and object

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
#include<string.h>
class student
{
int m1,m2;
public:
void get()
{
cout<<"enter m1,m2\n";
cin>>m1>>m2;
}
void display()
{
cout<<"Marks are\n";
cout<<m1<<endl<<m2<<endl;
}
};

void main()
{
student s;
clrscr();
fstream file;
file.open("mark1.txt",ios::in|ios::out);
s.get();
file.write((char *) &s,sizeof(s));
file.seekg(0);
cout<<"Entered marks are\n";
file.read((char *) &s,sizeof(s));
s.display();
file.close();
getch();
}
```

TEMPLATES

Using templates, it is possible to create generic functions and classes. In a generic function or class, the type of data upon which the function or class operates is specified as a parameter.

Thus, we can use one function or class with several different types of data without having to explicitly recode specific versions for each data type.

Generic functions (Function Templates)

A generic function defines a general set of operations that will be applied to various types of data. The type of data that the function will operate upon is passed to it as a parameter. Through a generic function, a single general procedure can be applied to a wide range of data.

The general form of a template function definition is shown here:

template <class Ttype>

return-type func-name (Ttype a1, Ttype a2,……., Ttype n)

{

// body of function

}

Here, Ttype is a placeholder name for a data type used by the function.

Write a generic function swap to interchange any two variables (integer, character, and float).

```
#include <iostream.h>
template <class T>
void swap(T p, T q)
{
T temp;
temp = p;
p = q;
q = temp;
cout<<p<<"\t"<<q;
}
void main()
{
int i=10, j=20;
float x=10.1, y=23.3;
char a='x', b='z';
swap (i, j); /*swaps integers*/
swap (x, y); /* swaps floats*/
swap (a, b); /*swaps chars*/
}
```

Output:

20 10
23.2 10.1
z x

Note: The line: template <class T> void swap (T p, T q) tells the compiler two things: that a template is being created and that a generic definition is beginning. Here, T is a generic type that is used as a placeholder. After the template portion, the function swap () is declared, using T as the data type of the values that will be swapped. In main () , the swap () function is called using three different types of data: ints, floats, and chars. Because swap () is a generic function, the compiler automatically

creates three versions of swap (): one that will exchange integer values, one that will exchange floating-point values, and one that will swap characters.

A Function with Two Generic Types:

We can define more than one generic data type in the template statement by using a comma separated list. For example; below program creates a template function that has two generic types.

```
#include <iostream.h>
template <class T1, class T2>
void myfunc (T1 x, T2 y)
{
cout << x << "\t" << y << "\n";
}
void main()
{
myfunc (10, "I like C++");
myfunc (98.6, 19);
}
```

Output:

```
10 I like C++
98.6 19
```

Generic Function Restrictions

Generic functions are similar to overloaded functions except that they are more restrictive. When functions are overloaded, you may have different actions performed within the body of each function. But a generic function must perform the same general action for all versions- only the type of data can differ.

Generic class (class templates)

In addition to generic functions, we can also define a generic class. When we do this, we create a class that defines all the algorithms used by that class; however, the actual type of the data being manipulated will be specified as a parameter when objects of that class are created.

Generic classes are useful when a class uses logic that can be generalized. For example, the same algorithms that maintain a queue of integers will also work for a queue of characters, and the same mechanism that maintains a linked list of mailing addresses will also maintain a linked list of auto part information.

The general form of a generic class declaration is:

```
template <class T>
class class-name
{
```

};

General form of a member function definition of template class:

template <class T>

Ret_type class_name <T>:: function()

{

}

General form of object creation of a template class:

class_name <data_type> object1, object2,…….

Write a program to add two numbers (either two integers or floats) using class templates.

```
#include <iostream.h>
template <class T>
class Add
{
T a, b;
public:
void getdata();
void display();
};
template <class T>
void Add <T>::getdata( )
{
cout<<”Eneter 2 nos”;
cin>>a>>b;
}
template <class T>
void Add <T>::display( )
{
cout<<”sum=”<<a+b;
}
void main()
```

```
{
Add <int> ob1;
Add <float> ob2;
ob1.getdata( );
ob1.display( );
ob2.getdata( );
ob2.display( );
}
```

Output:

```
Eneter 2 nos 4 5
Sum=9
Eneter 2 nos 4.8 5.1
Sum=9.9
```

Exception handling

The two most common types of bugs happen in a program are logic errors and syntactic errors.

Logic errors – Poor understanding of problem and solution procedure
Syntactic errors – Poor understanding of language

Debugging and testing procedure can solve this.

We often come across some peculiar problems other than logic or syntax errors. They are known as exceptions. Exceptions are runtime unusual conditions that a program may encounter while executing.

Example: Division by zero, access to an array outside of its bounds, running out of memory or disk space.

When a program encounters an exceptional condition it is important that it is identified and dealt with effectively.

Basics of exception handling

Exceptions are of two kinds. They are,

1. Synchronous exception
   "Out-of-range-index" and "over-flow" are belonging to this type.
2. Asynchronous exception
   Errors caused by events beyond the control of the program. Ex: keyboard interrupts

Exception handling mechanism in C++ is designed to handle only **synchronous exception**. Purpose of exception handling mechanism is to provide means to detect and report an "exceptional circumstance" so that appropriate action can be taken.

It involves the following steps

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

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

Exception handling mechanism

This mechanism is basically built upon tree keywords

1. Try
2. Throw
3. Catch

Try is used to preface a bloc of statements which may generate exceptions. This block is called as try block.

When an exception is detected, it is thrown using a throw statement in the try block. A catch block defined by the keyword catch 'catches' the exception and handles it appropriately.

The general form of these two blocks is

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

Note: Exceptions are objects used to transmit information about a problem.

When the try block throws an exception the program control leaves the try block and leaves the catch statement of the catch block.

If the type of object thrown matches the arg type in catch statement catch block is executed for handling the exception.

If they do not match the program is aborted with the help of the abort () function which is invoked by default. When no exception is detected and thrown control goes to the statement immediately after catch block.

Multiple catch statements

It is possible that a program segment has more than one condition to throw and exception. In such cases, we can associate more than one catch statement with a try as shown below.

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

When an exception is thrown, the exception handlers are searched in order for an appropriate match. The first handler that yields a match is executed.

After executing the handler, the controller goes to the first statement after the last catch block for that try.

When no match is found, the program is terminated.

Catch all exceptions

In some situations it may not be able to anticipate all possible types of exception and therefore may not be able to design independent catch handlers to catch them.

In such case a catch statement is forced to catch all exceptions instead of a certain type alone.

```
catch (.....)
{
.....
```

.....
}

Specifying exception

It is possible to restrict a function to throw only certain specified exception. This is achieved by adding a throw list clause to the function definition.

The general form is

Type functions (arg-list) throw (type-list)

{

Function body

}

Type-list specifies the type of exception that may be thrown. Throwing any other type of exception will cause abnormal program termination

Ex: void test(int x) throw(int, double)

Formatted console I/O operations

C++ supports a number of features that could be used for formatting the output. These features include

- ios class functions and flags
- Manipulators
- User-defined output functions

The ios class contains a large number of member functions that would help us to format the output in a number of ways. The most important ones among them are

Function	Task
width ( )	To specify the required field size for displaying an output value
precision( )	To specify the number of digits to be displayed after the decimal point or a float value
fill( )	To specify a character that is used to fill the unused portion of a field.
setf( )	To specify format flags that can control the form of output display(such as left-justification and right-justification)
unsetf( )	To clear the flags specified.

Manipulators are special functions that can be included in the I/O statements to alter the format parameters of a stream. To access these manipulators, the file iomanip should be included in the program. The most important manipulators are,

Manipulators	Equivalent ios function
setw( )	width ( )
setprecision( )	precision( )
setfill( )	fill( )
setiosflags( )	setf( )
resetiosflags( )	unsetf( )

In addition to these functions supported by the C++ library user can create their own manipulator functions to provide any special output formats.

Defining field width: width ( )

Width ( ) function is used to define the width of a field necessary for the output of an item. Since it is a member function, an object is used to invoke it as shown below

cout.width(w)

Where w is the field width (number of columns). The output will be printed in a field of w characters wide at the right end of the field.

The width( ) function can specify the field width for only one item(the item that follows immediately).

Example

```
cout.width(5)
cout<<543<<12<<"\n";
```

will produce the following output

		5	4	3	1	2

The value 543 is printed right-justified in the first five columns. The specification width(5) does not retain the setting for printing the number 12. This can be improved as follows.

```
cout.width(5)
cout<<543;
cout.width(5);
cout<<12<<"\n";
```

This produces the following output

		5	4	3				1	2

C++ never truncates the values and therefore if the specified field width is smaller than the size of the value to be printed, C++ expands the field to fit the value.

Setting precision: precision( )

By default, the floating numbers are printed with six digits after the decimal point. However user can specify the number of digits to be displayed after the decimal point while printing the floating-point numbers. This can be done by using the precision( ) member function as

cout.precision(d)

where d is the number of digits to the right of the decimal point.

For example

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

will produce the following output

1.141

3.142

precision( ) retains the setting in effect until it is reset. It is possible to combine the field specification with the precision setting.

Example

```
cout.precision(2);
cout.width(5);
cout<<1.2345
```

It instructs "Print two digits after the decimal point in a field of five character width". Thus output will be

	1	**.**	**2**	**3**

Filling and Padding: fill( )

While printing the values using much larger field widths than required by the values the unused positions of the field are filled with whitespaces by default. However we can use the fill() function to fill the unused positions by any desired character. It is used in the following form

cout.fill(ch);

where ch represents the character which is used for filling the unused positions.

cout.fill('*');

cout.width(10);

cout<<5250<<"\n";

The output would be

*	*	*	*	*	*	**5**	**2**	**5**	**0**

Formatting flags, Bit-fields and setf( )

When the function width( ) is used, the value is printed right-justified in the field width created.

But is a usual practice to print the text left-justified. How do we get a floating point number printed in the scientific notation?

The setf(), a member function of the ios class, can provide answers to these and many other formatting questions. The setf() (setf stands for set flags) function can be used as follows

cout.setf(arg1,arg2)

The arg1 is one of the formatting flags defined in the class ios. The formatting flag specifies the format action required for the output. Arg2 known as bit field specifies the group to which the formatting flag belongs

UNIT V

Standard Template Library – Manipulating Strings – Object Oriented Systems Development

STANDARD TEMPLATE LIBRARY (STL)

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms, and iterators. It is a generalized library and so, its components are parameterized.

COMPONENTS OF STL

STL has three components

- Algorithms
- Containers
- Iterators

Algorithms

The header algorithm defines a collection of functions especially designed to be used on ranges of elements. They act on containers and provide means for various operations for the contents of the containers. STL includes many different algorithms to provide support to take such as initializing, searching, popping, sorting, merging and copying. · They are implemented by template functions.

Containers

Containers or container classes store objects and data. There are in total seven standard "first-class" container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.

Types of containers

1. Sequence Containers

- They store elements in a linear sequence like a line.
- Each element is related to other elements by its position along the line.
- They all expand themselves through allow insertion of elements and support a number of operations.
- Some types of sequence container are vector, list, deque, etc

Vector

· It is a dynamic array.

· It allows insertion & deletion at back & permits direct access to any element.

List

· It is a bidirectional linear list.

· It allows insertion and deletion anywhere in the list.

Dequeue

· It is a double ended queue.

· It allows insertion and deletion at both ends.

1. **Associative Container**

· They are design to support direct access to elements using keys. They are 4 types.

Set

· It is an associative container for storing unique sets.

· Here, is no duplicate are allowed.

Multisets

· Duplicates are allowed.

Map

· It is an associate container for storing unique key.

· Each key is associated with one value.

Multimap

- It is an associate container for storing key value pairs in which one key may be associated with more than one value.
- We can search for a desired student using his name as the key.
- The main difference between a map and multimap is that, a map allows only one key for a given value to be stored while multimap permits multiple key.

3. Derived Container

· STL provides 3 derived container, stack, queue, priority queue. They are also known as container adaptor. They can be created from different sequence container.

Example: queue, priority_queue, stack

4. **Unordered Associative Containers** : implement unordered data structures that can be quickly searched

Iterators

- Iterators are used for working upon a sequence of values. They are the major feature that allows generality in STL. It is an object like a pointer that points to an element in a container. Iterators are used to move through the contents of container. Just like pointers it can be incremented or decremented.

- The STL implements five different types of iterators.

- These are **input iterators** (that can only be used to read a sequence of values), **output iterators** (that can only be used to write a sequence of values), **forward iterators** (that can be read, written to, and move forward), **bidirectional iterators** (that are like forward iterators, but can also move backwards) and **random access iterators** (that can move freely any number of steps in one operation).

OBJECT ORIENTED SYSTEMS DEVELOPMENT

CLASSICAL SOFTWARE DEVELOPMENT LIFE CYCLE

Software development life cycle (**SDLC**) is a series of phases that provide a common understanding of the software building process. The good software engineer should have enough knowledge on how to choose the SDLC model based on the project context and the business requirements.

Requirements analysis

This phase focuses on the requirements of the software to be developed. It determines the processes that are to be incorporated during the development of the software. To specify the requirements, users' specifications should be clearly understood and their requirements be analyzed. This phase involves interaction between the users and the software engineers and produces a document known as Software Requirements Specification **(SRS).**

Design

This phase determines the detailed process of developing the software after the requirements have been analyzed. It utilizes software requirements defined by the user and translates them into software representation. In this phase, the emphasis is on finding solutions to the problems defined in the requirements analysis phase.

Coding

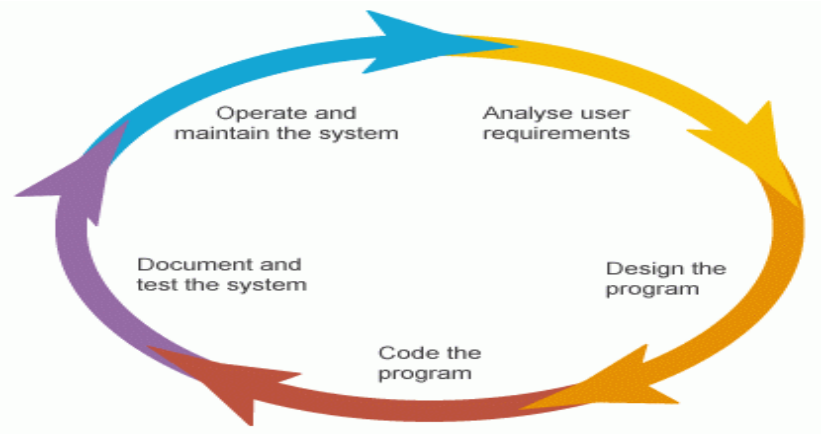
This phase emphasizes translation of design into a programming language using the coding style and guidelines. The programs created should be easy to read and understand. All the programs written are documented according to the specification.

Testing

This phase ensures that the software is developed as per the user's requirements. Testing is done to check that the software is running efficiently and with minimum errors. It focuses on the internal logic and external functions of the software and ensures that all the statements have been exercised (tested). Note that testing is a multistage activity, which emphasizes verification and validation of the software.

Implementation and maintenance

This phase delivers fully functioning operational software to the user. Once the software is accepted and deployed at the user's end, various changes occur due to changes in the external environment. The changes also occur due to changing requirements of the user and changes occurring in the field of technology. This phase focuses on modifying software, correcting errors, and improving the performance of the software.



OBJECT-ORIENTED SOFTWARE DEVELOPMENT

In object-oriented software engineering, the software developer identifies and organizes the application in terms of object-oriented concepts, prior to their final representation in any specific programming language or software tools. The major phases of software development using object–oriented methodology are **object-oriented analysis, object-oriented design, and object-oriented implementation.**

Object–Oriented Analysis

In this stage, the problem is formulated, user requirements are identified, and then a model is built based upon real–world objects. The analysis produces models on how the desired system should function and how it must be developed. The models do not include any implementation details so that it can be understood and examined by any non–technical application expert.

Steps in object-oriented Analysis

Problem understanding

First step in analysis process is to understand the problem of user. The problem should be refined and redefined in terms of computer system engineering to suggest a computer-based solution. The problem statement provides the basis for drawing the requirements specification of both the user and the software.

Requirement specification

Once the problem is clearly defined a list of user requirement is generated. A clear understanding should exist between the user and the developer of what is required. Based on user requirements the specification for the software should be drawn. The develop should state clearly,

What outputs are required?, What processes are involved to produce these outputs?, What inputs are necessary?, What resources are required?, etc.

This specification serves as a reference to test the final product.

Identification of objects

The best place to look for objects is the application itself. The application may be analyzed by using one of the following two approaches.

1. Dataflow diagram (DFD)
2. Textual Analysis (TA)

Data Flow diagram

A dataflow diagram indicates how the data moves from one point to another in the system. The boxes and data stores in the data flow diagram are gold candidates for the objects. The process bubbles correspond to the procedures.

Textual Analysis

It is based on the textual de3scription of the problem or proposed solution. The description may be of one or two sentences or one or two paragraphs depending on the type and complexity of the problem. Using one of the above approaches,

1. Prepare an object table
2. Identify the objects that belong to the solutions space.
3. Identify the attributes of the solution space objects.

Identification of services

Once the objects in the solution space have been identified the next step is to identify a set of services that each object should offer.

OBJECT–ORIENTED DESIGN

Object-oriented design includes two main stages, namely, **system design and object design.**

System Design

In this stage, the complete architecture of the desired system is designed. The system is conceived as a set of interacting subsystems that in turn is composed of a hierarchy of interacting objects, grouped into classes. System design is done according to both the system analysis model and the proposed system architecture. Here, the emphasis is on the objects comprising the system rather than the processes in the system.

Object Design

In this phase, a design model is developed based on both the models developed in the system analysis phase and the architecture designed in the system design phase. All the classes required are identified. The designer decides whether

- new classes are to be created from scratch,
- any existing classes can be used in their original form, or
- New classes should be inherited from the existing classes.

Reusability of classes from the previous designs, classification of objects into subsystems and determination of appropriate protocols are some of the considerations of the design stage. The OOD approach may involve the following steps

1. Review of objects created in the analysis phase
2. Specification of class dependences
3. Organization of class hierarchies
4. Design of classes
5. Design of member functions

Review of problem space objects

The main objective of this review is to refine the objects in terms of their attributes and operations and to identify other objects that are solution specific.

Class dependencies

It is to Analysis the relationship between the classes. It is important to identify appropriate classes to represent the objects and establish their relationships. The major relationships in design are

1. Inheritance relationships
2. Containment relationships
3. Use relationships

Organization of class hierarchies

Organization of the class hierarchies involves identification of common attributes and functions among a group of related classes and then combining them to form a new class. The new class will serve as the super-class and the others as subordinate-class.

This process may be repeated at different levels of abstraction with the sole objective of extending the classes. The Pictorial representation is

Design of classes

Some guidelines should be considered while designing classes are

1. A class should be dependent on as few classes as possible
2. Interaction between two classes must be explicit.
3. Each subordinate class should be designed as a specialization of the base class with the sole purpose of adding additional features.
4. The top class of a structure should represent the abstract model of the target concept
5. Function of a class should be defined as public interface.

Design of member function

The member function defines the operations that are performed on the objects data. Top-down functional decomposition technique is used to design them.

OBJECT–ORIENTED IMPLEMENTATION AND TESTING

In this stage, the design model developed in the object design is translated into code in an appropriate programming language or software tool. The databases are created and the specific hardware requirements are ascertained. Once the code is in shape, it is tested using specialized techniques to identify and remove the errors in the code.

C++ STRING CLASS AND ITS APPLICATIONS

A string defined as a sequence of characters. In C++ we can store string by one of the two ways

1. C style strings (or) Character Array
2. String class

C++ has in its definition a way to represent **sequence of characters as an object of class**. This class is called std:: string. String class stores the characters as a sequence of bytes with a functionality of allowing **access to single byte character**.

Difference between Character Array and String Class

Character Array	**String class**
A character array is simply an **array of characters** can terminated by a null character.	A string is a **class which defines objects** that be represented as stream of characters.
Size of the character array has to **allocated statically**, more memory cannot be allocated at	In case of strings, memory is **allocated dynamically**. More memory can be allocated

run time if required. Unused allocated **memory is wasted** in case of character array.	at run time on demand. As no memory is preallocated, **no memory is wasted**.
Implementation of **character array is faster** than std:: string.	**Strings are slower** when compared to implementation than character array.
Character array **do not offer** much **inbuilt functions** to manipulate strings.	String class defines **a number of functionalities** which allow manifold operations on strings.

String class is part of C++ library that supports a lot much functionality over C style strings. This class is very large and includes many constructors, member functions and operators.

Operations on strings

1. Creating String Objects

1) string s1; (null string) //using constructor with no argument

2) string s2("xyz"); //using one argument constructor

3) s1=s2; //assigning string objects

4) cin>>s1; //reading through keyboard

5) getline (cin s1)

2. Input Functions

- **getline()** :- This function is used to **store a stream of characters** as entered by the user in the object memory.
- **push_back()** :- This function is used to **input** a character at the **end** of the string.
- **pop_back()** :- Introduced from C++11(for strings), this function is used to **delete the last character** from the string.

3. Other string Manipulation functions

- **copy("char array", len, pos)** :- This function **copies the substring in target character array** mentioned in its arguments. It takes 3 arguments, **target char array, length to be copied and starting position in string to start copying.**
- **swap()** :- This function **swaps** one string with other.
- **size()** -- returns the length of the string
- **length()** -- returns the length of the string (same as size())

- **capacity()** -- returns the current allocated size of the string object (allocation might be larger than current usage, which is the length)
- **resize(X, CH)** -- changes the string's allocated size to X. If X is bigger than the currently stored string, the extra space at the end is filled in with the character CH
- **clear()** -- delete the contents of the string. Reset it to an empty string
- **empty()** -- return true if the string is currently empty, false otherwise
- **at(X)** -- return the character at position X in the string. Similar to using the [] operator
- **Substrings**

  substr(X, Y) -- returns a copy of the substring (i.e. portion of the original string) that starts at index X and is Y characters long

  substr(X) -- returns a copy of the substring, starting at index X of the original string and going to the end
- **Append** -- several versions. All of these append something onto the END of the original string (i.e. the calling object, before the dot-operator)

  append(str2) -- appends str2 (a string or a c-string)
- **Compare**

  str1.compare(str2) -- performs a comparison, like the c-string function strcmp. A negative return means str1 comes first. Positive means str2 comes first. 0 means they are the same

  str1.compare(str2, X, Y) -- compares the portions of the strings that begin at index X and have length Y. Same return value interpretation as above
- **Find**

  str.find(str2, X) -- returns the first position at or beyond position X where the string str2 is found inside of str

  str.find(CH, X) -- returns the first position at or beyond position X where the character CH is found in str
- **Insert**

  str.insert(X, Y, CH) -- inserts the character CH into string str Y times, starting at position X

  str.insert(X, str2) -- inserts str2 (string object or char array) into str at position X

Example:

```
// C++ program to demonstrate various function string class
#include <bits/stdc++.h>
int main()
{
    string str1("first string");                                  // initialization by raw string
    string str2(str1);                                           // initialization by another string
    string str3(5, '#');                 // initialization by character with number of occurence

    // initialization by part of another string
   string str4(str1, 6, 6); //    from 6th index (second parameter)
                        // 6 characters (third parameter)
     // initialization by part of another string : iterators version
   string str5(str2.begin(), str2.begin() + 5);
   cout << str1 << endl;
   cout << str2 << endl;
   cout << str3 << endl;
   cout << str4 << endl;
   cout << str5 << endl;

     string str6 = str4;                          //  assignment operator
      str4.clear();                          // clear function deletes all character from string

     //  both size() and length() return length of string and  //  they work as synonyms
   int len = str6.length(); // Same as "len = str6.size();"
  cout << "Length of string is : " << len << endl;

   // a particular character can be accessed using at /   // [ ] operator
   char ch = str6.at(2); //  Same as "ch = str6[2];"
   cout << "third character of string is : " << ch << endl;

   //  front return first character and back returns last charcter
   //  of string

   char ch_f = str6.front();
   char ch_b = str6.back();

   // append add the argument string at the end
   str6.append(" extension");      //  same as str6 += " extension"
  cout << str6 << endl;
   cout << str4 << endl;

     //  substr(a, b) function returns a substring of b length  //  starting from index a
```

```
    cout << str6.substr(7, 3) << endl;
    //  erase(a, b) deletes b characters at index a
    str6.erase(7, 4);
    cout << str6 << endl;

      str6 = "This is a examples";

    //  replace(a, b, str)  replaces b characters from a index by str
    str6.replace(2, 7, "ese are test");
     cout << str6 << endl;
string s1(“Road”);
string s2(“Read”);
s1.swap(s2);
}
```

ABSTRACT CLASSES

A class which represents generalization and provides functionality which is only intended to be extended but not instantiated is called as Abstract Class.

Abstract classes: Properties

- Objects cannot be created for the abstract classes. If a class has only one method as abstract, then that class must be an abstract class.
- The child class which extends an abstract class must define all the methods of the abstract class.
- If the abstract method is defined as protected in the parent class, the function implementation must be defined as either protected or public, but not private.
- The signatures of the methods must match, optional parameter given in the child class will not be accepted and error will be shown.
- It may contain static variables or methods
- It may contain constructor but it can’t be called directly since abstract class can’t be instantiated.

```
#include <iostream>
// Base class
class Shape
 {
   public:
     // pure virtual function providing interface framework.
     virtual int getArea() = 0;
     void setWidth(int w) {
       width = w;
     }
     void setHeight(int h) {
       height = h;
     }
```

```
   protected:
      int width;
      int height;
};

// Derived classes
class Rectangle: public Shape {
   public:
      int getArea() {
         return (width * height);
      }
};
class Triangle: public Shape {
   public:
      int getArea() {
         return (width * height)/2;
      }
};

void main( ) {
   Rectangle Rect;
   Triangle  Tri;
   Rect.setWidth(5);
   Rect.setHeight(7);
    // Print the area of the object.
   cout << "Total Rectangle area: " << Rect.getArea() << endl;
   Tri.setWidth(5);
   Tri.setHeight(7);

   // Print the area of the object.
   cout << "Total Triangle area: " << Tri.getArea() << endl;
   return 0;
}
```