MASTER CODING IN C++

C++ Programming

Unit 1 : Introduction to Object-Oriented Programming

Structure of the Unit:

1.0 Objective

After studying this unit you will be able to understand :

- The concept of Paradigms and its techniques
- Meaning of object-oriented Programming
- Class, object, its properties and method
- Benefits of OOP and applications of OPPS
- Advantages and disadvantages of OOPS

1.1 Paradigms of Programming Languages

The term paradigm describes a set of techniques, methods, theories and standards that together represent a way of thinking for problem solving. According to [Wegner, 1988], paradigms are "patterns of thought for problem solving ". Language paradigms were associated with classes of languages. First the paradigms are defined. Thereafter, programming languages according to the different paradigms are classified. The language paradigms are divided into two parts:- imperative and declarative paradigms. Imperative languages can be further as classified into procedural and object-oriented approach. Declarative languages can classified into functional languages and logical languages.

Imperative Paradigms:

The meaning of imperative is "expressing a command or order". So, the programming languages in this category specify the step-by-step explaination of command. Imperative programming languages describe the details of how the results are to be obtained in terms of the underlying machine model. The programs specify step-by-step the entire set of transitions that the program goes through. The programs starts from an initial state, goes through the transitions and reaches a final state. Within this paradigm, we have the procedural approach and object-oriented approach.

Procedural Paradigm

Procedural Languages are statement oriented with the variables holding values. In this language, the execution of a program is modeled as a series of states of variable locations. We have two kinds of statements. Non-executable statements and Executable statements.

Non-executable statements allocate memory, bind symbolic names to absolute memory locations and initialize memory.

Executable statements like computation, control flow, and input/output statements. The popular programming languages in this category are Ada, FORTRAN, BASIC, Algol, Pascal, COBOL, Modula, C etc.

Object-Oriented Paradigm

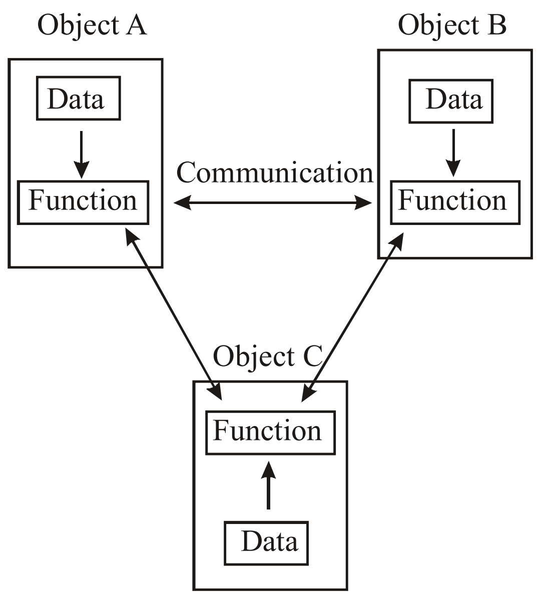
The object-oriented paradigm is centered on the concept of the object. Everything is focused on objects. Can you think what is an object? In this paradigm, program consists of two things: a set of objects and the way they interact with each other. Computation in this paradigm is viewed as the simulation of real world entities. The popular programming languages in this paradigm are : C++, Smalltalk and Java.

Declarative Paradigm

In this paradigm, programs declare or specify what is to be computed without specifying how it is to be achieved. Declarative programming is also known as value-oriented programming. Declarative languages describe the relationship between variables in terms of functions and inference rules. The language executor applies a fixed method to these relations to produce a desired result. It is mainly used in solving artificial intelligence and constraint-satisfaction problems. Declarative paradigm is further divided into two categories: Functional and Logical paradigms.

1.2 What is OOPS?

Object Oriented Programming System (OOPS) is a way of developing software using objects. Objects are the real world models, which are entities in themselves. They contain their own data and behaviour.



Organization of data & functions in OOP

An object resembles the physical world. When something is called as an object in our world, we associate it with a name, properties, etc. It can be called or identified by name and/or properties it bears. When these real world objects are called, they act in some or the other way. Similarly, object in OOPS are called or referenced by the way of messages. Objects have their own internal world (data and function) and external interface to interact with the rest of the program (real world).

Thinking in terms of objects results from the close match between objects in the programming sense and objects in the real world. What kind of things become objects in object-oriented programs? The answer depends on your imagination, but here are some typical categories to start you thinking.

Physical Objects

ATM in Automated Teller Machines

Aircraft in an Air traffic control system

Elements of the Computer user Environment

Windows

Menus

Graphic Objects (lines, rectangles, circles)

Data Storage Constructs

Arrays

Stacks

Linked Lists

Human Entities

Employees

Students

Customers

Let us think about an object employee. The question that we should ask for this object design is: "What are the data items related to an Employee entity? and, What are the operations that are to be performed on this type of data?"

One possible solution for employee type may be:

Object : Employee

Data: Name, DOB, Gender, Basic Salary, HRA, Designation, Department, Contact address, Qualification.

Operations: Find_Age, Compute_Salary, Find_address.

Create_new_employee, Delete_old_employee.

Object : Employee
DATA Name DOB Gender Basic Salary HRA Designation Department Contact Address Qualification ....
Functions Find Age Compute Salary Find Addresses Create- new_employee Delete-old_employee

Representing of object

But now the obvious Question is: How are the objects defined?

The objects are defined via the classes.

Class

Objects with similar properties are put together in a class. A class is a pattern, template, or blueprint for a category of structurally identical items (objects). OOPS programmers view Objects as instances of Class. Infact, objects are variables of the type Class. Once, class has been defined, we can create any number of objects belonging to that class. A class is thus a collection of objects of similar type.

Class contains basic framework, i.e., it describes internal organisation and defines external interface of an object. When we say a class defines basic framework, we mean that it contains necessary functionality for a particular problem domain. For example, suppose we are developing a program for calculator, in which we have a class called calculator, which will contain all the basic functions that exist in a real world calculator, like add, subtract multiply, etc., the calculator class will, thus, define the internal working of calculator and provides an interface through which we can use this class. For using this calculator class, we need to instantiate it, i.e., we will create an object of calculator class. Thus, calculator class will provide a blueprint for building objects. An object which is an instance of a class is an entity in itself with its own data members and data functions. Objects belonging to same set of class shares methods/functions of the class, but they have their own separate data values.

Calculator calc

Calculator calc will create an object calculator calc belonging to the class calculator. Class in OOPS contains its members and controls outside access, i.e., it provides interface for external access.

All the objects share same member data functions, but maintain separate copy of member data.

You can use class for defining a user defined data type. A class serves a plan, or a template that specifies what data and what functions will be included in an objects of that class. Defining the class does not create any objects. A class has meaning only when it is instantiated. For example, we can use a class employee directly.

Inheritance

Let us now consider a situation where two classes are generally similar in nature with just couple of differences. Would you have to re-write the entire class?

Inheritance is an important feature of OOPS that allows derivation of the new objects from the existing ones. It allows the creation of new class, called the derived class, from the existing classes called as base class.

The concept of inheritance allows the features of base class to be accessed by the derived classes, which in turn have their new features in addition to the old base class features. The original base class is also called the parent or super class and the derived class is also called as sub-class. This concept of inheritance provides the idea of receisability. This means that we can add additional features to an existing class without modigying it.

An example

Cars, mopeds, trucks have certain features in common, i.e., they all have wheels, engines, headlights, etc. They can be grouped under one base class called automobiles. Apart from these common features, they have certain distinct features which are not common like mopeds have two wheels and cars have four wheels, also cars use petrol and trucks run on diesel.

The derived class has its own features in addition to the class from which they are derived.

Let us extend our example of employee class in the context of Inheritance. After creating the class, Employee, you might make a sub-class called, Manager, which defines some manager-specific operations on data of the sub-class 'manager'. The feature which can be included may be to keep track of employee being managed by the manager.

We know that inheritance also promotes reuse. You do not have to start from scratch when you write a new program. You can simply reuse an existing repertoire of classes that have behaviour similar to what you need in the new program.

Data Abstraction and Encapsulation

Whenever we have to solve a problem then first we try to distinguish between the important and unimportant aspects of the problem. This is abstraction, thus, Abstraction identifies pattern and frameworks, and separate important and non-important problem spaces. Abstraction refers to the an act of representing essential features without including the background details.

To invent programs, you need to be able to capture the same kinds of abstractions as the problem have, and express them in the program design.

From the implementation point of view, a programmer should be concerned with "what the program is composed of and how does it works?" On the other hand, a user of the program is only concerned with "What it is and what it does."

All programming languages provides a way to express abstractions. In essence, abstraction is a way of grouping implementation details, hiding them, and giving them, at least to some extent, a common interface.

For example, you have a group of functions that can act on a specific data structures. To make those functions easier to use by, you can take the data structure out of the interface of the entity/object by supplying a few additional functions to manage the data. Thus, all the work of manipulating the data structure, viz., allocating data, initialising, output of information, modifying values, keeping it up to date, etc., can be done through the functions. All that the users do is to call the functions and pass the structure to them.

With these changes, the structure has become an opaque token that other programmers never need to look inside. They can concentrate on what the functions do, not how the data is organised. You have taken the first step toward creating an object. Because an object completely encapsulates their data (hide it), users can think of them solely in terms of their behaviour.

The hidden data structure combines all of the functions that share it. So, an object is more than a collection of random functions; it is a grouping, a bundle of related behaviours that are supported by shared data.

This progression from thinking about functions and data structures to thinking about object behaviour is the essence of object-oriented programming. It may seem unfamiliar at first, but as you gain experience with object-oriented programming, you will find that it is a more natural way to think about things. By providing higher level of Abstractions, object-oriented programming language give us a larger vocabulary and a richer model to program in.

The wrapping up of data and function into a single unit is known as encapsulation. Encapsulation is the word which came from the word “CAPSULE” which to put something in a kind of shell. Encapsulation is a way of organizing data and methods into a structure by hiding the way the object is implemented. It prevents the access to data by any means other than those specified. Encapsulation therefore guarantees the integrity of the data contained in the object. Encapsulation defines the access levels for elements of that class. These access levels define the access rights to the data, allowing us to access the data by a method of that particular class itself.

Polymorphism

Polymorphism is another important OOP concept. Polymorphism means the ability to take more than one form. An operation may exhibit different behaviours in different instances. The behaviour depends on the data types used in the operation. Polymorphism is a Greek word. Polymorphism enables us to “program in the general” rathar than “program in the specific.” Polymorphism results from the fact that every class lives in its own name space. The names assigned within a class definition won’t conflict with names assigned anywhere outside it.

1.3 Benefits of OOP

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

- Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map objects in the problem domain to those in the program.

- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more details of a model in implementable form.
- Object-oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

While it is possible to incorporate all these features in an object-oriented system, their importance depends on the type of the project and the preference of the programmer.

Developing a software that is easy to use makes it hard to build. It is hoped that the object-oriented programming tools would help manage this problem.

1.4 Uses of Object-Oriented Programming

OOPS is not only a programming tool, but also a whole modeling paradigm. In addition to general problem solving, two of the upcoming object-oriented applications that are emerging very fast are:

System Software

As an object-oriented operating system, its architecture is organised into frameworks of objects that are hierarchically classified by function and performance. By that, we mean that the whole operating system can be found as made up of objects. The object oriented programming has been a great help for Operating system designers; it allowed them to break the whole operating system into simple and manageable objects.

It allowed them to reuse existing codes by putting similar objects in related classes. KDE (a well known desktop of Linux) developers have extensively used the concepts of object oriented programming.

Linux Kernel itself is a well known application of object oriented programming.

DBMS

Also known as Object Oriented Database Management System (OODBMS), OODBMS store data together with the appropriate methods for accessing it; the fundamental concept of object oriented programming, i.e., encapsulation is implemented in them which allows complex data types to be stored in database. The Complex datatypes are not supported in Relational Data Base Management Systems. Every data type as well as its relations are represented as objects in OODBMS.

OODBMS have the following features:

- Complex data types can be stored.
- A wide range of data types can be stored in the same database (e.g., multimedia applications).
- Easier to follow objects through time, this allows applications which keeps track of objects which evolve in time.

The other promising area for application of OOP includes :

1. Image Processing
2. Pattern Recognition
3. Computer Aided Design and Manufacturing

4. Intelligent Systems
5. Web Based Applications
6. Distrubuted Compuring and Applications
7. Enterprise Resource Planning
8. Data Security and Management
9. Mobile Conputing
10. Parallel Computing

1.5 Advantages of Ojbective-Oriented Programming

The popularity of Object-oriented programming (OOP) was because of its methodology, which allowed breaking complex large software programs to simpler, smaller and manageable components. The costs of building large monolithic software were enormous. Moreover, the fundamental things in object oriented programming are objects which model real world objects. The following are the basic advantages of object-oriented systems:

- **Modular Design:** The software built around OOP are modular, because they are built on objects and we know objects are entity in themselves, whose internal working is hidden from other objects and is decoupled from the rest of the program.
- **Simple approach:** The objects, model real world entities, which results in simple program structure.
- **Modifiable:** Because of its inherent properties of data abstraction and encapsulation, the internal working of objects is hidden from other objects. Thus, any modification made to them should not affect the rest of the system.
- **Extensible:** The extension to the existing program for its adaptation to new environment can be done by simple adding few new objects or by adding new features in old classes/ types.
- **Flexible:** Software built on object-oriented programming can be flexible in adapting to different situations because interaction between objects does not affect the internal working of objects.
- **Reusable:** Objects once made can be reused in more than one program.
- **Maintainable:** Objects are separate entities, which can be maintained separately allowing fixing of bugs or any other change easily..

1.6 Object-Oriented Language

Object-oriented programming is not the right of any particular language. OOP concepts can be implemented using language such as C and Pascal. However, programming becomes clumsy and may generate confusion when the programs grow large. The object-oriented language can be classified into two categories:

- Object-based programming language
- Object-oriented programming language

The object-based programming language is the style of programming that primarily supports encapsulation and object identity without supporting important features of OOP language such as polymorphism, inheritance and message based communications. Ada is one of the typical object-based programming language:

Object-based language = Encapsulation + Object identity

Object-oriented language incorporate all the features of object-based programming languages along with inheritance and polymorphism,

Object-oriented language = Object-based language + inheritance + polymorphism

Language that support these features include C++. Smalltalk, Object Pascal and Java.

1.7 Usage of C++

C++ is a versatile language for handling very large programs. It is suitable for virtually any programming task including development of editors, compilers, database, communication system and any complex real-life application systems.

- Since C++ allows us to create hierarchy-related objects, we can build special objects-oriented libraries which can be used later by many programmers.
- While C++ is able to map the real-world problem properly, the C part of C++ gives the language the ability to get close to the machine-level details.
- C++ programs are easily maintainable and expandable.

1.8 Summary

The most popular phase till recently was procedure-oriented programming. The procedure-oriented programming employs top-down programming approach. Object oriented programming was invented to overcome the drawbacks of the procedure-oriented programming OOPs treats data as a critical element in the program development and does not allow it to flow freely. It ties data more closely to functions that operate on it in a data structure called class. The feature is called data encapsulation. Objects are instances of classes. Incapsulation of data from direct access by the program is call data hiding. Inheritance is the process by which objects of one class acquire properties of object of another class. Polymorphism means one name, multiple forms. It allows us to have more than one function with the same name in a program.

1.9 Self Assessment Questions

1. What is procedure-oriented programming? What are its main charactersistics ?
2. How are data and function organized in an object-oriented program?
3. What kinds of things can become objects in OOP?
4. What are the unique advantages of an object-oriented programming paradigm?
5. Explain various application areas of object-oriented programming.

Unit 2 : Introduction to C++

Structure of the Unit :

2.0 Objective

After studying this unit you will be able to understand :

- About C++ programming including how to create, compile & link program.
- The structure of C++ program and creating source file.
- About C++ streams and buffering.

2.1 Introduction

Like C, C++ began its life at AT&T Bell Labs, New Jersey, USA where Bjarne Stroustrup developed the language in the early 1980s. Its main purpose was to make writing good programs easier and more pleasant for the individual programmer.

A computer simulation language called Simula 67 inspired C++'s OOP aspect. Stroustrup added OOP features to C without significantly changing the C component. Thus, C++ is a superset of C, meaning that any valid C program is valid C++ program, too. There are some minor discrepancies, but nothing crucial. C++ programs can use existing C software libraries. Libraries are collection of programming modules that you can call up from a program. The name C++ comes from the C increment operator ++, which adds 1 to the value of a variable. The name C++ correctly suggests an augmented version of C. While the OOP aspect of C++gives the language the ability to relate concepts involved in the problem, the C part of C++ gives the language the ability to get close to the hardware. The three most important facilities that C++ adds on to C are classes, function overloading and operator overloading. These features enable us to create abstract data types, inherit properties from existing data types and suppor polymorphism, thus making C++ a truly object oriented language.

2.2 A simple C++ Program

Let us begin with a simple example of C++ program.

```
#include<iostream.h >   // include header file
```

```
using namespace std;
int main()
{
cout << "Hello World" << endl;
return (0);
}
```

The following is the output of the above example

```
Hello World
```

Program Feature

C++ follows in the footsteps of C where there is the concept of the kernel of the language and an additional set of library routines. The #include line is an instruction to the compiler to make available to the following program unit what is defined in iostream.h. There is the concept of compiler preprocessing in C and C++ programs. The # indicates a preprocessor directive. The < > characters are used to indicate a standard language header file, in this case iostream.h. I/O is not available in the kernel of the language. It is made available by the inclusion of iostream.h in the complete program.

The next line is the start of the program itself. All programs are called main in C++. There is also the concept of every program unit being a function. Functions in C++ either return a value (and in this case we are defining main to return an integer value) or not. If we do not want a function to return a value we use void rather than a data type in conjunction with the function name to indicate this.

The next thing of interest is the '{' character which indicates the start of the program. The next statement cout (pronounced see out) prints some text to the standard out stream or screen. Text is delimited in C++ with " " marks, endl is predefined in C++ to generate an end of line. The '<<' are C++ operators. They are used to separate items in the output stream. ; is the statement separator or terminator in C++. Finally the program terminates with the return(0) statement. The ')' character signifies the end of the program.

Comments

C++ introduces a new comment symbol // (double slash). Comments start with a double slash symbol and terminate at the end of the line. A comment may start anywhere in the line and whatever follows till the end of the line is ignored. We can say comments are non-executable statements ignored by compiler at the time of compilation.

The double slash comment is basically a single line comment. Multiline comments can be written as follows:

```
// This is an example of
// C++ program to illustrate
//some of its features
```

The C comment symbols /*, */ are still valid and are more suitable for multiline comments. The following comment is allowed

```
/* This is an example of  C++ program to illustrate
some of its features */
```

2.3 Structure of C++ Program

The typical C++ program would contain for section as shown in fig. 2.1. These sections may be placed in separate code files and then compiled independently or jointly.

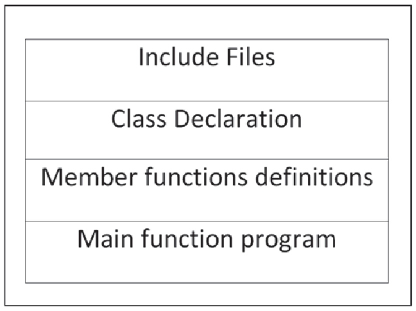


Fig 2.1 Structure of a C++ program

It is a common practice to organize a program into three separate files. The class declarations are placed in a header file and the definitions of member functions go into another file. This approach enables the programmer to separate the abstract specification of the interface (class definition) from the implementation details (member functions definition). Finally, the main program that uses the class is placed in a third file which "includes" the previous two files as well as any other files required.

This approach is based on the concept of client-server model. The class definition including the member functions constitute the server that provides services to the main program known as client. The client uses the server through the public interface of the class.

Basic structure of a C++ program

```
#include <iostream.h>
int main()
{
declaration1;
declaration 2;
.
.
execution statement 1;
execution statement 2;
.
.
return (0);
    }
```

2.4 Creating the Source File

Like C programs, C++ programs can be created using any text editor. For example, on the UNIX, we can use vi or ed text editor for creating and editing the source code. On the DOS system, we can use edlin or any other editor available or a word processor system under non-document mode.

Some systems such as Turbo C++ provide an integrated environment for developing and editing programs. Appropriate manuals should be consulted for complete details.

The file name should have a proper file extension to indicate that it is a C++ program file. C++

implementations use extensions such as .c, .C, .cc, .cpp and .cxx. Turbo C++ and Borland C++ use .c for C programs and .cpp (C plus plus) for C++ programs. Zortech C++ system uses .cxx while UNIX AT&T version uses .C (capital C) and .cc. The operating system manuals should be consulted to determine the proper file name extensions to be used.

2.5 Compiling and Linking

The process of compiling and linking again depends upon the operating system. A few popular systems are discussed in this section.

Unix AT&T C++

The process of implementation of a C++ program under UNIX is similar to that of a C program.We should use the "CC" (uppercase) command to compile the program. Remember, we use lowercase "cc" for compiling C programs. The command at the UNIX prompt would compile the C++ program source code contained in the file example.C. The compiler would produce an object file example.o and then automatically link with the library functions to produce an executable file. The default executable filename is a.out.

A program spread over multiple files can be complied as follows:

```
CC file1. C file2.o
```

The statement compiles only the file filel.C and links it with the previously compiled file2.o file.This is useful when only one of the files needs to be modified. The files that are not modified need not be compiled again.

Turbo C++ and Borland C++

Turbo C++ and Borland C++ provide an integrated program development environment under MS DOS. They provide a built-in editor and a menu bar which includes options such as File, Edit, Compile and Run.

We can create and save the source files under the File option, and edit them under the Edit option. We can then compile the program under the Compile option and execute it under the Run option. The Run option can be used without compiling the source code. In this case, the RUN command causes the system to compile, link and run the program in one step.

Visual C++

It is a Microsoft application development system for C++ that runs under Windows. Visual C++ is a visual programming environment in which basic program components can be selected through menu choices, buttons, icons, and other predetermined methods.

2.6 C++ Streams

The I/O system in C++ is designed to work with a wide variety of devices including terminals, disks, and tape drives. Although each device is very different, the I/O system supplies an interface to the programmer that is independent of the actual device being accessed. This interface is known as stream.

A stream is a sequence of bytes. It acts either as a source from which the input data can be obtained or as a destination to which the output data can be sent. The source stream that provides data to the program is called the input stream and the destination stream that receives output from the program is called the output stream. In other words, a program extracts the bytes from an input stream and inserts bytes into an output stream as illustrated in Fig. 2.2.

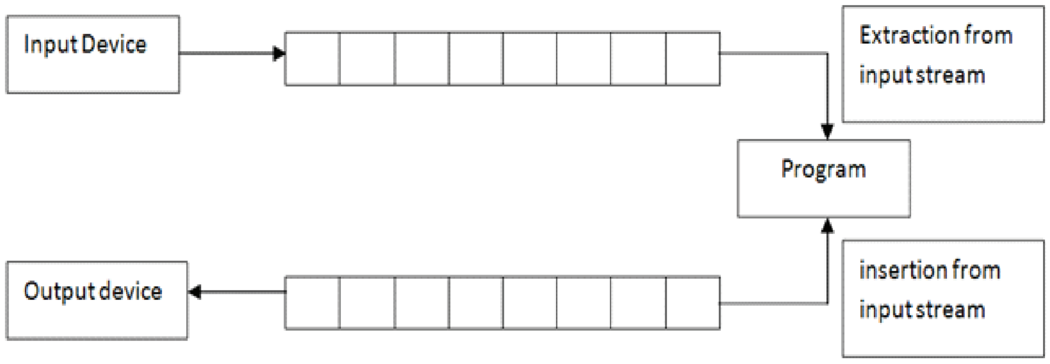


Fig 2.2 Data Streams

The data in the input stream can come from the keyboard or any other storage device. Similarly, the data in the output stream can go to the screen or any other storage device. As mentioned earlier, a stream acts as an interface between the program and the input/output device. Therefore a C++ program handles data (input or output) independent of the devices used.

C++ contains several pre-defined streams that are automatically opened when a program begins its execution. These include cin and cout which have been used very often in our earlier programs. We know that cin represents the input stream connected to the standard input device (usually the keyboard) and cout represents the output stream connected to the standard output device (usually the screen). Note that the keyboard and the screen are default options. We can redirect streams to other devices or files, if necessary.

2.7 C++ Stream Classes

The C++ I/O system contains a hierarchy of classes that are used to define various streams to deal with both the console and disk files. These classes are called stream classes. Figure 2.3 shows the hierarchy of the stream classes used for input and output operations with the console unit. These classes are declared in the header file iostream. This file should be included in all the programs that communicate with the console unit.

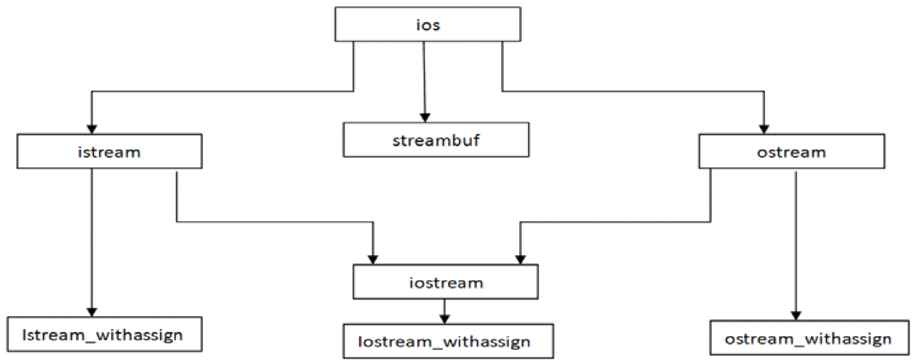


Fig 2.3 Stream classes for console I/O operations

As seen in the Fig. 2.3, ios is the base class for istream (input stream) and ostream (output stream) which are, in turn, base classes for iostream (input/output stream). The class ios is declared as the virtual base class so that only one copy of its members are inherited by the iostream.

The class ios provides the basic support for formatted and unformatted I/O operations. The class istream provides the facilities for formatted and unformatted input while the class ostream (through inheritance) provides the facilities for formatted output. The class iostream provides the facilities for handling both input and output streams. Three classes , namely, istream_withassign, ostream_withassign, and iostream_withassign add assignment operators to these classes.

The >> operator is overloaded in the istream class and << is overloaded in the ostream class. The following is the general format for reading data from the keyboard:

cin>> variable1 >> variable2 >> …………………>> variableN

Variable1, variable2,... are valid C++ variable names that have been declared already. This statement will cause the computer to stop the execution and look for input data from the keyboard. The input data for this statement would be:

data1 data2 …….. dataN

The input data are separated by white spaces and should match the type of variable in the cin list. Spaces, newlines and tabs will be skipped.

The operator >> reads the data character by character and assigns it to the indicated location. The reading for a variable will be terminated at the encounter of a white space or a character that does not match the destination type. For example, consider the following code:

```
int code;
        cin >> code;
```

Suppose the following data in given as input:

4258D

The operator will read the characters upto 8 and the value 4258 is assigned to code. The character D remains in the input stream and will be input to the next cin statement. The general form for displaying data on the screen is:

cout << item1<< item2 << .... << itemN

The items item1 through itemN may be variables or constants of any basic type. We have used such statements in a number of examples illustrated in previous units.

2.8 Summary

C++ is a superset of C language. C++ adds a number of object oriented features such as objects, inheritance, function overloading and operator overloading. C++ program begins at main(). The header file **iostream** should be included at the beginning of all programs.

2.9 Self Assessment Questions

1. Write a program to display the following output using a single cout statement

   Maths = 90 ;

   Physics = 77 ;

   Chemestry = 69

2. Write a program to read two numbers from the keyboard and display the large value on the screen.

3. Write a C++ program that will ask for a temperature in Fahrenheit and display in Celsiums.

4. How does a main () function in C++ differ from main( ) in C?

Unit 3 : Tokens, Expression and Control Structures

Structure of the Unit:

3.0 Objective

After studying this unit you will be able :

- To understand the token and keywords in C++
- To understand identifiers, constant and declaration of variables
- To understand basic data types and user defined data type of C++
- To understand operators and their precedence in C++
- To understand the control structures

3.1 Introduction

As we know C++ is a superset of C and therefore most constructs of C are legal in C++. But there are some exceptions and additions. In this unit, we shall discuss these exceptions and additions in C++.

3.2 Tokens

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

- Keywords
- Identifiers
- Constants
- Strings
- Operators

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

3.3 Keywords

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

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

3.4 Identifiers and Constants

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

* Only alphabetic characters, digits and underscores are permitted.

* The name cannot start with a digit.

* Uppercase and lowercase letters are distinct.

* A declared keyword cannot be used as a variable name.

asm	double	new	switch
auto	else	operator	template
break	enum	private	this
case	extern	protected	throw
catch	float	public	try
char	for	register	typedef

class	friend	return	union
const	goto	short	unsigned
continue	if	signed	virtual
default	inline	sizeof	void
delete	int	static	volatile
do	long	struct	while
Added by ANSI C++			
bool	export	reinterpret_cast	typename
const_cast	false	static_cast	using
dynamic_cast	mutable	true	wchar_t
explicit	namespace	typeid	

Table 3.1 C++ keywords

A major difference between C and C++ is the limit on the length of a name. While ANSI C recognizes only the first 32 characters in a name, ANSI C++ places no limit on its length and, therefore, all the characters in a name are significant.

Care should be exercised while naming a variable which is being shared by more than one file containing C and C++ programs. Some operating systems impose a restriction on the length of such a variable name.

Constants refer to fixed values that do not change during the execution of a program.

Like C, C++ supports several kinds of literal constants. They include integers, characters, floating point numbers and strings. Literal constant do not have memory locations. Examples:

```
123             //decimal integer
12.34           // floating point integer
037             // octal integer
0X2             // hexadecimal integer
"C++"           // string constant
'A'             // character constant
L 'ab'          // wide-character constant
```

The wchar_t type is a wide-character literal introduced by ANSI C++ and is intended for character sets that cannot fit a character into a single byte. Wide-character literals begin with the letter L.

C++ also recognizes all the backslash character constants available in C.

3.5 Basic Data Types

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

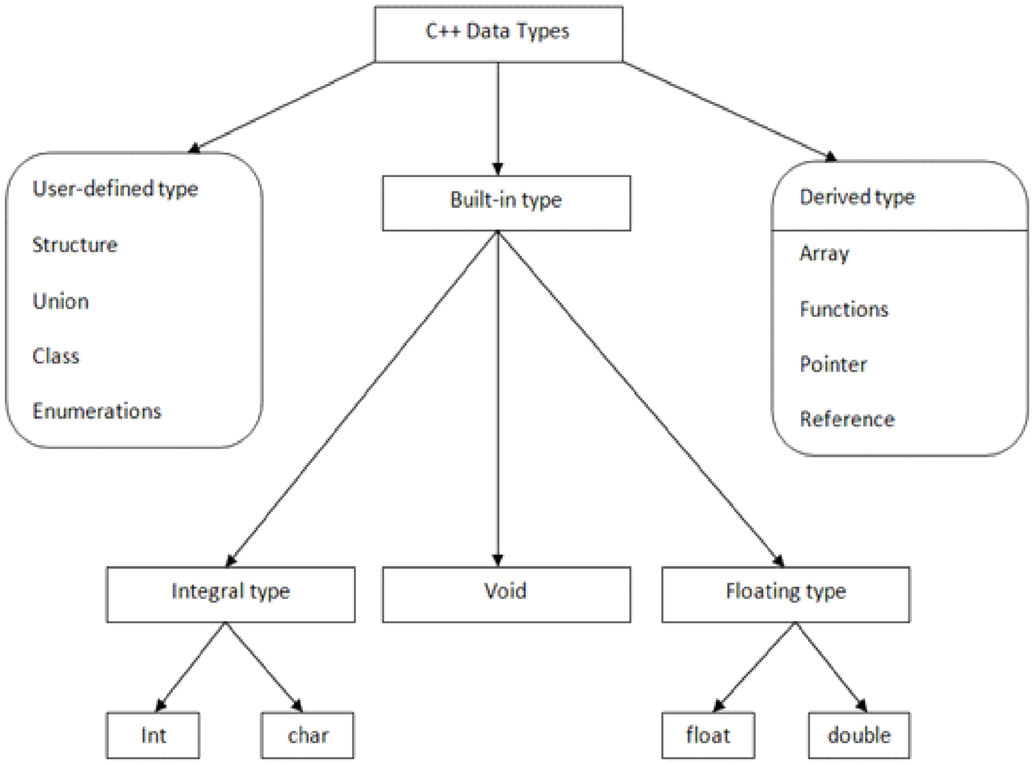


Fig 3.5 Hierarchy of C++ data types

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

Type	**Bytes**	**Range**
char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	- 128 to 127
int	2	- 32768 to 32767
nsigned int	2	0 to 65535
signed int	2	- 31768 to 32767
short int	2	- 31768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	-32768 to 32767
long int	4	-2147483648 to 2147483647
signed long int	4	-2147483648 to 2147483647

unsigned long int	4	0 to 4294967295
float	4	3.4 E-38 to 3.4E+38
double	8	1. 7E-308 to 1. 7E+308
long double	10	3.4E-4932 to 1.1E+4932

Table 3.2 Size and range of C++ Basic Data Types

ANSI C++ committee has added two more data types, bool and wchar_t. The type void was introduced in ANSI C. Two normal uses of void are (1) to specify the return type of a function when it is not returning any value, and (2) to indicate an empty argument list to a function. Example:

```
void funct1(void);
```

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

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

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

```
int *ip;                // int pointer
p  =  ip;               // assign int printer to void pointer
```

are valid statements. But, the statement

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

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

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

```
void  *ptr1;
char  *ptr2;
ptr2  = ptr1;
```

are all valid statements in ANSI C but not in C++. A void pointer cannot be directly assigned to other type pointers in C++.

3.6 User-Defined Data Types

Structures and Classes

Like C, C++ also support user defined data type such as struct and union. Some more features have been added to make them suitable for object oriented programming. C++ also permits us to define another user-defined data type known as class which can be used, just like any other basic data type, to declare variables. The class variables are known as objects, which are the central focus of object-oriented programming.

Enumerated Data Type

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

enum shape{circle, square, triangle};

enum colour{red, blue, green, yellow};

enum position{off, on};

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

shape ellipse;
// ellipse is of type shape

colour background; // background is of type colour

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

```
colour background = blue;           //  allowed

colour background = 7;              // Error in C++

colour background= (colour) 7;      // OK
```

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

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

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

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

enum colour{red=5, blue, green};

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

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

enum{off, on};

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

int switch_1 = off;

int switch_2 = on;

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

Example:

Enum shape

{

circle,

```
rectangle,
triangle
} ;
int main()
{
cout << "Enter shape code:";
int code;
cin>> code;
while(code >= circle && code <= triangle)
{
switch(code)
{
case circle:
……………..
……………..
break;
case rectangle:
……………..
……………..
break;
case triangle:
……………..
……………..
}
cout " "Enter shape code:";
cin " code;
}
cout " "BYE \n";
return 0;
}
```

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

3.7 Derived Data Types

Arrays

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

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

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

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

Functions

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

Pointers

Pointers are declared and initialized as- in C. Examples:

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

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

```
char * const prt1 = "GOOD";      // constant pointer
```

We can modify the address the ptr1 is initialized to

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

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

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

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

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

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

3.8 Symbolic Constants

The symbolic constants can be created in two ways:

- Using the qualifier const, and
- Defining a set of integer constants using enum keyword.

In both C and C++, any value declared as const cannot be modified by the program in any way. However, there are some differences in implementation. In C++, we can use const in a constant expression, such as

```
const int size = 10;

char name[size];
```

This would be illegal in C, const allows us to create typed constants instead of having to use #define to create constants that have no type information.

As with long and short, if we use the const modifier alone, it defaults to in. For example,

```
const size = 10;
```

C++ requires a const to be initialized. ANSI C does not require an initializer; if none is given, it initializes the const to 0. The scoping of const values differs. A const in C++ defaults to the internal linkage and therefore it is local to the file where it is declared. In ANSI C, const values are global in nature. They are visible outside the file in which they are declared. However, they can be made local by declaring them as static. To give a const value an extemallinkage so that it can be referenced from another file, we must explicitly define it as an extern in C++. Example:

This defines X, Y and Z as integer constants with values 0, 1, and 2 respectively. This is equivalent to:

```
const X=0;

const Y=1;

const Z=2;
```

Such values can be any integer values.

3.9 Type Compatibility

C++ is very strict with regard to type compatibility as compared to C. For instance, C++ defines int, short int, and long int as three different types. They must be cast when their values are assigned to one another. Similarly, unsigned char, char, and signed char are considered as different types, although each of these has a size of one byte. In C++, the types of values must be the same for complete compatibility, or else, a cast must be applied. These restrictions in C++ are necessary in order to support function overloading where two functions with the same name are distinguished using the type of function arguments.

Another notable difference is the way char constants are stored. In C, they are stored as int, and therefore,

```
sizeof('X');
```

is eqivalent to

```
sizeof(int);
```

in C.

In C++, however, char is not promoted to the size of int and therefore

```
sizeof('x');
```

equals

```
sizeof(char);
```

3.10 Declaration of Variables

We know that, in C, all variables must be declared before they are used in executable. statements. This is true with C++ as well. However, there is a significant difference between C and C++ with regard to the place of their declaration in the program. C requires all the variables to be defined at the beginning of a scope. When we read a C program, we usually come across a group of variable declarations at the beginning of each scope level. Their actual use appears elsewhere in the scope, sometimes far away from the place of declaration. Before using a variable, we should go back to the beginning of the program to see whether it has been declared and, if so, of what type.

C++ allows the declaration of a variable anywhere in the scope. This means that a variable can be declared right at the place of its first use. This makes the program much easier to write and reduces the errors that may be caused by having to scan back and forth. It also makes the program easier to understand because the variables are declared in the context of their use.

```
int main ( )
{
float x;
float sum = 0;
for (int i=l; i<5; i++)
{
cin ">>x;
sum = sum +x;
}
float average;
average = sum/(i-l);
cout <<average;
return 0;
}
```

The only disadvantage of this style of declaration is that we cannot see all the variables used in a scope at a glance.

3.11 Dynamic Initialization of Variables

In C, a variable must be initialized using a constant expression, and the C compiler would fix the initialization code at the time of compilation. C++, however, permits initialization ofthe variables at run time. This is referred to as dynamic initialization. In C++, a variable can be initialized at run time using expressions at the place of declaration. For example, the following are valid initialization statements:

```
..................
..................
int n = strlen(string);
..................
```

float area = 3.14159 * rad * rad;

Thus, both the declaration and the initialization of a variable can be done simultaneously at the place where the variable is used for the first time. The following two statements in the example of the previous section

```
float average;                    // declare where it is necessary
average = sum/i;
```

can be combined into a single statement;

```
float average = sum/i;            // initialize dynamically at run time
```

Dynamic initialization is extensively used in object-oriented programming. We can create exactly the type of object needed, using information that is known only at the run time.

3.12 Reference Variables

C++ introduces a new kind of variable known as the reference variable. A reference variable provides an alias (alternative name) for a previously defined variable. For example, if we make the variable sum a reference to the variable total, then sum and total can be used interchangeably to represent that variable. A reference variable is created as follows:

```
data-type & reference-name=variable-name
float total =100;
float &sum = total
```

total is a float type variable that has already been declared; sum is the alternative name declared to represent the variable total. Both the variables refer to the same data object in the memory. Now, the statements

```
cout<< total;
```

and

```
cout<< sum
```

both print the value 100. The statement

```
total = total + 10;
```

will change the value of both total and sum to 110. Likewise, the assignemnt

```
sum = 0;
```

will change the value of both the variables to zero.

A reference variable must be initialized at the time of declaration. This establishes the correspondence between the reference and the data object which it names. It is important to note that the initialization of a reference variable is completely different from assignment to it.

C++ assigns additional meaning to the symbol &. Here, & is not an address operator. The notation float & means reference to float. Other examples are:

```
int n[10];
int & x = n[10];                  // x is alias for n[10]
char & a = '\n';                  // initialize reference to a literal
```

The variable x is an alternative to the array element n[10]. The variable a is initialized to the newline constant. This creates a reference to the otherwise unknown location where the newline constant \n is stored.

```
i.  int  x;
int  *p  = &x;
int  & m =  *p;
ii. int  &n=50;
```

The first set of declarations causes m to refer to x which is pointed to by the pointer p and the statement in (ii) creates an int object with value 50 and name n.

A major application of reference variables is in passing arguments to functions. Consider the following:

```
void f(int & x)
    {
    x=x+10;
    }
    int main ( )
    {
    int m = l0;
     f(m) ;
    _  _ _
    }
```

When the function call f(m) is executed, the following intialization occurs:

int & x= m;

Thus x becomes an alias of m after executing the statement

    f(m);

The Such function calls are known as call by reference. This implementation is illustrated in Fig. 3.2. Since the variables x and m are aliases, when the function increments x, m is also incremented. The value ofm becomes 20 after the function is executed. In traditional C, we accomplish this operation using pointers and dereferencing techniques.

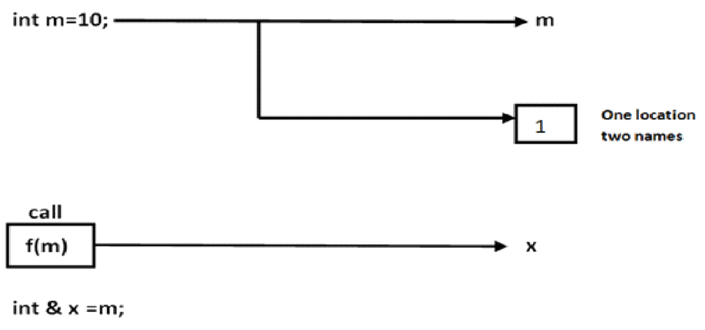


Fig 3.2 Call by refernce mechanism

The call by reference mechanism is useful in object-oriented programming because it permits the manipulation of objects by reference, and eliminates the copying of object parameters back and forth. It is also important to note that references can be created not only for builttin data types but also for user-defined data types such as structures and classes. References work wonderfully well with these user-defined data types.

3.13 Operators in C++

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

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

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

Unary Operators

These are the unary +(plus) sign and the unary – (minus) sign. They are used to indicate the sign of an integer, a floating point quantity, or the exponent of the power of 10 in a floating point quantity. The auto increment (++) and auto decrement (--) operators provide a convenient way of, respectively, adding and subtracting 1 from a numeric variable.

Example The fragment

```
int K = 5;
++K + 10;              //gives 16
K++ +10;               //gives 15
–K+10;                 //gives 14
K– +10;                //gives 15
```

Both operators can be used in prefix and postfix form. The difference is significant. When used in prefix form, the operator is first applied and the outcome is then used in the expression. When used in the postfix form, the expression is evaluated first and then the operator applied. Both operators may be applied to integer as well as real variables.

Arithmetical Binary Operators

The usual binary operations of arithmetic may be performed on integers and floating point numbers: addition, subtraction, multiplication and division have the symbols + – * /, and have algebraic precedence rules. Integral division, however, truncates the decimal part of the quotient. For example, 16/3 yields 5 and 20/3 yields 6. If at least one of the operands is negative, the result is compiler-dependent. Some compilers choose a/b ==$\lceil a / b \rceil$ whilst others choose a/b ==$\lceil a / b \rceil$. For example, on some compilers 16/–3 == –6 and on others, 16/–3 == –5. If we wish to obtain a decimal value for 16/3 then we must cast at least one of the numbers into a double.

```
16.0/3                      //or else 16/3.0

// or

else 16.0/3.0               //or else double (16)/3


//or

else 16/double (3)          //or else double (16)/double (3)
```

Example :

An integer variable initialized as a float, will truncate the decimal part, losing this information forever. The fragement

```
int a = 3.14;

cout << ″a =″ <<a << ‘\n’;

float b;

b = a –0.14;

cout << ″b= ″<<b;
```

will print

```
a=3 b = 2.86
```

There is a further operand on integers: %(read ″mod″or ″modulo″). Here a % b gives the remainder of a upon division by b. Again, if at least one of the operands is negative, the result is compiler dependent, but we always have(a/b) *b+a%b is identical to a. The % operator has the same level of precedence as multiplication or division, and it is left-associative.

Assignment Operators

The assignment operator = is used to indicate that what is to the left of it acquires the value to the right of it. Example The fragment

```
int a = 4;

cout << ″a = ″<< a << ″.″;

a= a+19;

cout <<″\na=″<<a <<″now.″;
```

will print

```
a= 4. a = 23 now.
```

Note : The assignment operator = does not test for equality.

An **lvalue** (left-value) is a storage area (memory location) bound to a variable during the programme execution. An **rvalue**(right-value) is the encoded value stored in the location associated with the variable.

Relational Operators

The relational operators <, < =, > =, ==, != evaluate to 1 if the relation is true and to 0 if the relation is false. The operator! = is read ″not equal to.″

Example: In the fragment

int x = 1, y = 4, z, w, a;

z = x < y;

w= (x + 5) < = y;

a = y! = 5;

z becomes 1 since x < y is true. w becomes 0 since (x + 5) < = y is false. Finally, a becomes 1 since y! = 5 is true.

Logical Operators

The logical negation operator ! gives output according to the following rules:

!1 gives 0 !0 gives 1

The logical AND operator && gives output according to the following rules:

1 && 1 gives 1 1 && 0 gives 00 && 1 gives 00 && 0 gives 0

Notice the similarity between this operator and regular multiplication by 0.

The logical OR operator || gives output according to the following rules:

1 || 1 gives 1 1 || 0 gives 1 0 || 1 gives 1 0 || 0 gives 0

Notice the similarity between this operator and regular addition to 0.

A value different from 0 will be interpreted as 1 by these operators.

Example The fragment of code

int a = -1, b = 3, x, y, z, w, r, s;

x = a & & b; y = (s + 1) || b; z = (a + 1) && (b -3); w = (a!= b) && (a < = 0); r = (a < b) || (y = =0); s =!b;

will produce x == 1, y ==1, z ==0, w==1, r ==1, and s ==0.

Conditional Operator

C++'s only ternary operator is the conditional operator ?:. which has the following syntax:

test ? alternative_1 : alternative_2

and evaluates as follows. Test is evaluated first. If it is true, then the whole expression becomes alternative_1, and alternative_2 is not evaluated at all. If test is false, alternative _1 is skipped, and the whole expression becomes alternative_2.

Example in the fragment

int a = 3, b = 6, c;

c = (a == b)?(a + 1):(b - 8);

c becomes -2, since a = = b

is false and so the conditional expression assumes the second alternative b-8.

Comma Operator

The comma, operator takes its arguments and evaluates them from left to right and returns the value of the rightmost expression.

Example in the fragment

int a = 1, b;

b = (a += 1, a += 2, a + 5);

The comma operator first evaluates a += 1 which makes a == 2. The next expression a += 2 is evaluated and so a == 4. Finally, the last expression a + 5 is evaluated becoming a + 5 == 9. Hence b == 9.

Bitwise Operators

C++ provides six bitwise operators for manipulating the individual bits in an integer quantity. These are:

Operator	Name	Example
~	Bitwise Negation	~'\011' //gives'\366'
&	Bitwise And	'\011' & '\027' //gives'\001'
|	Bitwise Or	'\011' | '\027' //gives'\037'
^	Bitwise Exclusive Or	'\011' ^ '\027' //gives'\036'
<<	Bitwise Left Shift	~'\011' << 2 //gives'\044'
>>	Bitwise Right Shift	~'\011' >> 2 //gives'\002'

3.14 Manipulators

Manipulators are operators that are used to format the data display. The most commonly used manipulators are endl and setw. The endl manipulator, when used in an output statement, causes a linefeed to be inserted. It has the same effect as using the new line character" \ n".

For example, the statement

```
.........................

.........................

cout « "m = "<< m << endl

« "n = "<< n << endl

« "p = “<< p << endl;
```

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

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

m =

n =

p =

It is important to note that this from is not the ideal ouput, It should rather appear as under:

m = 2597

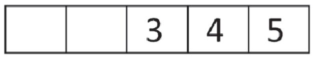
n = 14

p = 175

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

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

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



Example program to illustrate the use of manipulators

```
#include <iostream.h>
#include <iomanip.h>                    // for setw
int main( )
{
cout «setw(10) « "Basic" « setw(10) « Basic « endl
« setw(10) « "Allowance" « setw(10) « Allowance « endl
« setw(10) « "Total" « setw(10) « Total « endl;
return 0;
}
```

The following is the output of above program:

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

We can also write our own manipulators as follows:

```
#include <iostream.h>
ostream & symbol (ostream & output)
        {
return output << "\t Rs";
        }
```

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

3.15 Type Cast Operator

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

Traditional C casts are augmented in C++ by a function-call notation as a syntactic alternative. The following two versions are equivalent:

```
(type-name) expressi on        // C notation
type-name (expression)         // C++ notation average
```

examples :

```
average = sum/(float)i;        // C notation
average = sum/float(i);        // C++ notation
```

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

p = int*(q);

is illegal.

in such cases we must use c type notation

p= (int *)q;

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

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

ANSI C++ adds the following new cast operators:

- const_cast
- static_cast
- dynamic __ cast
- reinterpret_cast

3.16 Expressions and Their Types

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

- Constant expressions
- Integral expressions
- Float expressions
- Pointer expressions
- Relational expressions
- Logical expressions
- Bitwise expressions

An expression may also use combinations of the above expressions. Such expressions are known as compound expressions.

Constant Expressions

Constant Expressions consist of only constant values.

Examples:

15

20 + 5 / 2.0

‘X’

Integral Expressions

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

Examples:

m

m * n - 5

m * 'x'

Float Expressions

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

x + y

x * y / 10

5 + float(10)

10.75

Pointer Expressions

Pointer Expressions produce address values. Examples:

&m

ptr

ptr + 1

"xyz"

Relational Expressions

Relational Expressions yield results oftype bool which takes a value true or false. Examples:

x <= y

a + b == c + d

m + n > 100

When arithmetic expressions are used on either side of a relational operator, they will be evaluated first and then the results compared. Relational expressions are also known as Boolean expressions.

Logical Expressions

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

a > b && x == 10

x == 10 || y == 5

Bitwise Expressions

Bitwise Expressions are used to manipulate data at bit level. They are basically used for testing or shifting bits. Examples:

x << 3	//Shift three bit position to left
y << 1	// Shift one bit position to right

ANSI C++ has introduced what are termed as operator kevwords that can be used as alternative representation for operator symbols.

3.17 Operator Precedence

Although C++ enables us to add multiple meanings to the operators, yet their association and precedence remain the same. For example, the multiplication operator will continue having higher precedence than the add operator. Table 3.3 gives the precedence and associativity of all the C++ operators. The groups are listed in the order of decreasing precedence. The labels prefix and postfix distinguish the uses of ++ and --. Also, the symbols +. -, *, and & are used as both unary and binary operators.

Operator	**Associativity**
: :	left to right
–>.( ) [ ] postfix ++ postfix - -	left to right
prefix ++ prefix - - ~! unary + unary -	
unary * unary & (type) size of new delete	right to left
–> **	
	left to right
* / %	left to right
+ -	left to right
<< >>	left to right
<< = >> =	left to right
= = ! =	left to right
&	left to right
^	left to right

|	left to right
&&	left to right
| |	left to right
?;	left to right
= * = / = % = + = =	right to left
<< = >> = & = ^ = | =	left to right
, (comma)	

Table 3.3 Operator precedence and associativity

3.18 Control Structures

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

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

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

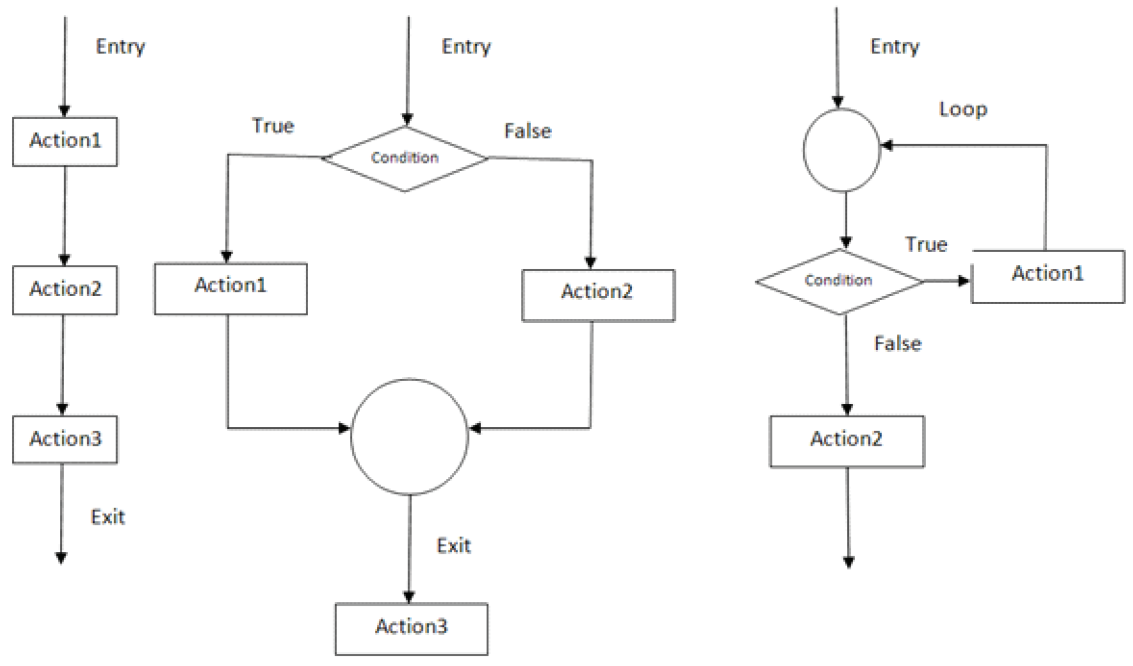


Fig. 3.4 Basic control structures

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

Using these three basic constructs, we may represent a function structure either in detail or in summary form as shown in Fig. 3.4.

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

The if statement

the if statement is implemented in two forms:

- Simple if statement
- if ... else statement

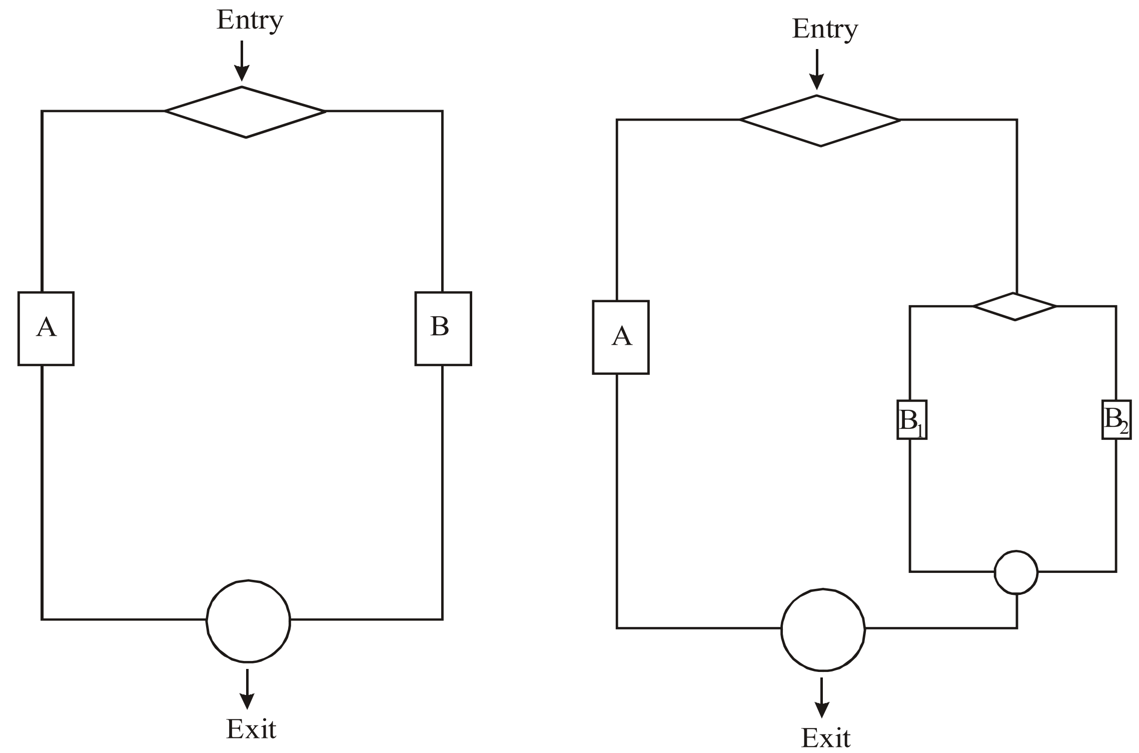


Fig. 3.5 Different levels of abstraction

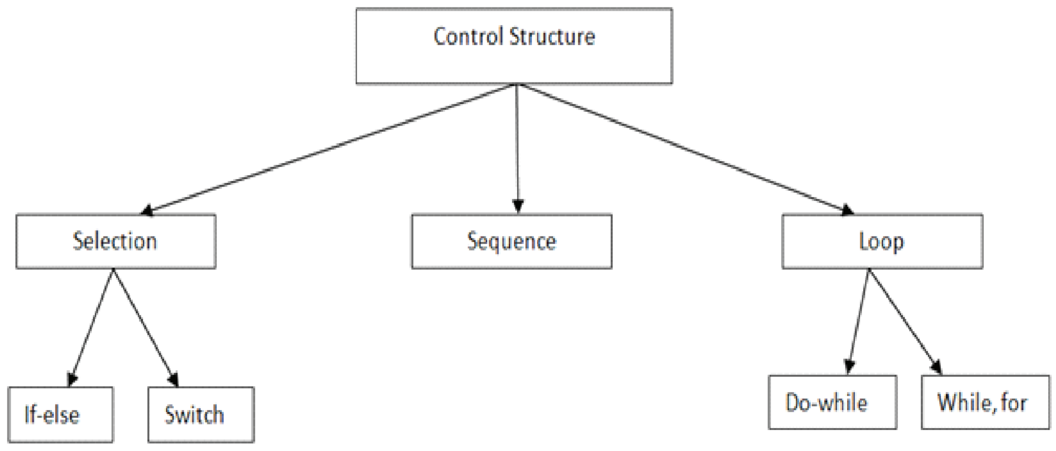


Fig. 3.6 C++ statements to implement in two forms

Examples:

Form 1

```
    if(expression is true)
{
     actionl;
}
    action2;
    action3;
    Form 2
    if(expression is true)
{
      actionl;
}
    else
{
      action2;
}
    action3;
```

The switch statement

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

```
switch(expression)
{
    casel:
{
    actionl;
}
    case2:
{
    action2;
}
```

```
    case3:
{
    action3;
}
    default:
{
    action4;
}
}
    action5;
```

The do-while statement

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

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

The while statement

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

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

The for statement

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

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

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

3.19 Summary

C++ provides various types of token-keywords, identifiers, constants and operators. Identifiers are name of variable, function, arrays, and classes. In C++, the size of character array should be one larger than the number of characters in the string, C++ provides const to declare named constant. C++ supports seven types of expression. C++ performs conversion automatically using rules. C++ also permits explicit type conversion of variables and expressions using the type cast operators. C++ also supports the three basic control structure namely, sequence, selection and loop and implements them, using various control statements such as if, if....else, switch, do...while, while and for.

3.20 Self Assessment Questions

1. Describe the differences in the implementation of enum data type in ANSI C and C++.
2. In C++, a variable can be declared anware in the scope. What is significance of this feature?
3. What is a reference variable? What is its major use?
4. Write a program for prime number.
5. Write a program to read any ten nuumbers and find sum and average.

Unit 4 : C++ Functions

Structure of the Unit:

4.0 Objective

After the end of this unit you will be able to understand

- about, function prototype and its definition
- about function call by value, call by reference
- about default argument and const. argument in function

4.1 Introduction

Functions can play an important role in C++. Dividing a program into functions is one of the major principles of top-down programming. Using functions, the size of program can be reduce.

Function is a self-contained program segment that carries out some specific well-defined task. Every C++ program consists of one or more functions. The most important function is main (). The definitions of functions may appear in any order in a program file because they are independent of one another. The general form of the function is :

```
    return_type function_name (parameter list)
{
    body of the function
}
```

The function of consists of two parts function header and function body. The function header is :

return_type function_name(parameter list)

The body of the function performs the computations.

Example program to illustrate the use of function

```
          #include<iostream.h>
          int factorial(int n);
int main ()
          {
          int n1, fact;
          cout <<"Enter the number whose factorial has to be calculated: "<< endl;
          cin >> n1;
          fact= factorial (n1);
          cout << "The factorial of"<<n1<< " is : " << fact << endl;
          return(0);
          }
          int factorial (int n)
          {
          int i=0, fact=1;
          if(n < =1)
          {
          return(1);
          }
          else
          {
          for (i=1; i< =n;i++)
          {
               fact = fact*i;
          }
          return (fact);
          }
          }
```

The following is the output result of the program is:

```
Enter the number whose factorial has to be calculated :5
The factorial of :5 is : 120
```

When the function is called, control is transferred to the first statement in the function body. After the execution of all the statments of function body control returns to the main program. A C++ function can be overloaded.

4.2 The main Function

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

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

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

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

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

The functions that have a return value should use the return statement for termination.

The main() function in C++ is, therefore, defined as follows:

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

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

Function should return a value

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

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

4.3 Function Prototyping

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

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

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

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

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

Example:

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

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

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

is illegal.

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

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

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

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

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

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

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

```
Float cube1= volumn(b1,w1mh1);                    //Function call
```

The variable b1, w1, and h1 are known as the actual parameters which specify dimensions of cube1. Their types (which have been declared earlier) should match with types declared in the prototype. Remember, the calling statement should not include type names in the argument list. We can also declare a function with an empty argument list, as in the following example:

```
void display();
```

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

void display(void);

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

Void do_something(…);

4.4 Call by Value

In Call by value mechanism copies of the arguments are created and which are stored in the temporary locations of the memory. The arguments are mapped to the copies of the arguments created. The changes made to the parameter do not affect the arguments. Call by value mechanism provides security to the calling program. Here is a program which illustrates the working of call by value mechanism.

Example program to illustrate the Call by value function

```
#include<iostream.h>
int add(int n);
int main()
{
int number, result;
number= 5;
cout << "The initial value of number: "<< number << endl;
result = add(number);
cout << "The final value of number : " << number << endl;
cout << " The result is : "<< result << endl;
return(0);
}
int add(int number)
{
number= number+100;
return (number);
}
```

The following is the output result of the program is:

```
The initial value of number : 10
The final value of number : 10
The result is :110
```

The value of the variable number before calling the function is 10. The function call is made and function adds 100 to the parameter number. When the function is returned the result contains the added value. The final value of the number remains same as 10. This shows that operation on parameter does not produce effect on arguments.

4.5 Call by Reference

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

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

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

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

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

will exchange the values of m and n using their aliases(reference variables) a and b. In traditional C, this is accomplished using pointers and indirection as follows:

```
void swap1(int *a, int *b)  /*function definition */
{
int t;
t=* a;                      /* assign the value at address a to t */
* a=* b;          /* put the value at b into a*/
*b=t;             /* put the value at t into b */
}
```

This function can be called as follows:

```
swap1(&x ,&y);  /*call by passing*/
            /* addresses of variables*/
```

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

4.6 Return by Reference

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

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

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

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

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

4.7 Default Arguments

C++ allows us to call a function without specifying all its arguments. In such cases, the function assigns a default value to the parameter which does not have a matching argument in the function call. Default values are specified when the function is declared. The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values. Here is an example of a prototype (i.e. function declaration) with default values:

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

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

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

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

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

passes an explicit value of 0.12 to rate.

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

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

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

Example to illustrates the use of default arguments.

```
        #include <iostrem.h>
        using namespace std;
        int main( )
        {
float amount;
        float value(float p, int n, float r=0.15);
        void printline(char ch='*', int len=40);
        print line();
        amount = value (5000.00,5);
        cout <<"\n                    Final value = " << amount << "\n\n";
        printline('=');
        return 0;
        }
/*-----------------------------------------------------*/
float value(float p, int n, float r)
        {
        int year = 1;
        float sum = p;
        while(year <= n)
        {
        sum = sum*(l+r);
        year= year+1;
        }
        return(sum);
        }
        void printline(char ch, int len)
        {
        for(int i=l; i<=len; i++)
        printf("%c", ch);

        printf("\n");

        }
```

The following is output of the above program :

```
Final value = 10056.8
```

Advantages of providing the default arguments are :

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

4.8 Const Arguments

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

```
int strlen(const char *p);

int length(const string &s);
```

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

4.9 Summary

Function can play an important role in C++. It is possible to reduce the size of program by using functions. In C++ main() returns a value of type int to the OS. Function protytyping gives the compiller the details about the function such as the number and types of arguments and the type of return values. When the function is called, control is transferred to the first statement in the function body. After the executing of all the statements of function body control returns to the main program. Reference variables in C++ permit us to pass parameters to function by reference. C++ allows us to assign the default value to the function parameters when the function is declared. Arguments to a function can be declared as const. C++ also allows function overloading. C++ also provides library functions by the C++ standard library. A C++ function can be overloaded.

4.10 Self Assessment Questions

1. What are the advantages of function prototypes in C++?
2. What is the main advantage of passing arguments by refernce?
3. Write a program to read two metric of two dimension and find sum by using funtion?
4. Write a program to read two metric of two dimension and display multiplication by using funtion?
5. Write a program to calculate Tower of Hanoi by using function?

Unit 5 : C++ the OOP Language

Structure of the Unit:

5.0 Objective

After the end of this unit you will be able to understand:

- about the structure and union in C++
- about defining class and its member function and scope of data memters and member functions
- about the use of scope resolution operator and member dereferencing operators

5.1 Introduction

Classes are the most important features of C++. A class is an extension of the idea of structure used in C. Classes are user defined data types. In this unit, we shall discuss the concept of class and its implementation.

5.2 C Structures revisited

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

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

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

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

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

```
strcpy(A.name, "Bharat");

A. roll_ number = 999;

A.total_marks = 595.5;

Final_total = A.total_marks + 5;
```

Structures can have arrays, pointers or structures as members.

Limitations of C structure

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

```
struct complex

{

float x;

float y;

};

struct complex c1, c2, c3;
```

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

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

is illegal in C.

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

Extensions to Structures

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

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

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

Remember, this is an error in C.

C++ incorporates all these extensions in another user-defined type known as class. Then is very little syntactical difference between structures and classes in C++ and, therefore, they can be used interchangeably with minor modifications. Since class is a specially introduced data type in C++, most of the C++ programmers tend to use the structures for holding only data, and classes to hold both the data and functions.

5.3 Specifying a class

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

1. Class declaration
2. Class function definitions

The class declaration describes the type and scope of its member. The class function definitions describe how the class functions are implemented. The general form of a class declaration:

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

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

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

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

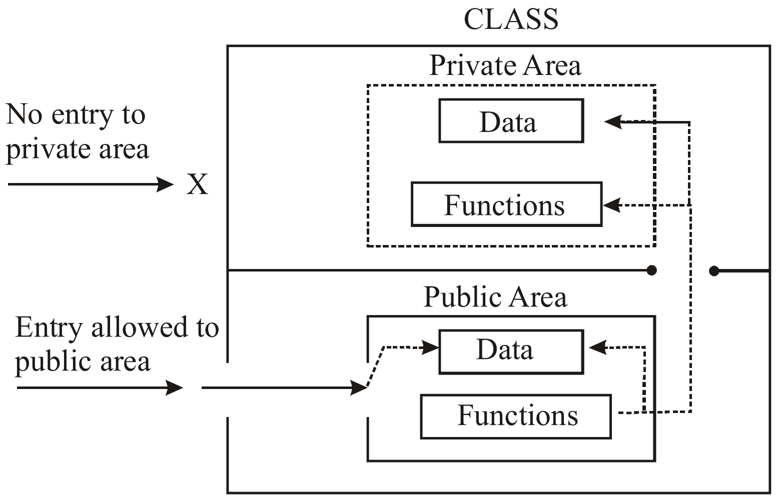


Fig 5.1 Data hiding in class

A Simple Class Example

A typical class declaration would look like:

```
        class item
        {
        int number;
        float cost;
public:
        void getdata(int a, float b);
        void putdata(void);
        };      //ends with semicolon
```

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

Creating Objects

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

For example:

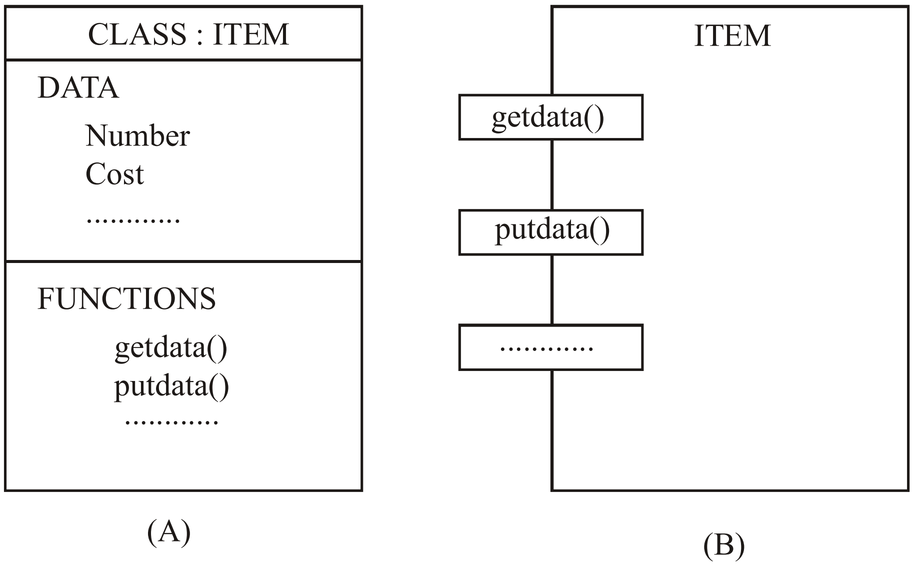


Fig. 5.2 Repersentation of a class

item x; / / memory for x is created

creates a variable x of type item. In C++, the class variables are known as objects. Therefore, x is called an object of type item. We may also declare more than one object in one statement. Example:

item x, y, z;

The declaration of an object is similar to that of a variable of any basic type. The necessary memory space is allocated to an object at this stage. Note that class specification. like a structure, provides only a template and does not create any memory space for the objects.

Objects can also be created when a class is defined by placing their names immediately after the closing brace, as we do in the case of structures. That is to say, the definition

```
class item
{
..........................
..........................
..........................
} x, y, z;
```

would create the objects x, y and Z oftype item. This practice is seldom followed because we would like to declare the objects close to the place where they are used and not at the time of class definition.

Accessing Class Members

As pointed out earlier, the private data of a class can be accessed only through the member functions of that class. The main( ) cannot contain statements that access number and cost directly. The following is the format for calling a member function:

o   bject_name. function_name (actual_arguments);

For example: the function call statement

```
x.getdata(100,75.5);
```

is valid and assigns the value 100 to number and 75.5 to cost of the object x by implement the getdata( ) function. The assignments occur in the actual function.

Similarly, the statement

```
x.putdata();
```

would display the values of data members. Remember, a member function can be invoke only by using an object (of the same class). The statement like

```
getdata(100,75.5);
```

has no meaning. Similarly, the statement

```
x.number = 100;
```

is also illegal. Although x is an object of the type item to which number belongs, then (declared private) can be accessed only through a member function and not by the object directly.

It may be recalled that objects communicate by sending and receiving messages. This achieved through the member functions. For example,

```
x. putdata () ;
```

sends a message to the object x requesting it to display its contents.

A variable declared as public can be accessed by the objects directly. Example:

```
class xyz
{
int x;
int y;
public:
int z;
};
..................
..................
xyz p;
p.x = 0;
p.z = 10
..................
..................
```

5.4 Defining member functions

Member function can be defined in two places:

- Outside the class definition
- Inside the class definition

It is obvious that, irrespective of the place of definition, the function should perform the same task. Therefore, the code for the function body would be identical in both the cases. However, there is a subtle difference in the way the function header is defined. Both these approaches are discussed in detail in this section.

Outside the Class Definition

Member functions that are declared inside a class have to be defined separately outside the class. Their definitions are very much like the normal functions. They should have a function header and a function body. Since C++ does not support the old version of function definition, the ANSI prototype form must be used for defining the function header.

An important difference between a member function and a normal function is that a member function incorporates a membership 'identity label' in the header. This 'label' tells the compiler which class the function belongs to. The general form of a member function definition is:

```
return-type class_name:: function_name (argument declaration)
    {
    Function body
    }
```

The membership label class-name :: tells the compiler that the function function-name belongs to the class class-name. That is, the scope of the function is restricted to the class-name specified in the header line. The symbol :: is called the scope resolution operator.

For instance, consider the member functions getdata( ) and putdata( ) as discussed above. They may be coded as follows:

```
void item:: getdata(int a, float b)
    {
number = a;
cost = b;
}
void item :: putdata(void)
    {
    count<< "Number :" <<number<<"\n";
    count<< "Cost :" <<cost<<"\n";
    }
```

Since these functions do not return any value, their return-type is void. Function arguments are declared using the ANSI prototype.

The member functions have some special characteristics that are often used in the program development. These characteristics are:

- Several different classes can use the same function name. The membership label' will resolve their scope.
- Member functions can access the private data of the class. A non-member function cannot do so.
- A member function can call another member function directly, without using dot operator.

Inside the class Definition

Another method of defining a member function is to replace the function declaration by actual function definition inside the class. For example, we could define the item class follows:

```
class item
{
int number;
float cost;
public:
void getdata(int a, float b);                //declaration inline function
void putdata(void)                           //definition inside the class
{
cout<<number <<"\n";
cout <<cost <<"\n";
}
};
```

When a function is defined inside a class, it is treated as an inline function. Therefore, the restrictions and limitations that apply to an inline function are also applicable here. Normally, only small functions are defined inside the class definition.

5.5 Structure, Union and Class

Data-encapsulation is key to the functionality of many high level or later generation programming languages such as C++ and Java. It is simply a means of creating "containers" called objects, and holding multiple variables of varying data types called member variables inside of those containers.

C++ offers three methods of data-encapsulation; struct, union and class. This is where C++ become object oriented.

Structure and Union

A Structure and a union are fairly similar, so they will be treated as one entity. A Structure is often useful to hold records of data, like a bank statement, or creating real-world objects, like dates and time. The syntax is as follows.

```
struct {
```

```
float savings;
 float checking;
float mortgage;
float interest;
float fees;
} Bank Statement;
union {
short month;
short day;
int year;
} DATE;
typedef struct {
int hour;
int min;
int sec;
char AmPm;
 } TIME;
```

A struct can also have a struct object as a member variable.

```
typedef struct {
bool deposit;
bool withdraw;
float Amount;
float newBalance;
DATE date;
TIME time;
 } Transaction;
```

There are no limits to how many, or what data type a struct can hold. Remember though, a struct can become quite large sometimes because it's size in memory is that of the sum total of the memory size of all of it's member variables.

When any object, a struct, union, or class, is declared as a pointer, it no longer uses dot-notation, but pointer-notation: object -> member_variable. The arrow, or "pointer" is used to signify the difference between a normal object and a pointer.

Classes

A class is where C++ become the most object oriented that it can. Classes offer many abilities and methods of storing, handling, and retrieving data. A class is very similar to a Structure in that it can have member variables, but it can also have member functions. These are functions within the class. Member functions are called the same way member variables are:

```
object.member_function(...)

class CAT {

int lives;

int getLives(); //returns the number of lives

 };
```

In a class, as well as in a struct, there are levels of access, they are public, private, and protected. We wont worry about protected access, thought, since that pertains to inheritance, and is out of the scope of this tutorial.

Anything that is in the public section of the class can be referenced and changed by anything outside of the class like a normal variable. In the private section, opposite is true. Nothing but member functions can access, change or call member variables or functions.

```
class CAT {

public:

int getLives(); //returns the number of lives

private:

int lives;

};
```

5.6 Scope Resolution Operator

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

```
................

................

{

int x = 10;

................

................

}

................

................

{
```

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

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

```
................
................
{
int x = l0;
................
................
{
int x = 1;
................
................
}
................
}
```

Block2 is contained in blockl. Note that a declaration in an inner block hides a declaration of the same variable in an outer block and, therefore, each declaration of x causes it to refer to a different data object. Within the inner block, the variable x will refer to the data object declared therein.

In C, the global version of a variable cannot be accessed from within the inner block. C++ resolves this problem by introducing a new operator :: called the scope resolution operator. This can be used to uncover a hidden variable. It takes the following form:

```
:: variable-name
```

This operator allows access to the global version of a variable. For example, ::count means the global version of the variable count (and not the local variable count declared in that block).

Example Program to illustrate the use of scope resolution operator.

```
#include <iostream.h>
using namespace std;
int m = 10;                //global m
int main( )
{
```

```
int m=20;
{
int k=m;
int m =  30;
cout  << “we  are  in  inner block  \n";
cout  << "k =  n  <<  k <<  "\n";
cout  << "m =  "  << m << "\n";
cout  << ": :m   " <<  ::m << “\n";
}
cout << "\nWe are in outer block \n";
cout << "m = " <<m << "\n";
cout <<"::m = "<< ::m<<"\n";
}
```

The following is the output of the above example

```
We are in inner block
        k  =  20
        m  =  30
        :: m = 10
We are in outer block
        m = 20
        ::m = 10
```

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

It is to be noted ::m will always refer to the global m. In the inner block, ::m refers to the value 10 and not 20.

A major application of the scope resolution operator is in the classes to identify the class to which a member function belongs.

5.7 Member Dereferencing Operators

As you know, C++ permits us to define a class containing various types of data and functions as members. C++ also permits us to access the class members through poiriters. In order to achieve this, C++ provides a set ofthree pointer-to-member operators. Table 5.1 shows these operators and their functions.

Operator	**Function**
::*	To declare a pointer to a member of a class
*	To access a member using object name and a pointer to that member
->*	To access a member using a pointer to the object and a pointer to that member

Table 5.1 Member dereferencing operator

5.8 Summary

A class is extensiton to the structure. By default, members of the class are private whereas that of structure are public. In C++ class variables are called objects. We can define member function inside or outside the class. A structure are useful to hold record type information. Block and scopes can be used in constructing program. C++ is also a block-structured programming. C++ permits us to access the class members through pointer.

5.9 Self Assessment Questions

1. Define the classe in C++ .
2. Differntiate objects vs clasess.
3. What is a class? How does it accomplish data hiding?
4. What are objects? How are they created?
5. How is a member function of a class defined?

Unit 6 : Constructors & Destructors

Structure of the Unit :

6.0 Objectives

After going through this unit you should be able to:

- Understand the concept of constructor
- Identify and describe the use of constructor
- Understand the types of constructor
- Understand the concept & use of destructor

6.1 Introduction to Constructors

In object-oriented
programming, a **constructor** (sometimes shortened to **ctor**) in a class is a special type of subroutine called at the creation of an object. It prepares the new object for use, often accepting parameters which the constructor uses to set any member variables required when the object is first created. A constructor is a member function that is automatically called when an object of that class is declared. It is called a constructor because it constructs the values of data members of the class.

A constructor similar an instance method, but it differs from a method in that it never has an explicit return-type, it is not inherited and it usually has different rules for scope modifiers. Constructors are often distinguished by having the same name as the declaring class. They have the task of initializing the object's data members. A properly written constructor will leave the object in a *valid* state. Unchangeable objects must be initialized in a constructor. The C++ run time system makes sure that the constructor of a class is the first member function to be executed automatically when an object of the class is created.

The syntax for defining a constructor with its prototype within the class, body and the actual definition outside it. Similar to other members, the constructor can be defined either within, or outside the body of a class. It can access any data members like all other member functions but cannot be invoked explicitly and must have public status to serve its purpose.

Syntax of constructor :

```
Class_Name
{ ….  // private members
Public:                           // public members
Class Name ( );                   // constructor prototype
};
Class Name :: Class Name ( )   // constructor definition
{
// Constructor body
}
```

Most languages allow overloading the constructor in that there can be more than one constructor for a class, each having different parameters.

6.2 Use of Constructors

The main use of construction is to initialize objects. The function of initialization is carried out by the use of a special member function called a constructor.

6.3 Types of constructors

6.3.1 Default Constructor

In computer programming languages the term "**default constructor**" refers to a constructor that is aut omatically generated in the absence of explicit constructors (and perhaps under other circumstances); this automatically provided constructor is usually a nullary constructor, constructor that takes no arguments. The "**default constructor**" may additionally refer to any constructor that may be called without arguments, either because it is a nullary constructor or because all of its parameters have default values.

In C++, the standard describes the default constructor for a class as a constructor that can be called with no arguments (this includes a constructor whose parameters all have default arguments). For example:

```
class MyClass
{
int x;
int y;
public:
MyClass();                          // constructor declared
};
```

```
MyClass :: MyClass()                    // constructor defined
{
x = 100;
y = 200;
}
 int main()
{
MyClass object_1;                    // object created
}
```

In C++, default constructors are significant because they are automatically invoked in certain circumstances:

- When an object value is declared with no argument list, e.g. MyClass x; or allocated dynamically with no argument list, e.g. new MyClass; the default constructor is used to initialize the object
- When an array of objects is declared, e.g. MyClass x[10]; or allocated dynamically, e.g. new MyClass [10]; the default constructor is used to initialize all the elements
- When a derived class constructor does not explicitly call the base class constructor in its initializer list, the default constructor for the base class is called
- When a class constructor does not explicitly call the constructor of one of its object-valued fields in its initializer list, the default constructor for the field's class is called

In the above circumstances, it is an error if the class does not have a default constructor.

The compiler will implicitly define a default constructor if no constructors are explicitly defined for a class. This implicitly-declared default constructor is equivalent to a default constructor defined with a blank body. For example:

```
class MyClass
    {
      .....                         // No Constructor
    };
    int main()
    {
    MyClass object_1;                    // No errors
    ....
    }
```

If some constructors are defined, but they are all non-default, the compiler will not implicitly define a default constructor. Hence, a default constructor may not exist for a class. This is the reason for a typical error which can be demonstrated by the following example.

```
class MyClass
```

```
        {
        private:
        int x;
        public:
        MyClass(int y);                         // A Constructor
        };
MyClass :: MyClass(int y)
        {
        x = y;
        }
int main()
        {
MyClass object_1(100);                  // Constructor Called
MyClass object_2;                       // Error: No default Constructor
        return 0;
        }
```

As a constructor of type other than default is defined the compiler does not define a default constructor and hence the creation of object_2 leads to an error.

6.3.2 Parameterized Constructors

Constructors that can take arguments are termed as parameterized constructors, just as the case of functions. The argument list can be specified within braces similar to the argument list in the function. The number of arguments can be greater or equal to one (1).

For example:

```
class example
{
int p, q;
public:
example(int a, int b);              //parameterized constructor
};
example :: example(int a, int b)
{
p = a;
q = b;
}
```

When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The normal way of object declaration may not work. The constructors can be called explicitly or implicitly.The method of calling the constructor implicitly is also called the *shorthand* method.

```
 example e = example(0, 50);                //explicit call
  example e(0, 50);                         //implicit call
```

6.3.3 Dynamic Constructors

Allocation of memory to objects at the time of their construction is known as dynamic construction of objects and such constructors are called as dynamic constructors. This results in saving of memory as it enables the system to allocate the right amount of memory for each object when the objects are not of the same size. The memory is allocated with the help of the new operator. For example,

```
class String
{
char *name;
int length;
public:
String()                          // constructor - 1
{
length = 0;
name = new char[length + 1];
}
String(char *e)
{
length strlen(e);                 // constructor - 2
name = new char[length + 1];
strcpy(name, e);
}
void join(example &a, example &b);
};
void String :: join(example &a, example &b)
{
 length = a.length + b.length;
delete name;
name = new char[length + 1];              // dynamic allocation
strcpy(name, a.name);
strcat(name, b.name);
};
```

6.3.4 Conversion Constructors-

Conversion constructors provide a means for a compiler to implicitly create an object of a class from an object another type. This type of constructor is different from copy constructor because it creates an object from **other class**. But copy constructor is from **the same class**. A converting constructor is a single-parameter constructor that is declared without the function specifier explicit. The compiler uses converting constructors to convert objects from the type of the first parameter to the type of the converting constructor's class. The following example demonstrates this:

```
class Y {
 int a, b;
char* name;
public:
Y(int i) { };
Y(const char* n, int j = 0) { };
};
void add(Y) { };
int main() {
  // equivalent to
// obj1 = Y(2)
Y obj1 = 2;
// equivalent to
// obj2 = Y("somestring",0)
Y obj2 = "somestring";
// equivalent to
// obj1 = Y(10)
obj1 = 10;
// equivalent to
// add(Y(5))
add(5);
}
```

The above example has the following two converting constructors:

- Y(int i)which is used to convert integers to objects of class Y.
- Y(const char* n, int j = 0) which is used to convert pointers to strings to objects of class Y.

The compiler will not implicitly convert types as demonstrated above with constructors declared with the explicit keyword. The compiler will only use explicitly declared constructors in new expressions, the static_cast expressions and explicit casts, and the initialization of bases and members.

The following example demonstrates this:

```
class A {
public:
explicit A() { };
explicit A(int) { };
};
int main() {
A z;
//  A y = 1;
A x = A(1);
A w(1);
A* v = new A(1);
A u = (A)1;
A t = static_cast<A>(1);
}
```

The compiler would not allow the statement A y = 1 because this is an implicit conversion; class A has no conversion constructors.

6.3.5 Copy constructor

A **copy constructor** is a special constructor in the C++ programming language for creating a new object as a copy of an existing object. The first argument of such a constructor is a reference to an object of the same type as is being constructed (const or non-const), which might be followed by parameters of any type (all having default values).

Normally the compiler automatically creates a copy constructor for each class (known as a **default** copy constructor) but for special cases the programmer creates the copy constructor, known as a **user-defined** copy constructor. In such cases, the compiler does not create one. Hence, there is always one copy constructor that is either defined by the user or by the system.

A user-defined copy constructor is generally needed when an object owns pointers or non-shareable references, such as to a file, in which case a destructor and an assignment operator should also be written.

Copying of objects is achieved by the use of a copy constructor and an assignment operator. A copy constructor has as its first parameter a reference to its own class type. It can have more arguments, but the rest must have default values associated with them. The following would be valid copy constructors for class X:

```
X(const X& copy_from_me);
X(X& copy_from_me);
```

```
X(const volatile X& copy_from_me);

X(volatile X& copy_from_me);

X(const X& copy_from_me, int = 10);

X(const X& copy_from_me, double = 1.0, int = 40);
```

The first one should be used unless there is a good reason to use one of the others. One of the differences between the first and the second is that temporaries can be copied with the first. For example:

```
X a = X();

// valid given X(const X& copy_from_me) but not valid given X(X& copy_from_me)

// because the second wants a non-const X&

// to create a, the compiler first creates a temporary by invoking the default constructor

// of X, then uses the copy constructor to initialize a as a copy of that temporary.

// However, for some compilers both the first and the second actually work.
```

Another difference between them is the obvious:

```
const X a;

X b = a; // valid given X(const X& copy_from_me)
```

but not valid given X(X& copy_from_me)

```
// because the second wants a non-const X&
```

The X & form of the copy constructor is used when it is necessary to modify the copied object. A reference must be provided:

```
X a;

X b = a;        // valid if any of the copy constructors are defined

                 // since a reference is being passed.
```

The following are invalid copy constructors (Reason - copy_from_me is not passed as reference) :

```
X(X copy_from_me);

X(const X copy_from_me);
```

because the call to those constructors would require a copy as well, which would result in an infinitely recursive call.

The following cases may result in a call to a copy constructor:

1. When an object is returned by value.
2. When an object is passed (to a function) by value as an argument.
3. When an object is thrown.
4. When an object is caught.
5. When an object is placed in a brace-enclosed initializer list.

These cases are collectively called copy-initialization and are equivalent to: T x = a;

6.4 Consctuctor Overloading

Like any other functions a constructor can also be overloaded in C++. The constructor's name is predetermined by the name of the class, it would seem that there can be only one constructor. Whenever multiple constructors are needed, they are implemented as overloaded functions. In case of constructors, which are automatically called when an object is created, the one executed is the one that matches the arguments passed on the object declaration. A default constructor takes no parameters, instantiating the object members with a value zero.

For example, a default constructor for a restaurant bill object written in C++ might set the Tip to 15%:

```
Bill ()
{
tip = 15.0;
total = 0.0;
}
```

The drawback to this is that it takes two steps to change the value of the created Bill object. The following shows creation and changing the values within the main program:

```
Bill cafe;
cafe.tip = 10.00;
cafe.total = 4.00;
```

By overloading the constructor, one could pass the tip and total as parameters at creation. This shows the overloaded constructor with two parameters:

```
Bill(double setTip, double setTotal)
{
tip = setTip;
 total = setTotal;
}
```

Now a function that creates a new Bill object could pass two values into the constructor and set the data members in one step. The following shows creation and setting the values:

```
Bill cafe(10.00, 4.00);
```

This can be useful in increasing program efficiency and reducing code length.

6.5 Dynamic Initialization of Objects

In C++ objects can be initialized dynamically too. That is to say, the initial value of an object may be provided during run time. Dynamic initialization is that in which initialization value isn't known at compile-time. It's computed at runtime to initialize the variable. One advantage of dynamic initialization is that we can provide various initializations formats using overloaded constructors. This provides the flexibility of using different formats of data at run time depending upon the situation.

Example:

```
        int factorial(int n)
{
if ( n < 0 )      return -1; //indicates input error
else if ( n == 0 ) return 1;
else              return n * factorial(n-1);
}
int a = 10 ;        //static initialization
                    //10 is known at compile time. Its 10!
int b = factorial(8); //dynamic initialization
                    //factorial(8) isn't known at compile time,
                    //rather it's computed at runtime.
```

That is, static-initialization *usually* involves constant-expression (which is known at compile-time), while dynamic-initialization involves non-constant expression.

C++ distinguishes between two initialization types for objects with static storage duration (global objects are an example of objects having static storage). Static initialization consists of either zero-initialization or initialization with a constant expression. These two types roughly correspond to compile-time initialization and runtime initialization. For example:

```
int x = func();
int main()
{
}
```

The global variable x has static storage. Therefore, it's initialized to 0 at the static initialization phase (this is the default value of objects with static duration). The subsequent dynamic initialization phase initializes x with the value returned from the function func(). Note that func() must be invoked for that purpose – an operation that can only take place only at runtime.

Thus, global objects might be initialized twice: once by static initialization, during which their memory storage is initialized to binary zeroes, and afterwards, they are dynamically initialized by their constructors.

6.6 Desctructors

When an object is no longer needed it can be destroyed. A class can have another special member function called the *destructor*.

Destructors are usually used to de-allocate memory and do other cleanup for a class object and its class members when the object is destroyed. A destructor is called for a class object when that object passes out of scope or is explicitly deleted. This destructor completes the operation performed by any of the constructors. *A destructor is a member function with the same name as its class prefixed by a ~ (tilde) followed by the name of its class and brackets.*

For example:

```
class X {
public:
```

```
// Constructor for class X
X();
// Destructor for class X
~X();
};
```

Similar to constructors, a destructor must be declared in the public section of a class so that it is accessible to all its users. A destructor takes no arguments and has no return type. Its address cannot be taken. Destructors cannot be declared const, volatile, const volatile or static. A destructor can be declared virtual or pure virtual. A class cannot have more than one desctructor.

If no user-defined destructor exists for a class and one is needed, the compiler implicitly declares a destructor. This implicitly declared destructor is an inline public member of its class.

The compiler will implicitly define an implicitly declared destructor when the compiler uses the destructor to destroy an object of the destructor's class type. Suppose a class A has an implicitly declared destructor. The following is equivalent to the function the compiler would implicitly define for A:

```
A :: ~A() { }
```

The compiler first implicitly defines the implicitly declared destructors of the base classes and non-static data members of a class A before defining the implicitly declared destructor of A

A destructor of a class A is *trivial* (unimportant) if all the following are true:

· It is implicitly defined

· All the direct base classes of A have trivial destructors

· The classes of all the non-static data members of A have trivial destructors

If any of the above are false, then the destructor is *nontrivial*.

A union member cannot be of a class type that has a nontrivial destructor.

Class members that are class types can have their own destructors. Both base and derived classes can have destructors, although destructors are not inherited. If a base class A or a member of A has a destructor, and a class derived from A does not declare a destructor, a default destructor is generated.

The default destructor calls the destructors of the base class and members of the derived class. The destructors of base classes and members are called in the reverse order of the completion of their constructor:

1. The destructor for a class object is called before destructors for members and bases are called.
2. Destructors for non-static members are called before destructors for base classes are called.
3. Destructors for non-virtual base classes are called before destructors for virtual base classes are called.

When an exception is thrown for a class object with a destructor, the destructor for the temporary object thrown is not called until control passes out of the catch block.

They are implicitly called at program termination for constructed external and static objects. Destructors are invoked when you use the delete operator for objects created with the new operator.

For example:

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

```
#include <string.h>
class Y {
private:
 char * string;
 int number;
public:
 // Constructor
Y(const char*, int);
 // Destructor
 ~Y() { delete[] string; }
};
// Define class Y constructor
Y::Y(const char* n, int a) {
 string = strcpy(new char[strlen(n) + 1 ], n);
 number = a;
}
int main () {
 // Create and initialize
// object of class Y
 Y yobj = Y("somestring", 10);
 // ...
 // Destructor ~Y is called before
 // control returns from main()
}
```

You can use a destructor explicitly to destroy objects, although this practice is not recommended. However to destroy an object created with the placement new operator, you can explicitly call the object's destructor. The following example demonstrates this:

```
#include <new.h>
#include <iostream.h>
using namespace std;
class A {
 public:
 A() { cout << "A::A()" << endl; }
~A() { cout << "A::~A()" << endl; }
};
int main () {
char* p = new char[sizeof(A)];
A* ap = new (p) A;
ap->A::~A();
 delete [] p;
}
```

The statement A* ap = new (p) A dynamically creates a new object of type A not in the free store but in the memory allocated by p. The statement delete [] p will delete the storage allocated by p, but the run time will still believe that the object pointed to by ap still exists until you explicitly call the destructor of A (with the statement ap->A::~A()).

6.7 Summary

C++ provides a special member function called the constructor which enables an object to initialize itself when it is created. This is known as automatic initialization of objects. A constructor has the same name as that of class. Constructor may be over loaded. When an object is created and initialized at the same time, a copy constructor gets called. C++ also provides another member function called the destructor that destroys the objects.

6.8 Self Assessment Questions

1. What is constructor?
2. Is it mandatory to use constructor in a class?
3. How they differ from normal functions?
4. How do we invoke a constructor?
5. What is a paramaterized constructor?
6. What are copy constructor and explain their need?
7. Describe the importance of destructors?

Unit 7 : More on C++ Functions

Structure of the Unit :

7.0 Objectives

In this unit we shall study function overloading, friend and virtual functions, inline functions, static member functions and object as Arguments.

7.1 Introduction

A function is a set of program statements that can be processed independently. Two functions are overloaded if they have the same name, are declared in the same scope, and have different parameter lists. In this unit, we first look at how to declare a set of overloaded functions and why it is useful to do so.

We then briefly look at how function overload resolution proceeds — that is, how a function call is resolved to one function in a set of overloaded functions. Function overload resolution is one of the most complex aspects of C++.

A virtual function is a function or method whose behavior can be overriden within inheriting class by a function with the same signature. Inline functions is the optimization techniques used by the compilers. The actual code then gets placed in the calling program.

A static member functions can be used to static members. In objects as function arguments, a copy of the entire object can be passed by using pass-by-value and pass-by-reference to the function.

7.2 Function overloading

Function polymorphism or function overloading is a concept that allows multiple functions to show the same name in the different argument types. Function polymorphism implies that the function definition can have multiple forms. Assign one or more function body to the same name is known as function overloading or function non a overloading. Using the concept of function overloading, we can design a family of functions with one function name but with different argument lists.

7.2.1 Overloaded Function Declaration

We know how to declare and define functions and how to use functions in our programs, Function overloading allows multiple functions that provide a common operation on different parameter types to share a common name. If you have written an arithmetic expression in a programming language, you have used a predefined overloaded function. For example, the expression.1 + 3 invokes the addition operation for integer operands, whereas the expression 1.0 + 3.0 invokes a different addition operation that handles floating point operands. The operation that is actually used is transparent to the user. The addition operation is overloaded to handle the different operand types. It is the responsibility of the compiler, and not of the programmer, to distinguish between the different operations and to apply the appropriate operation depending on the operands' types. In this unit, we see how to define our own overloaded functions.

7.2.2 Why Overload a Function Name

As is the case with the built-in addition operation, we may want to define a set of functions that perform the same general action but that apply to different parameter types. For example, suppose we want to define functions that return the largest of their parameters' values.

Without the ability to overload a function name, we must give each function its own unique name. For example, we could define a set of max() functions as follows:

```
int i_max( int, int );

int vi_max( const vector<int> & );

int matrix_max( const matrix & );
```

Each function, however, performs the same general action; each one returns the largest of its parameters' set of values. From a user's viewpoint, there is only one operation, that of determining a maximum value.

The implementation details of how that is accomplished are of little interest to the users of the function. This lexical complexity is not intrinsic to the problem of determining the largest of a set of values but rather reflects a limitation of the programming environment in which each name occurring at the same scope must refer to a unique entity (a unique object, function, class type, and so on). Such complexity presents a practical problem to the programmer, who must remember or look up each name. Function overloading relieves the programmer of this lexical complexity. With function overloading, the programmer can simply write the following:

```
int ix = max( j, k );

vector<int> vec;

// ...

int iy = max( vec );
```

This technique gets the largest value in a variety of situations.

7.3 Friend and Virtual Function

7.3.1 Friend function

As discussed in earlier section on access specifies, when a data is declared as private inside a class, then it is not accessible from outside the class. A function that is not a member will not be able to access the private data, that is, a non-member function cannot have an access to the private data of class. However, there could be a situation where we would like two classes to share a particular function.

A frient function is used for accessing the non-public members of a class. A class can allow non-member function and other classes to access its own private data, by making them friends. Thus, a friend function is an ordinary function or a member of another class.

Special characteristics of friend function:

1. The scope of a friend function is not limited to the class in which it has been declared as a Friend.
2. A friend function cannot be called using the object of that class.
3. It cannot access members directly.
4. It can be declared either in the public or the private part of a class without affecting its meaning.

Example program to illustrates the use of a friend function.

```
# include < isotream.h>
class sample {
int a;
int b;
public:
void set value ( ) {a = 25; b = 40;}
friend float mean (sample  s);
};
float mean (sample s)
{
return float (s.a + s.b)/2.0;
}
int main ( ) {
Sample X; //object x
X.set values ( ) ;
cout << "Mean value =" <<(X) << "m"; return 0;
}
```

The following is the output of above program :

```
Mean value = 32.5
```

7.3.2 Virtual Functions

By default, C++ matches a function call with the correct function definition at compile time. This is called static binding. You can specify that the compiler match a function call with the correct function definition at run time; this is called dynamic binding. You declare a function with the keyword virtual if you want the compiler to use dynamic binding for that specific function.

C++ virtual function is a member function of a class, whose functionality can be over ridden in its derived classes. The whole function body can be replaced with a new set of implementation in the desired class. The concept of virtual function is different from C++ function overloading. C++ virtual function is

- A member function of a class
- Declared with virtual keyword
- Usually has a different functionality in the desired class.
- A function call is resolved at run-time.

The following examples demonstrate the differences between static and dynamic binding.

```
        #include <iostream.h>
using namespace std;
        struct A {
        void f() { cout << "Class A" << endl; }
        };
struct B: A {
        void f() { cout << "Class B" << endl; }
        };
void g(A& arg) {
        arg.f();
        }
        int main() {
        B x;
        g(x);
        }
```

The following is the output of the above example:

```
Class A
```

When function g() is called, function A::f() is called, although the argument refers to an object of type B. At compile time, the compiler knows only that the argument of function g() will be a reference to an object derived from A; it cannot determine whether the argument will be a reference to an object of type A or type B. However, this can be determined at run time. The following example is the same as the previous example, except that A: : f() is declared with the virtual keyword:

```
        #include <iostream.h>
        using namespace std;
        struct A {
        virtual void f() { cout << "Class A" << endl; }
        };
```

```
struct B: A {
void f() { cout << "Class B" << endl; }
};
void g(A& arg) {
arg.f();
}
int main() {
B x;
g(x);
}
```

The following is the output of the above example:

```
Class B
```

The virtual keyword indicates to the compiler that it should choose the appropriate definition of f() not by the type of reference, but by the type of object that the reference refers to.

Therefore, a virtual function is a member function you may redefine for other derived classes, and can ensure that the compiler will call the redefined virtual function for an object of the corresponding derived class, even if you call that function with a pointer or reference to a base class of the object.

A class that declares or inherits a virtual function is called a polymorphic class.

You redefine a virtual member function, like any member function, in any derived class. Suppose you declare a virtual function named f in a class A, and you derive directly or indirectly from A a class named B. If you declare a function named f in class B with the same name and same parameter list as A::f, then B::f is also virtual (regardless whether or not you declare B::f with the virtual keyword) and it overrides A::f. However, if the parameter lists of A::f and B::f are different, A::f and B::f are considered different, B::f does not override A: :f , and B::f is not virtual (unless you have declared it with the virtual keyword). Instead B::f hides A:: f.

The following example demonstrates this:

```
#include <iostream.h>
using namespace std;
struct A {
virtual void f() { cout << "Class A" << endl; }
};
struct B: A {
void f(int) { cout << "Class B" << endl; }
};
struct C: B {
void f() { cout << "Class C" << endl; }
};
int main() {
B b; C c;
A* pa1 = &b;
A* pa2 = &c;
// b.f();
pa1->f();
pa2->f(); }
```

The following is the output of the above example:

```
Class A
Class C
```

The function B::f is not virtual. It hides A::f. Thus the compiler will not allow the function call b.f(). The functionC::f is virtual; it overrides A::f even though A::f is not visible in C.If you declare a base class destructor as virtual, a derived class destructor will override that base class destructor, even though destructors are not inherited.

The return type of an overriding virtual function may differ from the return type of the overridden virtual function. This overriding function would then be called a covariant virtual function. Suppose that B::f overrides the virtual functionA::f. The return types of A::f and B::f may differ if all the following conditions are met:

- The function B::f returns a reference or pointer to a class of type T, and A::f returns a pointer or a reference to an unambiguous direct or indirect base class of T.
- The const or volatile qualification of the pointer or reference returned by B::f has the same or less const or volatile qualification of the pointer or reference returned by A::f.
- The return type of B::f must be complete at the point of declaration of B::f, or it can be of type B.

The following example demonstrates this:

```
#include <iostream.h>
using namespace std;
struct A { };
class B : private A {
friend class D;
friend class F;
};
A global_A;
B global_B;
struct C {
virtual A* f() {
```

```
cout << “A* C::f()” << endl;
return &global_A;
}
};
struct D : C {
B* f() {
cout << “B* D::f()” << endl;
return &global_B;
}
};
struct E;
struct F : C {
//  Error:
//  E is incomplete
//  E* f();
};
struct G : C {
//  Error:
//  A is an inaccessible base class of B
//  B* f();
};
int main() {
D d;
C* cp = &d;
D* dp = &d;
A* ap = cp->f();
B* bp = dp->f();
};
```

The following is the output of the above example:

```
B* D::f()
B* D::f()
```

The statement A* ap = cp->f() calls D::f() and converts the pointer returned to type A*. The statement B* bp = dp->f() calls D::f() as well but does not convert the pointer returned; the type returned is B*. The compiler would not allow the declaration of the virtual function F::f() because E is not a complete class. The compiler would not allow the declaration of the virtual function G::f() because class A is not an accessible base class of B (unlike friend classes D and F, the definition of B does not give access to its members for class G).

A virtual function cannot be global or static because, by definition, a virtual function is a member function of a base class and relies on a specific object to determine which implementation of the function is called. You can declare a virtual function to be a friend of another class.

If a function is declared virtual in its base class, you can still access it directly using the scope resolution (::) operator. In this case, the virtual function call mechanism is suppressed and the function implementation defined in the base class is used. In addition, if you do not override a virtual member function in a derived class, a call to that function uses the function implementation defined in the base class.

7.4 Static member functions

C++ allows the definition of static functions. Then static functions can access only the static members (data or function) declared in same class. You cannot have static and nonstatic member functions with the same names and the same number and type of arguments.

Like static data members, you may access a static member function f() of a class A without using an object of class A.

A static member function does not have a this pointer. The following example demonstrates this:

```
#include <iostream.h>
using namespace std;
struct X {
private:
int i;
static int si;
public:
void set_i(int arg) { i = arg; }
static void set_si(int arg) { si = arg; }
void print_i() {
cout << "Value of i = " << i << endl;
cout << "Again, value of i = " << this->i << endl;
}
static void print_si() {
cout << "Value of si = " << si << endl;
//   cout << "Again, value of si = " << this->si << endl;
}
};
int X: : si = 77;      // Initialize static data member
int main() {
X xobj;
xobj.set_i(11);
xobj.print_i();
 // static data members and functions belong to the class and
// can be accessed without using an object of class X
X::print_si();
X::set_si(22);
X::print_si();
Return 0 ;
}
```

The following is the output of the above example:

```
Value of i = 11
Again, value of i = 11
Value of si = 77
Value of si = 22
```

The compiler does not allow the member access operation this >si in function A::print_si() because this member function has been declared as static, and therefore does not have a this pointer.

You can call a static member function using the this pointer of a nonstatic member function. In the following example, the nonstatic member function printall() calls the static member function f() using the this pointer:

```
#include <iostream.h>
using namespace std;
class C {
static void f() {
cout << "Here is i: " << i << endl;
}
static int i;
int j;
public:
C(int firstj): j(firstj) { }
void printall();
};
void C::printall() {
cout << "Here is j: " << this->j << endl;
this->f();
}
int C::i = 3;
int main() {
C obj_C(0);
obj_C.printall();
return 0;
}
```

The following is the output of the above example:

```
Here is j: 0
Here is i: 3
```

A static member function cannot be declared with the keywords virtual, const, volatile, or const volatile. It can also be defined in the private region of a class.

A static member function can access only the names of static members, enumerators, and nested types of the class in which it is declared. Suppose a static member function f() is a member of class X. The static member function f() cannot access the nonstatic members X or the nonstatic members of a base class of X.

7.5 Inline Functions

The inline functions are C++ enhancement designed to speed up programs. The coding of normal functions and inline functions is similar except that inline functions definitions start with the keyword inline.

Inline functions are functions whereas the call is made to inline functions. The actual code then gets placed in the calling program. The format of inline function is same as normal function but when it is completed it is compiled as inline code. The function is placed separately as inline function, thus adding readability to the source program. When the program is compiled, the code present in function body is replaced in the place of function call.

The following example demonstrates this:

```
# include <iostream.h>
using name space std;
int exforsys (int); // function prototype
void main ()
{
int x;
count <<"in enter the input value:";
cin >>x;
cont << "In the output is;" << exforsys (x); //calling inline function
}
```

```
inline int exforsys (int x1) // function definition
{
return 5*D;
}
```

The following is the output of the above example.

```
Enter the inpur value, 10
The output is: 50
```

Some of the situations where inline function is unable to perform:

- for functions within the returning values.
- if a loop, a switch, or a goto exists.
- if return statement exists.
- if functions contain static variables
- if it is recursive.

7.6 Objects as Function Arguments

It is possible to have functions which accepts objects of a class as arguments, just as there is functions which accept other variable as arguments. This can be done in two ways:

- A copy of the entire object is passed to the function.
- Only the address of the object is transferred to the functions implicitly or explicitly.

Example Program to illustrates the use of objects as function arguments.

```
# include <isostream.h>
using namespace std;
class time
{
int hours;
int minutes;
public:
void gettime (int h, int m)
{hours = h; minutes = m;}
void putline (void)
{count << hours << "hours and";
cont << minutes << "minutes" << "in ";
}
void sum (time, time) // declaration with objects as arguments
};
void time :: sum(time t1, time t2)
{
minutes + t1. minutes + t2 minutes;
hours = minutes/60;
minutes = minutes? 60;
hours = hours + t1. hours + t2. hours;}
void main( )
{
time T1, T2, T3;
T1. gettime (2, 45); // get T1
T2. gettime (3, 30); // get T2
T3. sum (T1, T2); // T3 = t1 + T2
count << "T1= "; T1. puttime ();
cont <, "T2= ", T2. puttime ();
cont << "T3= "; T3 puttime () ;
}
```

The following is the output of the above program:

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

7.7 Nesting of Member function

A member function of a class can be called only by an object of the same class. However, there is an exception to this. A member function can be called by using its name inside another member function of the same class. This is known as nesting of member functions.

Example Program to illustrates the use of nesting of member function.

```
# include <iastream.h>
using namespace std;
class sat
{
 int m, n;
public:
void input (void);
void display (void);
void largest (void);
}:
int set :: largest (void)
{
if (m > n)
return (m);
else
return (n);
}
void set :: input (void)
{
cout << “input value of m and n” << “n”;
cin >> m >> n;
}
void set :: display (void)
{
cont << “largest value = “ << largest ( ) << ‘In”; // Nesting of member function.
}
void main ( )
{
Set A;
A. input ( ) ;
A display ;
}
```

The following is the output of the above program;

```
input value of m and n 25. 18
largest value = 25
```

7.8 Summary

It is possible to reduce the size of program by calling and using function at different places in the program. C++ allows function over loading. That is, we can have more than one function with same name in our program. C++ supports two new types of functions, namely friend functions and virtual functions. Non member function may access non-public member of a class. A member function of a class can be called by member function of same class this known as a nesting of a member function.

7.9 Self Assessement Questions

1. What do you meant by overloading of a function?
2. When will you make a function virtual and friend?
3. When will you use function inline?
4. Write the use of object passing as argument in function?

UNIT 8 : Operator Overloading and Type Conversions

Structure of the Unit :

8.0 Objectives

After going through this unit, you would be able to

- Explain the concept of Overloading
- Manipulation of String
- Rules of Overloading

8.1 Introduction

This unit discusses about Operator overloading, Operator overloading is the ability to tell the compiler how to perform a certain operation when its corresponding operator is used on one or more variables. It is an important technique that has enhanced the power of extensibility of C++.

In C++ we cannot overload the following operator:

1. Class member access operators (., . *)
2. Scope resolution operator (::).
3. Size operator (size of).
4. Conditional operator (?:)

The semantics of an operator can be extended but we cannot change its syntax, grammer rule, precedence and associativity.

8.2 Operator overloading

The operator keyword declares a function specifying what operator-symbol means when applied to instances of a class. This gives the operator more than one meaning, or "overloads" it. The compiler distinguishes between the different meanings of an operator by examining the types of its operands.

To define an operator loading, we must define a relation to which the operator is applied. The general term of an operator function is :

```
return type classname : : operator op(argument list)
{
function body                  // task defined
}
```

Where, return type is the type of value returned by the specified operation, and op is the operator being overload.

The operator keyboard is used with function name (op). The operator function may be either member function or friend functions. If it is a number function, then it doesnot use any arguments for unary operator and only one for binary operator. If it is friend function, then it require one argument for unary operators and two for binary operators, the process of overloading requires

1. Creation of class,
2. Declaration the operator function in public section of class.
3. Define the operator function.

```
int  operator == (int);
friend int operator ==  (vector, vector);
```

We can redefine the function of most built-in operators globally or on a class-by-class basis. Overloaded operators are implemented as functions.

The name of an overloaded operator is operator x, where x is the operator as it appears in the following table. For example, to overload the addition operator, you define a function called operator+. Similarly, to overload the addition/assignment operator, +=, define a function called operator+=.

Redefinable Operators

Operator	Name	Type
,	Comma	Binary
!	Logical NOT	Unary
!=	Inequality	Binary
%	Modulus	Binary
%=	Modulus assignment	Binary
&	Bitwise AND	Binary
&	Address-of	Unary
&&	Logical AND	Binary
&=	Bitwise AND assignment	Binary
( )	Function call	-
( )	Cast Operator	Unary
*	Multiplication	Binary
*	Pointer dereference	Unary
*=	Multiplication assignment	Binary

+	Addition	Binary
+	Unary Plus	Unary
++	Increment1	Unary
+=	Addition assignment	Binary

8.3 Overloading Unary Operators

You overload a unary operator with either a nonstatic member function that has no parameters, or a nonmember function tha t has one parameter. Suppose a unary operator @ is called with the statement @t, where t is an object of type T. A nonstatic member function that overloads this operator would have the following form:

```
return_type operator@()
```

A nonmember function that overloads the same operator would have the following form:

```
return_type operator@(T)
```

An overloaded unary operator may return any type.

Example overloads the ! operator:

```
#include <iostream.h>
using namespace std;
struct X { };
void operator!(X) {
cout << "void operator!(X)" << endl;
}
struct Y {
void operator!() {
cout << "void Y::operator!()" << endl;
}
};
struct Z { };
int main() {
X ox; Y oy; Z oz;
!ox;
!oy;
// !oz;
}
```

The following is output of the above example

```
void operator! (X)
void Y:: operator!( )
```

8.4 Overloading Binary Operators

The concept of overloading unary operators applies also to the binary operators. You overload a binary unary operator with either a nonstatic member function that has one parameter, or a nonmember function that has two parameters. Suppose a binary operator @ is called with the statement t @ u, where t is an object of type T, and u is an object of type U. A nonstatic member function that overloads this operator would have the following form:

return_type operator@(T)

A nonmember function that overloads the same operator would have the following form:

return_type operator@(T, U)

An overloaded binary operator may return any type.

Example overloads the * operator:

```
        struct X {
        // member binary operator
        void operator*(int) { }
};

// non-member binary operator
void operator*(X, float) { }

int main() {
  X x;
  int y = 10;
  float z = 10;

  x * y;
  x * z;
}
```

Overloading Binary Operator using friends

The friend function may be used in the place of member function of overloading a binary operator. The friend function requires two arguments to be explicitly passed to it, while a member function requires only one.

Example C++ program to illustrate overloading binary operator using friend function

```
#include<iostream.h>
#include<conio.h>
class complex
```

```
{
float x,y;
public:
complex() {}
complex(float real,float img)
{
x=real;
y=img;
}
friend complex operator+(complex,complex);
void display();
};
complex operator+(complex c,complex d)
{
complex temp;
temp.x=c.x+d.x;
temp.y=c.y+d.y;
        return(temp);
        }
        void complex::display()
        {
        cout<<x<<"\t"<<y<<endl;
        }
        void main()
        {
        clrscr();
        complex c1(2.5,3.5),c2(1.6,2.7),c3=c1+c2;
        c1.display();
        c2.display();
        c3.display();
        getch();
       }
```

The following is output of the above program:

```
2.5   3.5
1.6   2.7
4.1   6.2
```

8.5 Manipulation of string

C++ provides convenient and powerful tools to manipulate strings.strings are not a built-in data type,whenever we want to use strings or string manipulation tools, we must provide the appropriate #include directive, as shown below:

```
#include <string.h>
using namespace std;    // Or using std::string;
```

We now use string in a similar way as built-in data types, as shown in the example below, declaring a variable name:

```
string name;
```

Unlike built-in data types (int, double, etc.), when we declare a string variable without initialization (as in the example above), we do have the guarantee that the variable will be initialized to an empty string - a string containing zero characters.

C++ strings allow you to directly initialize, assign, compare, and reassign with the intuitive operators, as well as printing and reading (e.g., from the user), as shown in the example below:

```
string name;
cout << "Enter your name: " << flush;
cin >> name;
// read string until the next separator
 // (space, newline, tab)
// Or, alternatively:
getline (cin, name);
// read a whole line into the string name
if (name == "")
{
cout << "You entered an empty string, "
<< "assigning default\n";
name = "John";
}
else
{
cout << "Thank you, " << name
<< "for running this simple program!"
<< endl;
}
```

C++ strings also provide many string manipulation facilities. The simplest string manipulation that we commonly use is concatenation, or addition of strings. In C++, we can use the + operator to concatenate (or "add") two strings, as shown below:

```
string result;
string s1 = "hello ";
string s2 = "world";
result = s1 + s2;
// result now contains "hello world"
```

8.6 Rules for overloading operators

The rules constrain how overloaded operators are implemented

- You cannot define new operators, such as **.
- You cannot redefine the meaning of operators when applied to built-in data types.
- Overloaded operators must either be a nonstatic class member function or a global function. A global function that needs access to private or protected class members must be declared as a friend of that class. A global function must take at least one argument that is of class or enumerated type or that is a reference to a class or enumerated type. For example :

```
// rules_for_operator_overloading.cpp
class Point
{
public:
Point operator<( Point & );  // Declare a member operator
                //  overload.
// Declare addition operators.
friend Point operator+( Point&, int );
friend Point operator+( int, Point& );
};
int main()
{
}
```

The preceding code sample declares the less-than operator as a member function; however, the addition operators are declared as global functions that have friend access. Note that more than one implementation can be provided for a given operator. In the case of the preceding addition operator, the two implementations are provided to facilitate commutativity.

- Operators obey the precedence, grouping, and number of operands dictated by their typical use with built-in types. Therefore, there is no way to express the concept "add 2 and 3 to an object of type Point," expecting 2 to be added to the *x* coordinate and 3 to be added to the *y* coordinate.
- Unary operators declared as member functions take no arguments; if declared as global functions, they take one argument.
- Binary operators declared as member functions take one argument; if declared as global functions, they take two arguments.
- If an operator can be used as either a unary or a binary operator (**&**, *****, **+**, and **-**), you can overload each use separately.
- Overloaded operators cannot have default arguments.
- All overloaded operators except assignment (**operator=**) are inherited by derived classes

8.7 Type Conversions

It is the process of converting one type into another. In other words converting an expression of a given type into another is called type casting.

There are two ways of achieving the type conversion namely: Automatic Conversion otherwise called as Implicit Conversion, Explicit Conversion.

In implicitly conversion, this is not done by any conversions or operators. In other words the value gets automatically converted to the specific type to which it is assigned.

Let us see this with an example:

```
        #include <iostream.h>
using namespace std;
void main()
{
short x=6000;
int y;
y=x;
}
```

In the above example the data type short namely variable x is converted to int and is assigned to the integer variable y. So as above it is possible to convert short to int, int to float and so on.

The explicit conversion can be done using type cast operator and the general syntax for doing this is

datatype (expression);

Here in the above datatype is the type which the programmer wants the expression to gets changed as.

The type casting can be done in either of the two ways mentioned below namely:

C-style casting

C++-style casting

The C-style casting takes the synatx as

(type) expression

This method is also allowed in C++. The C++ style casting is : type (expression)

8.8 Summary

Operator overloading is the ability to tell the compiler how to perform a certain operation when its corresponding operator is used on one or more variables. Overload a binary unary operator with either a nonstatic member function that has one parameter, or a nonmember function that has two parameters. While overload a unary operator with either a nonstatic member function that has no parameters, or a nonmember function that has one parameter.

The process of converting one type into another. Automatic Conversion otherwise called as Implicit Conversion, Type casting otherwise called as Explicit Conversion.

8.9 Self Assessment Questions

1. What is Operator Overloading ?
2. Define Unary Operater ?
3. Explain Binary Operator with Friend function ?
4. Write about rule of overloading operators.
5. Write note on type of conversions

Unit 9 : Inheritance

Structure of the Unit

9.0 Objectives

After going through this lesson, you would be able to

- Explain the concept of inheritance
- Need of Inhertiance
- Describe the five forms of inheritance
- Explain the visibility modes

9.1 Introduction

This unit discusses about inheritance, the capability of one class to inherit properties from another class as a child inherits some properties from his/her parents. The most important advantage of inheritance is code reusability. Once a base class is written and debugged, it can be used in various situations without having to redefine it or rewrite it. Reusing existing code saves time, money and efforts of writing the code again. Without redefining the old class, you can add new properties to desired class and redefine an inherited class member function.

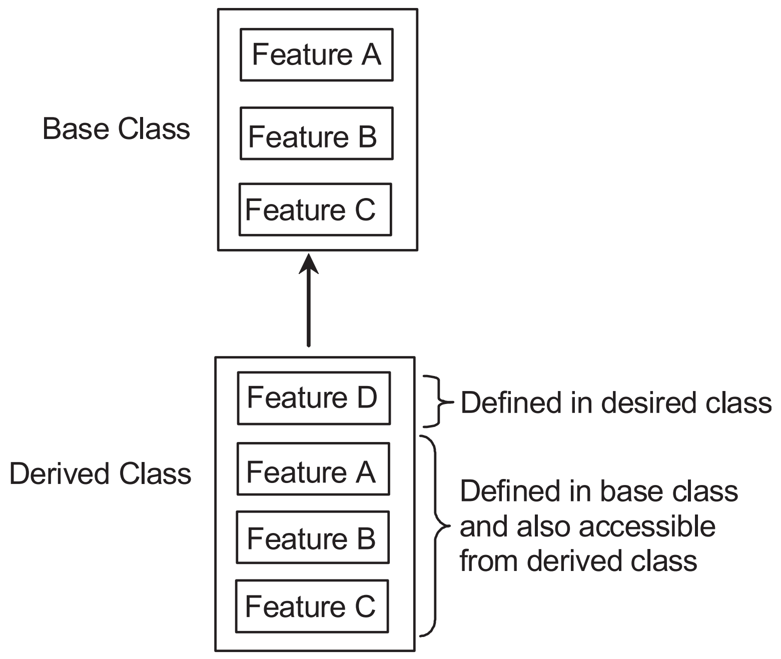
9.2 Inheritance

Inheritance is one of the important concepts of object-oriented language. There are several reasons why this concept was introduced in object oriented language. Some major reasons are:

(i) The capability to express the inheritance relationship which ensures the closeness with the real world model.

(ii) Idea of reusability, i.e., the new class can use some of the features of old class.

(iii) Transitive nature of inheritance, i.e., it can be passed on further.



Base class and derived class relationship

9.3 Defining Derrived Classes

A derived class is defined by specifying its relationship with the base class using visibility mode. The general form of defining a derived class is :

```
class derived_class : visibility_mode base_class
{
___________________
___________________// members of derived class.
};
```

The colon indicates that the derived_class is derived from base_class.

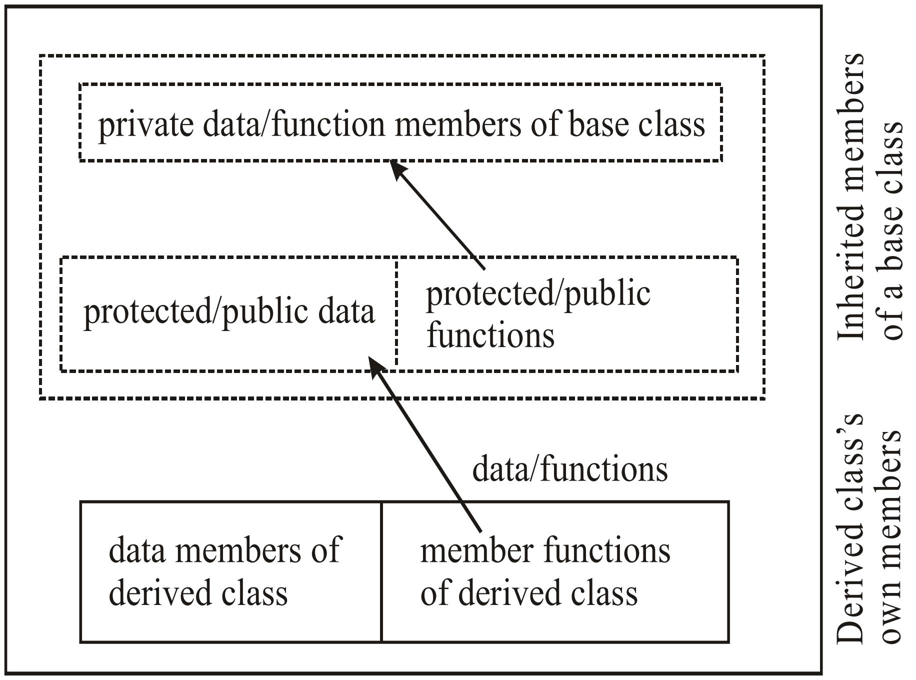
The base class(es) name(s) follow(s) the colon (:). The names of all the base classes of a derived class follow : (colon) and are separated by comma. The visibility-mode can be either private or public or protected. If no visibility mode is specified, then by default the visibility mode is considered as private.

Following are some examples of derived class definitions:

```
class Marksheet : public student / / public derivation
{
// members of derived class
};
class Marksheet : private student / / private derivation
// members of derived class
};
class Marksheet : protected student // protected derivation
{
// members of protected class
};
```

In the above definitions, Marksheet is the derived class of student base class. The visibility mode public indicates that student is a public base class. Similarly, the visibility modes private or protected indicates that student is private base class or protected base class respectively.

When we say that members of a class are inheritable, it means that the derived class can access them directly. However, the derived class has access privilege only to the non-private members of the base class. Although the private members of the base class cannot be accessed directly, yet the objects of derived class are able to access them through the non-private inherited members.


private data/function members of base class
protected/public data
protected/public functions
data/functions
data members of derived class
member functions of derived class
Inherited members of a base class
Derived class's own members

Members of derived class on inheritance

Example Program to illustrate the use of inheritance

```
#include <iostream.h>
class Base
{
public;
    int m_nPublic; //can be accessed by anybody
private;
```

```
    int m_nPrivate; // can only be accessed by Base member functions (but not derived classes)
protected;
    int m_nProtected; // can be accessed by Base member function, or derived classes.
};

Class Derived : public Base
{
public;
    Derived()
    {
        // Derived's access to Base members is not influenced by the type of inheritance used,
        // so the following is always true;

        m_nPublic = 1; //allowed : can access public base members from derived class
        m_nPrivate =2; //not allowed: can not access private base members from derived class
        m_nProtected =3; //allowed: can access protected base members from derived class
    }
};

int main()
{
    Base cBase;
    cBase.m_nPublic =1; //allowed: can access public members from outside class
    cBase.m_nPrivate= 2; // not allowed: can not access private members from outside class
    cBase.m_nProtected =3; // not allowed: can not access protected members from outside class
    }
```

9.4 Different forms of inheritance

The mechanism of deriving a new class from an old one is called inheritance (or derivation). The old class is referred to as the base class and new one is called the derived class. There are various forms of inheritance.

9.4.1 Single inheritance : A derived class with only one base class is called single inheritance.

Example Program to illustrate the Single of inheritance

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

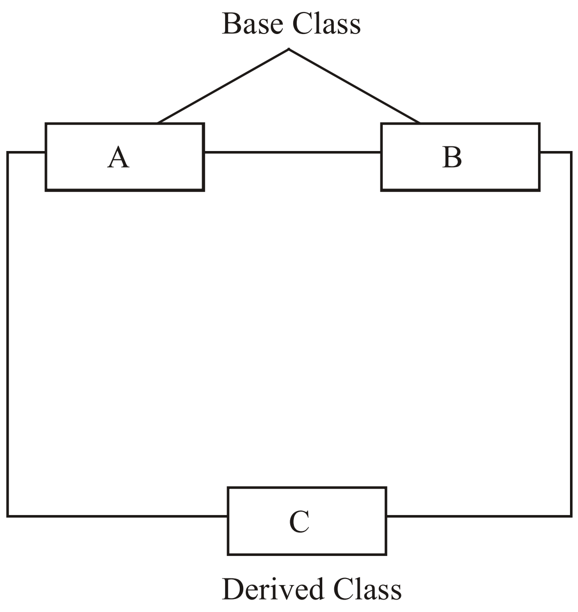
```
{
int a1, a2;
public:
void getdata()
{
cout<<"\n Enter value of a1 and a2";
cin >> a1>>a2;
}
void putdata()
{
cout<<"\n value of a1 is" <<a1 <<" and a2 is" <<a2;
}
};
class B: public A //class B is publicly derived by class A
{
int b1, b2;
public:
void indata()
{
cout <<"\n Enter the value of b1 and b2";
cin>>b1>>b2;
}
void outdata()
{
cout<<"\n value of b1 is "<<b1<<" and b2 is"<<b2;
}
};
void main()
{
B b;
b.getdata();     //base class member function
b.indata();      //derived class member function
b.putdata();
b.outdata();
}
```

The following is the output of the above example:

```
Enter value of a1 and a2
2
3
Enter value of b1 and b2
4
5
value of a1 is 2 and a2 is 3
value of b1 is 4 and b2 is 5
```

9.4.2 Multiple inheritance:

A derived class with several base classes is called multiple inheritance.



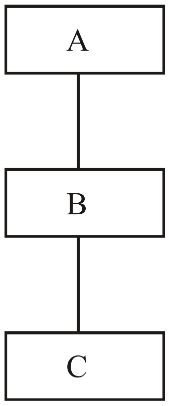
Example Program to illustrate Multiple Inheritance

```
#include<iostream.h>
class A
{
int a1. a2;
public:
void getdata()
```

```
{
cout<<"\n Enter value of a1 and a2";
cin>>a1>>a2;
}
void putdata()
{
cout<<"\n the value of a1 is "<<a1 <<" and a2 is "<<a2;
}
};
class B
{
int b1, b2;
public:
void indata()
{
cout<<"\n Enter the value of b1 and b2";
cin>>b1>>b2;
}
void outdata()
{
cout<<"\n the value of b1 is "<<b1<<" and a2 is "<<b2;
}
};
classC:public A, public B
{
int c1, c2;
public:
void input()
{
cout<<"\n Enter the value of c1 and c2";
cin >>c1>>c2;
}
void output()
```

```
{
cout<<"\n the value of c1 is "<<c1<<" and c2 is "<<c2;
}
};
void main()
{
C c
c.getdata(); //member function of class A
c.indata(); //member function of class B
c.input(); // member function of class C
c.putdata(); //member function of class A
c.outdata(); //member function of class B
c.output(); //member function of class C
}
Enter the value of a1 and a2
5
4
Enter the value of b1 and b2
8
7
enter the value of c1 and c1
9
3
the value of a1 is 5 and a2 is 4
the value of b1 is 8 and b2 is 7
the value of c1 is 9 and c2 is 3
```

9.4.3 Multilevel inheritance: The mechanism of deriving a class from another derived class is called multilevel inheritance.



Multilevel inheritance

Example Program to illustrate Multilevel Inheritance

```
#include<iostream.h>
class A
{
int a1,a2;
public:
void getdata()
{
cout<<"\n Enter value of a1 and a2";
cin>>a1>>a2;
}
void putdata()
{
cout<<"\n the value of a1 is "<<a1<<" and a2 is"; <<a2;
}


};
classB:public A //class B is publicly derived by class A
{
int b1, b2;
public:
void indata()
{
cout<<"\n Enter the value of b1 and b2";
cin>>b1>>b2;
}
void outdata()
{
cout<<"\n the value of b1 is" <<b1 <<"and b2 is:" <<b2;
}
};
classC; public B
```

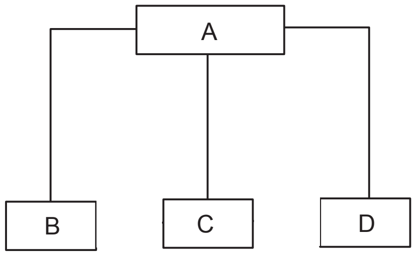
```
{
int c1, c2;
public:
void input()
{
cout<<"\n Enter the value of c1 and c2";
cin>>c1>>c2;
}
void output()
{
cout<<"\n the value of c1 is "<<c1<<" and c2 in:" <<c2;
}
};
void main()
{
C obj
obj.getdata();   //member function of class A
obj.indata();    //member function of class B
obj.input();     //member function of class C
obj.putdata();   //member function of class A
obj.outdata();   //member function of class B
obj.output();    //member function of class C
}
```

The following is the output of the above example:

```
Enter value of a1 and a2
3
4
Enter value of b2 and b2
6
7
Enter value of c1 and c2
8
9
```

```
the value of a1 is 3 and a2 is 4
the value of b1 is 6 and b2 is 7
the value of c1 is 8 and c2 is 9
```

9.4.4 Hierarchical inheritance: One class may be inherited by more than one classes. This process is known as hierarchical inheritance.



Hierarchical Inheritance

Example Program to illustrate Hierarchical Inheritance

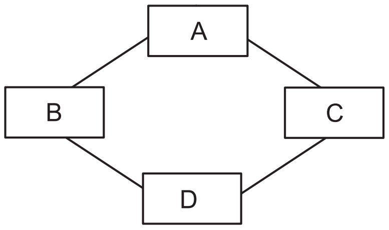
```
#include<iostream.h>
class A
{
int a, b;
public:
void getdata()
{
cout<<"\n Enter the value of a and b";
cin>>a>>b;
}
void putdata()
{
cout<<"\n The value of a is:"<<"a and b"<<b;
}
};
class B: public A
{
int c, d;
public:
void intdata()
```

```
{
cout<<"\n Enter the value of c and d";
cin>>c>>d;
}
void outdata()
{
cout<<"\n the value of c"<<"c and d is :" <<d;
}
};
class C: public A
{
int e, f;
public:
void input()
{
cout<<"\n Enter the value of e and f";
cin>>e >>f;
}
void output()
{
cout<<"\n the value of e is " <<"e and f is" <<f;
}
void main()
{
B obj1
C obj2;
obj1.getdata();          //member function of class A
obj1.indata();           //member function of class B
obj2.getdata();          //member function of class A
obj2.input();            //member function of class C
obj1.putdata();          //member function of class A
obj1.outdata();          //member function of class B
obj2.output();           //member function of class A
obj2.outdata();          //member function of class C
}
```

The following is the output of the above example

```
Enter the value of a and b
3
4
Enter the value of c and d
6
8
Enter the value of a and b
9
4
Enter the value of e and f
1
8
the value of a is 3 and b is 4
the value of c is 6 and d is 8
the value of a is 9 and b is 4
the value of e is 1 and f is 8
```

9.4.5 Hybrid inheritance: It is a combination of hierarchical and multiple inheritance.



Hybrid Inheritance

9.4.6 Making a private members Inheritable:

We have just seen how to increase the capabilities of an existing class without modifying it. The private members of a base class cannot inherited and therefore it is not available for the derived class directly. C++ provides a third visibility modifier, protected, which serves a limited purpose in inheritance. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it. It cannot be accessed by the functions outside these two classes. A class can now use all the three visibility modes as illustrated below;

```
class alpha {
private: //optional
---. //visible to member functions
```

---. //Within its class

protected;

---. //visible to member functions

---. //of its own and derived class

public;

---. //visible to all functions

---. //in the program

};

When a protected member is inherited in public mode. It becomes protected in the derived class too, and therefore is accessible by the member functions of the derived class. A protected member, inherited in the private mode derivation, becomes private in the derived class. Although it is available to the member functions of the derived class, it is not available for further inheritance (since private members cannot be inherited).

9.4.7 Visibility Modes

It can be public, private or protected. The private data of base class cannot be inherited.

(i) If inheritance is done in public mode, public members of the base class become the public members of derived class and protected members of base class become the protected members of derived class.

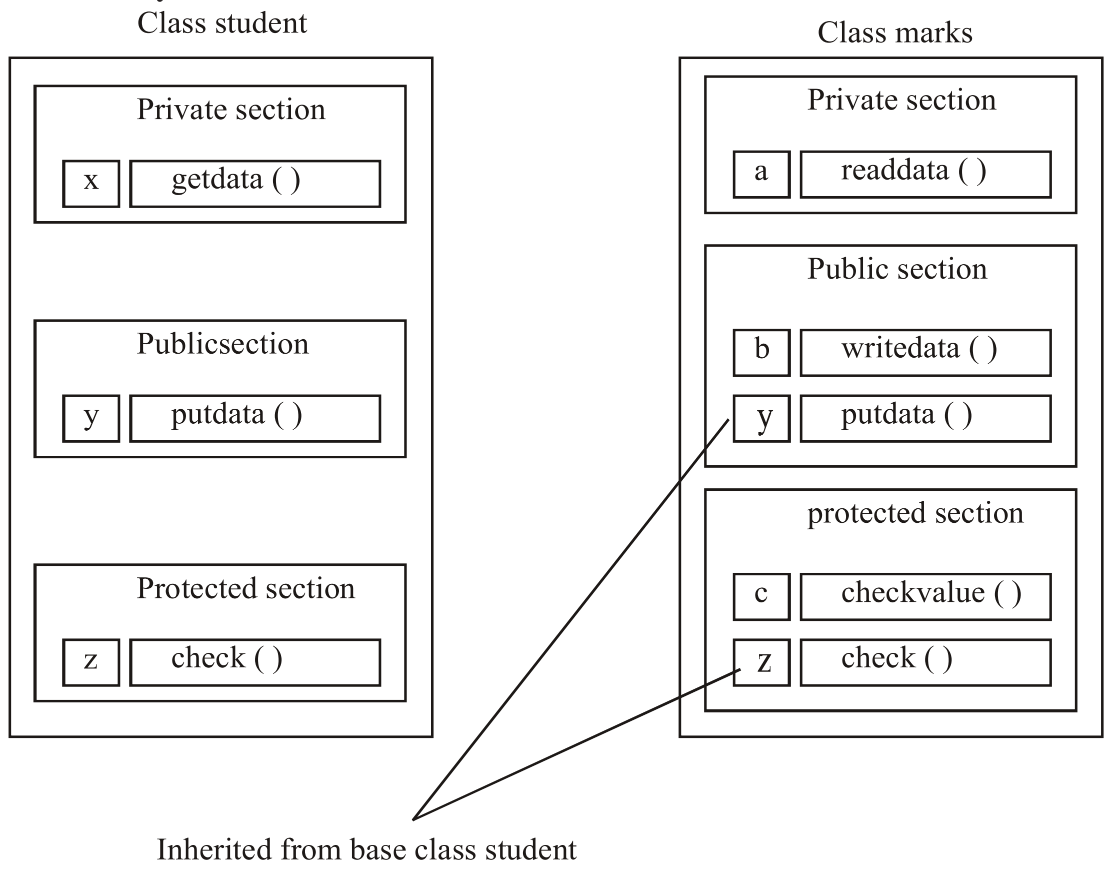
(ii) In inheritances is done in a private mode, public and protected members of base class become the private members of derived class.

(iii) If inheritance is done in a protected mode, public and protected members of base class become the protected members of derived class.

The following table shows the three types of inheritance:

Base class Access specifier	**Derived class**		
	private	private	protected
public	public	private	protected
private	Not inherited	Not inherited	Not inherited
protected	protected	protected	protected

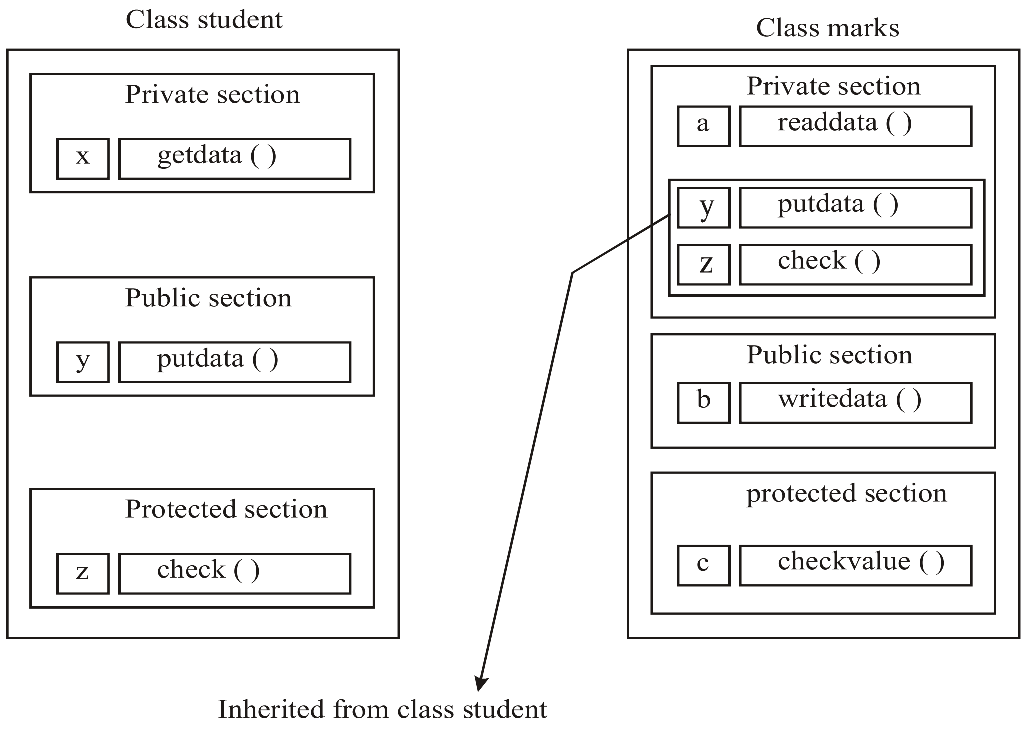
The Public Visibility Mode



Public derivation of class

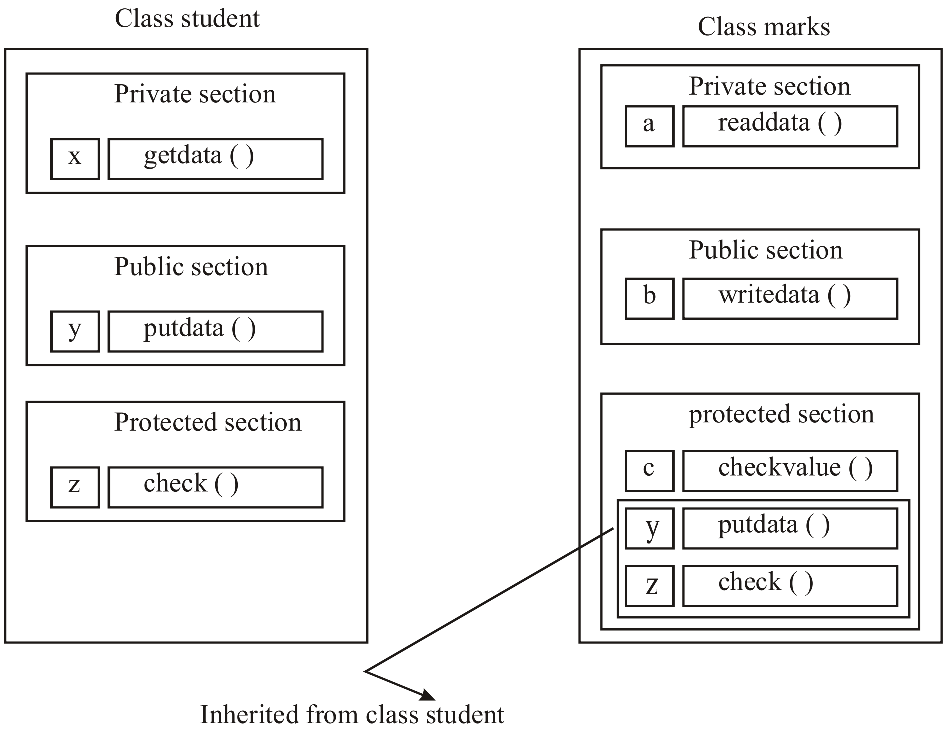
The public derivation does not change the access specifiers for inherited members in the derived class. The private data of base class student cannot be inherited.

The Private Visibility Mode



Private derivation of class

The Protected visibility mode



Protected derivation of class

9.5 Summary

Inheritance, the capability of one class to inherit properties from another class as a child inherits some properties from his/her parents. The most important advantage of inheritance is code reusability. A derived class is defined by specifying its relationship with the base class using visibility mode.The old class is referred to as the base class and new one is called the derived class. In Single inheritance a derived class with only one base class, In Multiple inheritance a derived class with several base classes inheritance,In Multilevel inheritance the mechanism of deriving a class from another derived class,in Hierarchical inheritance one class may be inherited by more than one classes.

9.6 Self Assessment Questions

1. What is inheritance?
2. What are the different forms of inheritance?
3. What are the mode of inheritance?
4. What are the differences between private and protected sections?
5. Fill in the blanks:

   (a) The base class is also called ....

   (b) The derived class can be derived from base class in ....or .... way.

   (c) When a derived class is derived from more than one base class then the inheritance is called .... inheritance.

4. State whether the following are True or False.

   (a) Inheritance means child receiving certain traits from parents.

   (b) The default base class is visible as public mode in derived class.

Unit-10 : Pointers, Virtual Functions and Polymorphism

Structure of Unit:

10.0 Objective

After reading this unit you would be able to understand polymorphism and pointer.

- Pointers can make programs quicker, straightforward and memory efficient.

- You will be able to operate various types of pointers like pointer to array, pointer to objects, pointer to structure and this pointer.

- You will learn something about polymorphism, function & operator overloading and virtual functions.

10.1 Introduction

Polymorphism is one of the crucial features of OOP. It simply means 'one name, multiple forms'. Polymorphism is devideed in two categories : Compile time and Runtime polymorphism.

Compile time polymorphism is achieved by function overloading or operator overloading. In function overloading , the same name functions perform different different task. In operator overloading, the operators perform on the user defined data-types just like the built-in datatypes.

Runtime polymorphism is achieved by virtual function. When the same name function in both base and derived class then the function of base, we made virtual. If a virtual function has no body or it is equated to zero then it is known as pure virtual function.

Pointer is one of the key aspects of C++ similar to C. Pointer is the derived data type that refers to another data variable by storing the variable's memory address rather than data. It store the address of normal variable rather than the value. We can perform the arithmetic operations on pointers. Pointers can be used in functions, structures, array etc. We can also define the array of pointers.

This pointer refers to an object that currently invokes a member fuction.It is used in constructor when there are same name variable in class and parameter of the parameterized constructor .

10.2 Polymorphism

Polymorphism is the important OOP concept. It refers to the ability to take more than one forms. "Poly" means "many" and "morph" means "forms" . It is the ability of an object (or reference) to assume (be replaced by) or become many different forms of object. Polymorphism gives different meanings or functions to the operators or methods. It refers to codes, operations or objects that behave differently in different contexts.

Below is the simple example of the above concept of polymorphism :

10 + 20 Integer addition

The same + operator can be used with different meanings with strings:

"Computer" + "Programming"

The same + operator can also be used for floating point addition :

8.20 + 6. 18

It means the single operator + behaves differently in different context such as integer, float or strings. This concept leads to operator overloading. In polymorphism, a single function or an operator functioning in many ways depends upon the usage to function properly. In order for this to occur, the following conditions must apply :

- All different classes must be derived from a single base class.
- The members function must be declared virtual in the base class.

The concept of polymorphism provides some advantages such as :

(1) Helps in reusability of code.

(2) Provides easier maintenance of applications.

(3) Helps in achieving robustness in applications.

10.3 Types of Polymorphism

(1) Compile time polymorphism/static binding/early binding.

(2) Run time polymorphism/dynamic binding

10.3.1 Complete time Polymorphism

It is achieved using operator and function overloading :

10.3.1.1 Operator Overloading

Operators overloading is very important feature of Object Oriented Programming. By using this facility programmer would be able to create new definitions to existing operators. In operator overloading we can overload both unary and binary operators.

The general syntax for defining an operator overloading is as follows :-

```
Returntype classname :: Operator operator-symbol (argument)
{
Statements ;
}
```

It clearly specify that operator overloading is defined as a member function by making use of the keyword operator.

In the above :-

Return type :- It is the datatype returned by the function.

Classname :- It is the name of the class.

Operator :- It is keyword.

Operator-Symbol :- It is the symbol of the operator which is being overloaded or defined for new functionality.

:: It is scope resolution operator which is used to use the function definition outside the class.

Unary Operator Overloading : As the name implies, unary operators operates on one operand. Some unary operators are ++, --, !, ~ .

For Example :- Program to illustrate the use of unary ++ operator.

```
#include<iostream.h>
#include<conio.h>
class unary
{
        int x;
        public:
                void getdata(int);
                void operator ++();
                void display();
};
void unary::getdata(int a)
{
        x=a;
}
void unary::operator++()
{
        x=x+1;
}
void unary::display()
{
cout<<x;
}
void main()
{
    unary obj;
    clrscr();
    obj.getdata(20);
   cout<<"The number is:";
    obj.display();
     obj++;
   cout<<"\n\nAfter overloading the ++ operator , number is : ";
    obj.display();
    getch();
}
```

The following is the output of above example

```
The number is: 20
After overloading the ++ operator , number is : 21
```

The Function operator ++ ( ) takes no argument because this function is a member function of class unary so it can directly access the member of object which activated it. This function increment the value of data member of object obj. It is possible to overload a unary ++ operator using a friend function.

The friend function is not the member function of the class so they are accessible directly without using object of the class and they can not access the members of object directly so we pass the object as an argument when we use friend function.

Binary Operator Overloading : Binary operators operates on two operators such as + , - , * , / etc.

Example program to illustrate the use of binary + operator.

```
#include<iostream.h>
#include<conio.h>
class complex
```

```
{
        int x,y;
        public:
                complex(){}
                complex(int real,int img)
                {
                x=real;
                y=img;
                }
                complex operator +(complex &obj);
                void display();
};
complex complex :: operator+(complex &c1)
{
        complex c2;
        c2.x=x+c1.x;
        c2.y=y+c1.y;
        return c2;
}
void complex::display()
{
cout<<x<<" + i"<<y;
cout<<"\n";
}
void main()
{
    clrscr();
    complex c1,c2,c3;
    c1=complex(20,30);
    c2=complex(10,10);
     c3=c1+c2;
   cout<<"\n\n********OUTPUT*********\n\n";
    cout<<"c1 = ";
    c1.display();
    cout<<"c2 = ";
    c2.display();
   cout<<"________________\n\n";
    cout<<"c3 = ";
    c3.display();
    getch();
}
```

The following is the output of the above Program.

```
*************OUTPUT**************
c1 = 20 + i30
c2 = 10 + i10
______________
c3 = 30 + i40
```

The function operator + takes one argument. In the operator + ( ) function, the data members of C1 are accessed directly and the data members of C2 (that is passed as an argument) are accessed using the dot operator. The result of addition of C1 and C2 is stored in res that is return to C3.

It is also possible to overload + operator using friend function as follows:-

```
Friend complex operator + ( Complex &C1 , Complex &C2);
Complex operator + (Complex &C1,Complex &C2)
{
        Complex C3;
        C3. x = C1.x + C2 . x;
        C3.y = C1.y + C2. y ;
        return C3;
}
```

Friend function takes two arguments when we overload the binary operators.

10.3.1.2 Function Overloading

We can use the same function name to create functions that perform different tasks. But the functions that share the same function name must follow at least one condition from the following conditions :-

(1) Either the function have different number of arguments.

(2) Either the functions have different datatypes from other same name functions.

(3) Either if the same name function have the same no of arguments then their datatype must bedifferent.

Example program is to illustrate function overloading

```
#include<iostream.h>
#include<conio.h>
class Fun_overload
{
        int a,b,a1,a3;
        float r,a2;
        public:
                void putdata();
                void area(int,int);
                void area(float);
};
void Fun_overload::area(int h,int w)
{
        a1=h*w;
}
void Fun_overload::area(float r)
{
        a2=3.14*r*r;
}
void Fun_overload::putdata()
{
cout<<"\n\narea of rectangle : "<<a1;
cout<<"\n\narea of circle : "<<a2;
}
void main()
{
        clrscr();
        Fun_overload obj;
        obj.area(10,5);
        obj.area(3.5);
        cout<<"************OUTPUT************";
        obj.putdata();
        getch();
}
```

The following is the output of above example :

```
************OUTPUT************
area of rectangle: 50
area of circle: 38.465
```

10.3.2 Runtime Polymorphism

It is also known as dynamic binding . It is implemented through the virtual function.

10.3.2.1 Virtual function

Virtual , as the name implies, is something that exist in effect but not in reality. The object oriented programming like C ++ implements the concept of virtual function as a simple member function like all member function of the class. The functionality of virtual functions can be overridden in its derived classes. Virtual function is a mechanism to implement the concept of polymorphism.

Need for virtual function

The reason for having a virtual function is to implement a different functionality in the derived class.

Properties of virtual function :

- Virtual function are resolved during run time or dynamic binding.
- They cannot be static members
- Virtual functions are member function of a class.
- Virtual functions are declared with the keyword virtual.
- They are accessed by using object pointer.
- Virtual function takes a different functionality in the derived class.

Declaration of Virtual Function

```
class Class_Name
{
public : virtual void member- function name ( )
        {
        }
}
```

10.3.2.2 Pure Virtual Function

A pure virtual function has no body. The programmer must add the notation = 0 for declaration of the pure virtual function in the base class.

```
Syntax :- class class_Name
                  {
                  public :
                          virtual void virtual_function_name ( )= 0;
                  }
```

Pure virtual function in the base class has been defined 'empty'. Such functions are called "do-nothing" or 0 function.

A class containing pure virtual function cannot be used to declare any objects of its own. Such classes are called abstract base classes.

```
class Base
{
        public : virtual void show ( ) = 0;
};
class derived 1 : public Base
{
public : void show ( )
                {
                  cout <<"In Derived1";
                }
};
  class derived 2 : public Base
{
public : void show ( )
        {
                cout <<"In Derived2";
        }
};
 void main ( )
  {
        Base * b ;
        Derived1 d1;
        Derived2 d2;
        b= &d1 ;
        b. show ( );
        b= &d2;
        b.show ( );
```

A pure virtual function is useful when we have a function that to put in the base class, but only the derived classes know what it should return. A pure virtual makes it so the base class can not be instantiated , and the derived classes are forced to define these function before they can be instantiated. This helps ensure the derived classes do not forget to redefine functions that the base class was expecting them to.

10.4 Pointers

The use of pointer offers a high degree of flexibility in the management of data. Knowledge of memory organization plays very important role for understanding the concept of pointers. Pointer are derived data and are widely used in programming. They are used to refer to memory location of another variable without using variable identifier itself. They are mainly used in linked list and call by reference functions.

	100	1
	101	2
	102	3
YPTR 100	103	4
	104	5
*YPTR 1	105	6

Declaring pointers can be very confusing and difficult at times . To declare pointer variable we need to use *operators (indirection/dereferencing operator) before the variable identifier and after data type pointer can only point to variable of the same data type.

Syntax : Datatype * identifier ;

```
int * ptr;  // ptr is a pointer to integer
```

Example:

```
# include < iostream.h>
 main ( )
{
        char a = 'b' ;
        char * Ptr ;
        cout << a ;
        Ptr = & a ;
        cout <<ptr;
        *Ptr = 'd';
        cout << a;
}
```

Address Operator :- Address operator (&) is used to get the address of the operand. For ex. if variable x is stored at location 100 of memory; & x will return 100. This operator is used to assign value to the pointer variable. It is unary operator.

Indirection Operator :- The * is know as Indirection operator . It returns the contents of the memory location pointed. The indirection operator is also called deference operator.

For example : int * x ;

```
int C = 100;
int p;
```

```
x = &C;
p= *x;
```

Variable x is the pointer to integer type. It points to the address of the location of the variable C. The pointer x will contain the contents of the memory location of variable C. It will contain value 100. When statement.

```
p = *x ;
```

is executed '*' operator returns the content of the pointer x and variable p will contain value 100 as its memory location.

Example program to illustrate the use of pointer.

```
#include<iostream.h>
#include<conio.h>
class Pointer_intro
{
        int x,*y,z;
        public: void getdata(int a)
                {
                  x=a;
                }
                void operation()
                {
                  y=&x;
                  z=*y;
                  cout<<"\t\t***************OUTPUT***************";
                  cout<<"\n\nvalue of x : "<<x;
                  cout<<"\nvalue of y=&x : "<<y;
                  cout<<"\nvalue of z=*y : "<<z;
                }
};
void main()
{
        Pointer_intro pobj;
        clrscr();
        pobj.getdata(10);
        pobj.operation();
        getch();
}
```

The following is the output of above program:

```
***************OUTPUT****************
value of x : 10
value of y=&x : 0x8b8fff0
value of z=xy : 10
```

Pointer Arithmetic :

There are only two operations that can be performed on pointers such as addition and subtraction. The integer value can be added or subtracted from the pointer. When a pointer is incremented it points to the memory location of the next element of its base type and when it is decremented it points to the memory location of the previous elements of the same base type. For example :

```
int * x ;
int * p ;
x ++;
```

here x and p are pointers to integer type. Pointer x is increments by 1. Now variable p points to the memory location next to the memory location of the pointer x. Suppose memory address of x is 2000 and as a result x will contain memory address 2002 because integer type takes 2 bytes so memory address is increment by 2.

Complete Program to illustrate pointer Arithmetic:

```
#include<iostream.h>
#include<conio.h>
void main()
{
        int *x,*y,*z;
        int a,b;
        clrscr();
        a=100;
        x=&a;
        y=x+2;
        z=x-2;
        cout<<"\t   ***********OUTPUT**********\n\n";
        cout<<"\nvalue of a="<<a;
        cout<<"\nvalue of *x="<<*x;
        cout<<"\nafter incrementing the value of x by 2="<<y;
        cout<<"\nafter decrementing the value of x by 2="<<z;
        getch();
}
```

The following is the output of above program:

```
***************OUTPUT***************
value of a= 100
value of *x= 100
after incrementing the value of x by 2= 0x8b8fff4
after decrementing the value of x by 2= 0x8b8ffec
```

10.5 Types of Pointers

There are following types of pointers used in C++.

10.5.1 Pointer and Array

The concept of array is very much bound to the one of pointer. The identifier of an array is equivalent to the address of its first element, as a pointer is equivalent to the address of first element that it points to for example, supposing these two declaration

```
int num [20]
int * p;
```

The following assignment operation would be valid ;

```
p = num ;
```

After that p and num would be equivalent and would have the same properties. The only difference is that we could change the value of pointer p by one, where as num will always point to the first of the 20 elements of type int with which it was defined. Therefore, unlike p, which is an ordinary pointer, num is an array and array can be considered as a constant pointer. Therefore , the following allocation would not be valid.

```
num = p ;
```

Because num is an array, so it operates as a constant pointer and we cannot assign values to constants.

Example Program to illustrate the Pointer and Array.

```
#include<iostream.h>
#include<conio.h>
void main()
{
        int num[5],sum=0,i;
        int *p;
        clrscr();
        cout<<"**********OUTPUT*********\n\n";
        cout<<"enter five numbers\n";
        for(i=0;i<5;i++)
        {
                cin>>*p;
                num[i]=*p;
                sum=sum+(*p);
                p++;
        }
        cout<<"\nSum = "<<sum;
        cout<<"\nlast number = "<<num[4];
        getch();
}
```

The following is the output of above program

```
***************OUTPUT****************
enter five numbers
11
12
13
14
15
Sum = 65
last number = 15
```

10.5.2 Pointers to pointers

C++ allows the use of pointers that point to other pointers , that means point to data. In order to do that, we only need to add an asterisk (*) for each level of reference in their declaration :

```
char a ;
char *b;
char **c;          // pointer to pointer
a = 'z';
b = &a ;
c= &b ;
```

This , supposing the randomly chosen memory locations for each variable of 7320 , 8092 and 10502 , could be represented as :

A	b	c
'z'	7230	8092
7230	8092	10502

The value of each variable is written inside each cell, under the cells are their respective addresses in memory.

In previous example :-

- c has type char ** and a value of 8092.
- * c has type char * and a value of 7230
- **c has type char and a value of 'z'

10.5.3 Pointers to functions

C++ allows operations with pointers to functions. It is a very useful feature of C++. It order to declare a pointer to a function we have to declare it like the prototype of the function except that the name of the function is enclosed between parentheses ( ) and an asterisk (*) is inserted before the name. Its syntax

```
        data type (* function name) (argument list);
        int(*func) (int a, int b); // pointer to function returning int
        int *func(int x, int y); // WRONG, for pointer to function
```

Example program to illustrate the use of pointer to function.

```
#include<iostream.h>
#include<conio.h>
void func (int);              // Function Prototype
void main()
{
void (*fp)(int);
clrs(rc);
        fp = func;
        (*fp)(10);
        fp(20);
}
void func (int arg)
{
        cout << "In value" << arg;
}
```

The following is the output of the above example.

```
Value : 10
Value : 20
```

10.5.4 Pointers to Structure

Like any other type, structures can be pointed by its own type of pointers ;

```
struct movies_it
{
          string title ;
                    int year ;
                    }
                    movies_it amovie ;
                    movies_it * bmovie ;      // bmovie pointer to structure
```

Here amovie is an object of structure type movies_it, and b movie is a pointer to point to objects of structure type movies_it. So, the following code would also be valid

```
                    bmovie = & amovie ;
```

/**********************POINTER AND STRUCTURE***********************/

Example Program to illustrate the use of pointer and structure.

```
#include<iostream.h>
#include<conio.h>
struct Employee
{
          char name[30];
          int age, salary;
}e1,*e2;
void main()
{
          clrscr();
          cout<<"**************OUTPUT***************\n\n";
          cout<<"Enter name of the first employee" ;
          cin>>e1.name;
          cout<<"\nEnter age of the first employee";
          cin>>e1.age;
          cout<<"\nEnter salary of the first employee";
          cin>>e1.salary;
          cout<<"\nEnter age of the second employee";
```

```
        cin>>e2->age;
        cout<<"\nEnter salary of the second employee";
        cin>>e2->salary;
        cout<<"\n\n\t\tInformation of I employee";
        cout<<"\nname : "<<e1.name;
        cout<<"\nage : "<<e1.age;
        cout<<"\nsalary : "<<e1.salary;
        cout<<"\n\n\t\tInformation of II employee";
        cout<<"\nage : "<<e2->age;
        cout<<"\nsalary : "<<e2->salary;
        getch();
}
```

The following is the output of the above example

```
***************OUTPUT***************
Enter name of the first employee
amit
Enter age of the first employee
22
Enter salary of the first employee
20000
Enter age of the second employee
25
Enter salary of the second employee
15000
Information of I employee
name : amit
age : 20
salary : 20000
Information of II employee
age : 25
salary : 15000
```

10.5.5 Pointers to Object

It is perfectly valid to create pointers that point to objects of classes. We simply have to consider that once declared, a class becomes a valid type, so we can use the class name as the type for the pointer.

For example :

```
CRectangle * prect ;
// is a pointer to an object of class CRectangle .
```

As it happened with data structures, in order to refer directly to a member of an object pointed by a pointer we can use the arrow operator (→) of the indirection.

Next we have a summary on how can we read some pointer and class operators (* , & . , []) that are

Expression	Can be read as
* x	pointed by x
& x	address of x
x. y	member y of object x
x y	member y of object pointed by x
(*x) . y	member y of object pointed by x
X[0]	1st object pointed by x
X[1]	2nd object pointed by x
X[n]	(n+1)th object pointed by x

Example Program to illustrate the use of pointer to object

```
#include<iostream.h>
#include<conio.h>
class Pointer_object
{
        int a,b;
        public: void getdata(int x,int y)
                {
                  a=x;
                  b=y;
                }
                void display()
                {
                  cout<<"\n\ta = "<<a;
                  cout<<"\n\tb = "<<b;
                }
};
void main()
```

```
{
        Pointer_object obj1,*obj2;
        clrscr();
        obj1.getdata(10,20);
        cout<<"********OUTPUT********\n\n";
        cout<<"values of normal object";
        obj1.display();
        obj2->getdata(20,30);
        cout<<"\n\nvalues of pointer object";
        obj2->display();
        getch();
}
```

The following is the output of above example

```
***************OUTPUT***************
values of normal object
a = 10
b = 20
values of pointer object
a = 20
b = 30
```

10.6 This Pointer

When a member function is called, how does C++ know which object it was called on ? The answer is that C++ utilizes a hidden pointer named is "this". C++ uses a unique keyword called "this" to represent an object that invokes a member function. The pointer this acts as an implicit argument to all the member functions. The following is a simple class that holds an integer and provide a constructor and access functions.

Example program to illustrate the use of "this" pointer.

```
#include<iostream.h>
#include<conio.h>
class This_Pointer
{
        int a,b;
        public: This_Pointer(int a,int b)
                {
```

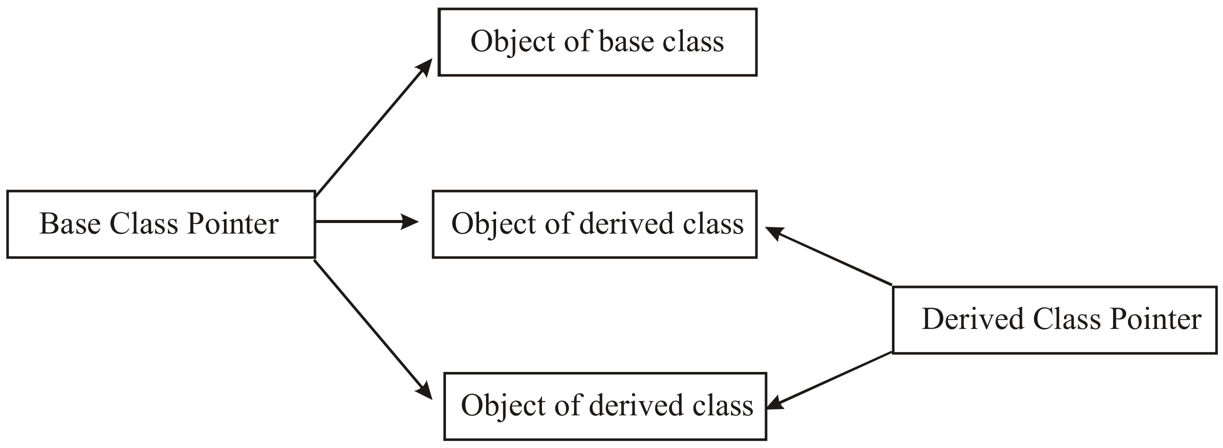
```
                this->a=a;
                this->b=b;
            }
            void display()
            {
                cout<<"**********OUTPUT**********";
                cout<<"\n\nValue of a = "<<a;
                cout<<"\nValue of b = "<<b;
            }
};
void main()
{
        This_Pointer obj(20,50);
        clrscr();
        obj.display();
        getch();
}
```

The following is the outout of the above program.

```
***************OUTPUT***************
value of a = 20
value of b = 50
```

10.7 Pointers to Derived Classes

We can use pointers not only to the base objects but also to the objects of derived classes. The pointers to objects of a base class are type-compatible with points to objects of a derived class & single pointer variable can be made to point to objects belonging to different classes.



Example program to illustrated the use of pointer to base class and derived class objects.

```
# include <iostream.h>
# include <iostream.h>
class father
{
        protected:
                int f_ age;
Public:
        father (int n)
        {
        f_age = n;
        }
        int Get Age (void)
        {
        return f_age;
        {
};
class son: public father
{
        protected:
                int s_age;
        public:
                son(int n, int m): father (n)
        {
                s_age = m;
        }
        int GetAge (word)
                {
                return s_age;
                }
        void son func ( )
                {
                cout << “son’s own function”;
                }
        };
void main ( )
{
        father * basep;
        basep = new father (40);                // pointer to father
        cout << "basep points to base object.... " <<endl;
        cout << " Father's Age : ";
        cout << basep → GetAge ( ) <<endl;  // calls Father :: Get Age
        delete basep;
        // accessing derived object
        basep = new Son (40, 20);       // pointer to son
        cout << "basep points to derived object ............ "  << endl;
        cout << "son's Age:";
        cout << basep –> GetAge( ) << endl;             // calls father : GetAge ( )
        cout << "By type casting, ((son*) basep)... " << endl; cout << "son's Age:";
        cout << ((son*) basep) –> GetAge ( ) << endl; // calls son :: GetAge( )
        delete basep;
        //accessing with derived object pointer
        son son1 (40,20);
        son * derivedp = & son1;
        cout << "accessing through derived class pointer....." << endl;
        cout << "son's age: ";
        cout << derivedp –> GetAge ( );
}
```

The following is output of the above example

```
basep points to base object.....
father's Age : 40
basep points to derived object........
sons Age : 40
By typecasting, ((son*)basep)......
son's age : 20
accessing through derived class pointer......
son's Age : 20
```

10.8 Summary

Runtime polymorphism is achieved only when a virtual function is accessed through a pointer to the base class. It cannot be achieved using object name along with the dot operator to access virtual function.When a function is made virtual, C++ determines which function to use at runtime based on the type of object pointed to by the base pointer, rather than the type of the pointer. By making the base pointer to point to different objects, we can execute different versions of the virtual function.

We can have virtual destructor but not virtual constructors. If a virtual function defined in the base class, It need not be necessarily redefined in the derived class. A virtual function equated to zero is known as pure virtual function. It is a function declared in a base class that has no definition relative to the base class. A class containing such pure function is called an abstract class.

A this pointer refers to an object that currently invokes a member function. Pointers to object of a base class type are compatible with pointers to objects of derived class. Therefore we can use a single pointer variable to point to objects of base class as well as derived classes.

10.9 Self Assessment Questions

Q.1 What is Polymorphism?

Q.2 What are difference between Compile Time Polymorphism and Runtime Polymorphism?

Q.3 What is the need of operator overloading?

Q.4 How to implement function overloading?

Q.5 Explain runtime polymorphism. What is the difference between virtual function and pure virtual function?

Q.6 Why we use pointers? Explain the concept of array of pointers.

Q.7 What is difference between pointer and array?

Q.8 How to pass pointer arguments in function?

Q.9 What is this pointer?

Q.10 How to implement Pointer object?

Unit-11: Arrays

Structure of Unit:

11.0 Objective

This chapter provides a basic idea of arrays. You will get knowledge about array, its types and initialization. This chapter discusses how to define array within a class, memory allocation for objects, array of objects and how to create member functions that works with the class.

11.1 Introduction of Array

An array is a collection of elements of the same type that are referred to through a common name. A specific element in an array is accessed by an index. In C/C++, all arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element. Arrays may have from one to several dimensions. The most common array is the null-terminated string, which is simply an array of characters terminated by a null.

Arrays are a way to group a number of items into a large unit. Arrays can have data items of simple type like int or float, even of user defined types like structures and objects.

Arrays are of different types :

(i) one-dimensional arrays : comprised of finite homogeneous elements.

(ii) multi-dimensional arrays : comprised of elements, each of which is itself an array.

A two dimensional array is the simplest of multidimensional array.

Single Dimensional Array

The simplest form of an array is single dimensional array. The array is given a name and its elements are referred to by their subscripts or indices. C++ array's index numbering starts with 0. An array definition specifies a valid C++ type and a name (avoid C++ identifier) along with one or more feature, size to specify how many data items the array will contain. The general form of an array declaration is as shown below :

type array-name [size];

where type declares the base type of the array, which is the type of each element in the array. The array-name specifies the name with which the array will be referenced and size defines how many elements the array will hold. The size must be an integer value or integer constant without any sign.

NOTE:- The data type of array elements is known as the base-type of the array.

Following statement declares an array marks of base type int and which can hold 50 elements.

```
int marks [50];
```

The above statement declared array has 50 elements, marks[0] to marks[49].

Example program reads marks of 50 students and stores them under an array.

```
# include<iostream.h>
# include<conio.h>                    // for clrscr()
int main()
{
        clrscr();
        float marks[50];
                // loop to read 50 marks
        for(int i=0; i<50; i++)
        {
                cout << "Enter the marks of"<< i << "th student";
                cin>>marks[i];
        }
                cout << "\n";
                // loop to print the elements of array.
        for(ii=0; i<50; i++)
                cout << "Marks["<< i <<"] = "<< marks[i] << "\n";
return 0;
}
```

A vector is a mathematical term which refers to the collection of numbers which are analogous i.e., a linear array (one dimensional array). Thus, a vector in C++ can represent only integers and floating-point numbers.

String as an Array

C++ does not have a data type rather it implements string as single-dimension character arrays. A string is defined as a character array that is terminated by a null character '\0'. For this reason, the character array declared one character longer that the largest string they can hold. For instance, to declare an array strg that holds a 10-character string, you would write :

```
char strg[11];
```

This makes room for the null character at the end of the string.

Individual characters of a string can be easily accessed as they make the elements of the character array. The index 0 refers to the first character, the index 1 to the second, 2 to the third, and so on. The end of a string is determined by checking for null character.

Example program checks whether a string is palindrome or not.

```
# include<iostream.h>
# include<conio.h>                         // for clrscr()

int main()
{
clrscr();
char string[80], c;
int i, j, flag = 1;
cout << "Enter a string (max. 79 characters) \n";
cin.getline(string, 80);
                                    // loop to find length of the string.
        for(int len = 0; string[len] !='\0'; len++ )

for (i = 0, j = len - 1; i < len/2; i++, j--)
{
        if(string[i] != string[j])
        {
                flag = 0;
                break;
        }
}                            // loop over
        if (flag)
                cout << "It is a palindrome \n";
        else
                cout << "It is not a  palindrome \n";
return 0;
}
```

The following is the output of the above program

```
Enter a string (max.79 characters)
Madam
It is a palindrome
Enter a string (max. 79 characters)
```

Education

It is not a palindrome

The above program compares characters form both end of a string; if any character is different, string is reported not to be a palindrome.

Two Dimensional Array

A two dimensional array is an array in which each element is itself an array. For instance, an array A [M][N] is an M by N table with M rows and N columns containing M x N elements .

The number of elements in a 2-D array can be determined by multiplying number of rows with number of columns. For example, the number of elements in an array A[7][9] is calculated as 7x9 = 63.

The simplest form of a multidimensional array, the two-dimensional array, is an array having single-dimension array as its elements. The general form of a two dimensional array declaration in C++ is as follows :

type array-name [rows][columns];

where type is the base data type of the array having the name array-name; rows, the first index refers to the number of rows in the array and columns, the second index, refers to the number of columns in the array. Following declaration declares an int array sales of size 5, 12.

int sales [5][12];

The array sales have 5 elements sales[0], sales[1], sales[2], sales[3], and sales[4] each of which is itself an int array with 12 elements. The elements of sales are referred to as sales[0][0], sales[0][1], ….., sales[0][11], sales[1][0], sales[1][1], …… and so forth.

Example program reads sales of 5 salesmen in 12 months.

```
# include<iostream.h>
# include<conio.h>              // for clrscr()

int main()
{
        clrscr();
        int i, j, total;
        int sales[5][12];
        for(i=0, i<5; i++)                  // loop to read sales
        {
                total = 0;                  // initialize for each salesman
                cout << "Enter sales for salesman " << i+1 << ": \n";
        for(j=0, j<12; j++)                 // loop for monthly sales reading
        {
                cout << "\n Months " << j+1 << ":";
                cin >> sales[i][j];              // read monthly sales
                total = total + sales[i][j];     // calculate total
        }                           // end of for j
        cout << "\n";
        cout << "Total sales of salesman is :" << i+1 << "=" << total << "\n";
        }                                   // end of for i
return 0;
}
```

The above program reads for each statement the monthly sales and simultaneously calculates total sales made by the salesman. but before reading sales for another salesman, it first prints total sales made by the salesman whose sales have been recently read.

Array Initialization

The general form of array initialization is as shown below :

type array-name[size N] = {value list };

The value-list is a comma separated list of array elements' values. The element values in the value-list must have the same data type as that of type, the base type of the array.

Following code initializes an integer array with 12 elements.

int days-of-month[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}:

This would place first value 31 in month [0], 28 in month [1], and so on.

Character arrays can also be initialized like this as shown below :

char string[10] = "Program";

The above code will initialize the string string with "Program".

Alternatively, above declaration can be written as :

```
char string[10] = {'P', 'r', 'o', 'g', 'r', 'a', 'm', '\0'};
```

Because all string in C++ terminates with a null, you must make sure that the array you declare is long enough to include the null.Two-dimensional arrays are also initialized in the same way as single-dimension ones. For example, the following code initializes a two-dimensional array cube with numbers 1 through 5 and their cubes :

```
int cube[5][2] = {1, 1,
                  2, 8,
                  3, 27,
                  4, 64,
                  5, 125};
```

We have put the values of different rows in different lines for the sake of readability. All the values can appear in the same line also as shown below :

int cube[5][2] = {1, 1, 2, 8, 3, 27, 4, 64, 5, 125};

The element cube[0][0] will be having value1, cube[0][1] value 1, cube[1][0] value 2, cube[1][1] value 8, cube[[2][0] value 3, and so on.

Array of string can be initialized as it is given below :

```
char star[7][11] = {"Sunday", "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday"};
```

Sometimes you do not know all the elements, still you want to initialize the array. For example,

```
int sal[4][2] = {{1}, {2, 6}, {3}, {4, 8}};
```

According to the above code, sal[0][0] will be 1, sal[1][0] = 2, sal[1][1] = 6, sal[2][2] = 3, sal[3][0] = 4, sal[3][1] = 8.

Note that in such cases, elements of each row are enclosed separately in curly braces {}.

Unsized Array Initialization

C++ allows you to skip the size of the array in an array initialization statement. This is called an unsized array. Unsized array initialization are not restricted to one dimensional arrays. For multidimensional arrays, you must specify all but the leftmost dimension (The other dimensions are needed to allow the compiler to index the array properly). C++ automatically calculate the dimensions of unsized array. Following are some examples of unsized array initializations :

```
char S1[ ] = "First String";
int val[ ] = {2, 3, 4, 5, 6, 7, 8, 9};
float amount[ ] = {2341.57, 1800.70, 4456.78, 6657.89, 3435.00};
int cube[ ][2] = {1, 1,
                  2, 8,
                  3, 27,
                  4, 64,
                  5, 125};
```

The advantage of this declaration is that you may lengthen or shorten the value-list without changing the array dimensions.

11.2 Arrays Within a Class

A class can have an array as its class member variable. An array in a class can be private or public member of the class. If an array happens to be a private data member of the class, then, only member functions of the class can access it. On the other hand, if an array is a public data member of the class, it can be accessed directly using objects of this type. For instance consider following class definition :

```
class Exarray    {
int arr[10];              // private data member
public :
        int largest(void);
        int sum(void);
};
```

The variable arr[10] is a private data member (as it is declared under no label, and thus, by default is private) of class Exarray. It can be accessed only through the member functions largest() & sum() like any other variable. Any operations can be performed on the array Exarray, but (not directly by using objects) only through the member functions. Let us consider following example program that stores price list of 50 items in array and can print the largest price and the sum of all the item prices.

Example program using a class to store price list of 50 items and to print the largest price as well as the sum of all prices.

```
# include <iostream.h>
# include <conio.h>              // for clrscr()
class ITEM      {
        int itemcode[50];
        float it_price[50];
public:
        void initialize (void);
        float largest(void);
        float sum(void);
        void display_items(void);
        };
        //……Member function definition follow……
Void ITEM : : initialize(void)
{
        for(int i=0; i<50; i++)
        {
                cout << "\n" << "Item No :" <<(i+1);
                cout << "\n" << "Enter item code :" ;
                cin >> itemcode[i];
                cout << "\n" << "Enter item price :" ;
                cin >> it_price[i];
                cout << "\n";
        }
}
float ITEM :: largest(void)
{
        float large = it_price[0];
        for(int i=1; i<50; i++)
        {
                if(large < it_price[i])
                large = it_price[i];
        }
        return large;
}
float ITEM :: sum(void)
{
        float sum = 0;
        for(int i=0; i<50; i++)
        {
                if(large < it_price[i])
                sum += it_price[i];
        }
        return sum;
}
void ITEM :: display_items(void)
{
        cout << "\nCode Price\n";
        for(int i=0; i<50; i++)
        {
                cout << "\n" << itemcode[i];
                cout << "          " << it_price[i];
        }
        cout << "\n";
        return;
}

int main()
{
        ITEM order;
        order.initialize();                    // initialize array
        float total, biggest;
        int ch = 0;
```

```
clrscr();
do
{
          cout << "\nMain Menu.\n";
          cout << "\n1.Display largest price.";
          cout << "\n2.Display sum of prices.";
          cout << "\n3.Display item list";
          cout << "\nEnter your choice (1-3) :";
          cin >> ch;
switch(ch)
{
          case1 : biggest = order.largest();
                    cout << "The Largest Price is " << biggest << "\n";
                    break;
          case2 : total = order.sum();
                    cout << "The Sum of Prices is " << total << "\n";
                    break;
          case 3: order.display_items();
                    break;
          default: cout << "\nWrong Choice!\n";
                    break;
          }                    // end of switch
}while(ch>=1 && ch<=3);
return 0;
}
```

The above program first declares a class ITEM having two arrays as its private data members. Since we are familiar with class specification, we'll discus how the main() function uses thus class.The function main() first declares an object order of class ITEM (see first statement of main()). Then the function main() initializes both the arrays by invoking the function initialize(). Observe that the function initialize() is invoked through the object of this class type. See the statement order.initialize(). Then main() displays a menu and asks for user's choice. Depending upon user's choice, corresponding function is invoked through the object order and the desired information id displayed.

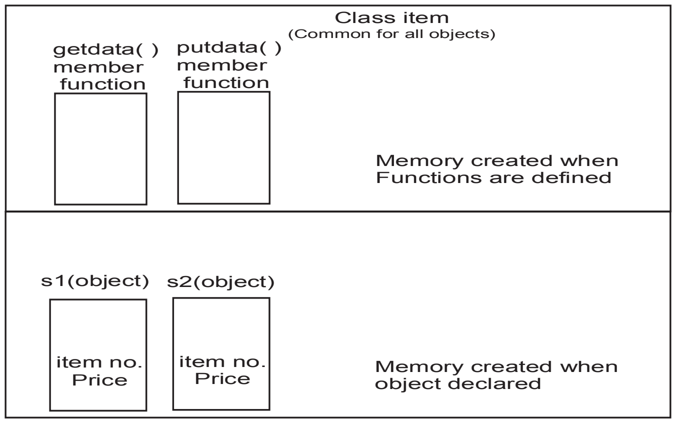
After the above program, now two things are clear. First that a class can have arrays as its member and second the public member functions are used like ordinary function but along with object name and a (.) dot. After this program, you can write simple C++ program using classes.

11.3 Memory Allocation for Objects

We have just stated before that the memory space for objects is allocated when they declared and not when the class is defined. We modify the statement now :

Member functions are created and placed in the memory space only once when the class is defined. The memory space is allocated for objects' data member onlywhen the objects are declared. No separate space is allocated for member functions when the objects are created.

Since all the objects belonging to a class use the same member functions, there is no point allocating separate space for member functions for each object. Thus member functions are created and stored when the class is defined and this memory space (containing member functions) can be read by any of the objects of that class. Separate memory space is allocated to objects at the time of their declaration for their data member only, because the data members hold different values for different objects. Following figure shows the memory allocation for class item and its objects s1 and s2 defined above.


Class item
(Common for all objects)
getdata( ) member function
putdata( ) member function
Memory created when Functions are defined
s1(object)
s2(object)
item no. Price
item no. Price
Memory created when object declared

The class item, its member functions and objects in memory

11.4 Static Data Members

A data member of a class can be qualified as static just like C static variable. A static data member of a class is just like a global variable for its class. That is, this data member is globally available for all the objects of that class type. The static data members are usually maintained to store values common to the entire class. For instance, a class may have a static data member keeping track of its number of existing objects.

A static data member is different from ordinary data members of a class in various respects:

1. There is only one copy of this data member maintained for the entire class which shared by all the objects of that class.

2. It is visible only within the class, however, its lifetime (the time for which it remains in the memory) is the entire program.

3. It is initialized to zero when the first object of its class is created.

Two things are needed for making a data member static :

(i) Declaration within the class definition.

(ii) Definition outside the class definition.

The declaration of a static data member within the class definition, is similar to any other variable declaration except that it start with the keyword static as shown below :

```
class X {     static int count ; // declaration within the class
              :
                   :
              };
```

The definition of above declared static member count outside the class will be as follows :

```
int X :: count ;
```

Therefore, the class definition would look like as :

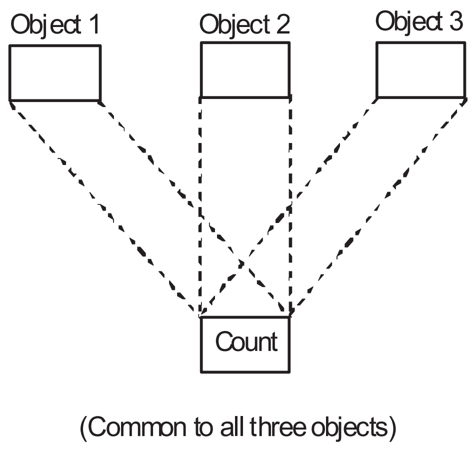
```
class X {     static int count ;
                   :
                   :
              };
int X :: count ;                    // definition of static data member
```

A static data member must be defined outside the class definition as these are stored separately rather than as a part of an object.Since, they are associated with the class itself rather than with any class object, they are also known as class variable. Following figure shows a static data member is shared by all objects.

A static data member can be given as initial value at time of its definition as shown below :

```
int X :: count = 10;
```

The above definition initializes count with value 10.



Sharing of a Static Data Member

When you precede a member variable's declaration with static, you are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable. Unlike regular data members, individual copies of static member variable are not made for each object. No matter how many objects of a class are created, only one copy of a static data member exists. Thus all objects of that class use that same variable. All static variables are initialized to zero before the first object is created.

When you declare a static data member within a class, you are not defining it. (That is, you are not allocating storage for it). Instead, you must provide a global definition for it elsewhere, outside the class. This is done by redeclaring the static variable using the scope resolution operator to identify the class to which it belongs. This causes storage for the variable to be allocated.

Example program to illustrate the use of static data member :

```
# include <iostreams.h>
using namespace std;
class shared     {
                 static int a;
                 int b;
public :
                 void set(int i, int j)       {a=i; b=j;}
                 void show();
} ;
int shared :: a;          // define a
void shared :: show()
{
                 cout << " This is static a: " << a;
                 cout << " \nThis is non-static b: " << b;
                 cout << " \n ";
}
int main()
{
         shared x, y;
         x.set(1, 1);            // set a to 1
         x.show();
         y.set(2, 2);            // set a to 2
         y.show();
         x.show();               /* here, a has been changed for both x and y
                                 because a is shared by both objects. */
         return 0;
}
```

The following is the output of above program.

```
This is static a: 1
This is non-static b: 1

This is static a: 2

This is non-static b: 2

This is static a: 2

This is non-static b: 1
```

Notice that the integer a is declared both inside shared and outside of it. As mentioned earlier, this is necessary because the declaration of a inside shared does not allocate storage.

A static member variable exists before any object of its class is created. For example, in the following short program, a is both public and static. Thus it may be directly accessed in main(). Further, since a exist before an object of shared is created, a can be given a value at any time. As this program illustrates, the value of a is unchanged by the creation of object x. For this reason, both output statements display the same value: 99.

```
# include <iostream.h>

using namespace std;

class shared      {

public;

                  static int a;

} ;

int shared :: a;              // define a

int main()

{

        //initialize a before creating any objects

        shared::a = 99;

        cout << "This is initial value of a: " << shared::a;

        cout << " \n ";

        shared x;

        cout << " This is x.a: " << x.a;

        return 0;

}
```

Notice how a is referred to through the use of the class name and the scope resolution operator. In general, to refer to a static member independently of an object, you must qualify it by using the name of the class of which it is a member.Another interesting use of a static member variable is to keep track of the number of objects of a particular class type that are in existence. For example :

```
# include<iostream.h>

using namespace std;

class Counter
```

```
{
        public :
                static int count;
                Counter() { count++; }
                ~Counter() { count--; }
};
int Counter :: count ;
void f();
int main(void)
{
                Counter o1;
                cout << "Objects in existence :";
                cout << Counter :: count << "\n";
                Counter o2;
                cout << "Objects in existence :";
                cout << Counter :: count << "\n";
                f();
                cout << "Objects in existence :";
                cout << Counter :: count << "\n";
                return 0;
}
void f()
{
                Counter temp;
                cout << "Objects in existence :";
                cout << Counter :: count << "\n";
                // temp is destroyed when f() returns
}
```

The following is the output of the above examples.

```
Objects in existence :    1
Objects in existence :    2
Objects in existence :    3
Objects in existence :    2
```

As you can see, the static member variable count is incremented whenever an object is created and decremented when an object is destroyed. This way, it keeps track of how many objects of type Counter are currently in existence.

By using static member variables, you should be able to virtually eliminate any need for global variables. The trouble with global variables relative to OOP is that they almost always violate the principle of encapsulation.

11.5 Arrays of Objects

C++ supports arrays of any data type including class type. An array having class type elements known as Array of objects. An array of objects is declared after the class definition is over and it is defined in the same way as any other type of array is defined. Consider the following code fragment :

```
class Item
        {
                public :
                int itemno;
                float price;
                void getdata(int i, float j)
                {
                        itemno = i;
                        price = j;
                }
                void putdata(void)
                        {
                                cout << "Item No :" << itemno;
                                cout << "\n" << "Price" << price << "\n";
                }
        };
item order[10];                 // Array order to contain 10 objects of Item type
```

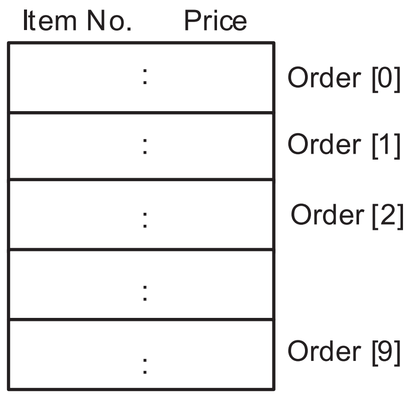
The array order contains 10 objects, namely, order[0], order[1], order[2], …….order[9]. To access data member itemno of 3rd object in the array, we'll give

order[2].itemno

Similarly, to invoke putdata() for 7th object in the array order, we'll give

Order[6].putdata();

An array of objects is stored in the same way as any other multidimensional array. However the array elements being objects have space only for their data members. Their member functions are stored separately at one place and will be used by all the objects of that class type. The array order's memory representation has been shown in below figure :



Memory presentation of order, an array of 10 Item objects

We can use an array of objects just like any other array. Following example program illustrates the use of object array :

Example: Program to illustrate the use of object arrays by storing details of 10 items in an array of objects.

```
#include <iostream.h>

#include <conio.h>                    // for clrscr()

class Item

        {

                int itemno;

                float price;

        public:

                voidgetdata(int i, float j)

                {

                        itemno = i;

                        price = j;

                }

                void putdata(void)

                {

                        cout << "Item No :" << itemno << "\n";

                        cout << "Price :" << price << "\n";

                }

        };

const int size = 10;

item order[size];                    // object array created
```

```
int main()
{
        clrscr();
        int ino;
        float cost;
                        // Get values for all items
        for(int a=0; a<size; a++)
        {
                cout << "Enter Item no & Price for Item" << a+1 << "\n";
                cin >>ino>>cost;
                order[a].getdata(ino, cost);            // Invoke getdata() for a particular
// object with the given values
        }
                        // Print details
        for(a=0; a<size; a++)
        {
                cout << "Item " << a+1 << "\n";
                order[a].putdata();             // Invoke putdata() for a particular object
        }
        return 0;
}
```

The above program first asks for details for 10 items one by one and then prints the details of these items.

11.6 Summary

You can use different types of arrays like single-dimension and two-dimension array in C++. Now you have been introduced to classes and objects, you yourself can determine how beneficial they are. One major benefit is the correspondence between the things being modeled by the program and the C++ objects in the program. Now you are able to define array within a class, using the objects in class and allocate memory to objects of class.

11.7 Self Assessment Questions

Q.1 What do you mean by single-dimension and two-dimension array?

Q.2 Write a program that checks whether a string is palindrome or not?

Q.3 Explain how do you initialize an array? Also give an example.

Q.4 What is the relationship of a class and its objects?

Q.5 How is memory allocated to a class and its objects?

Q.6 What do you mean by static data member of a class? Explain the characteristics of a static data member.

Q.7 When do we declare a member of a class static?

Q.8 How are static members different from non-static members of a class?

Unit-12 : Managing Console & I/O Operations

Structure of Unit:

12.0 Objectives

After going through this unit you should be able to:

- Understand the use of classes for file stream operations
- Identify and describe the purpose of the C++ standard I/O library
- How to manage a formatted I/O and unformatted I/O
- Extend your capabilities to manage output using manipulators

12.1 Introduction

This unit is an introduction to C++ stream input and output. This explains the iostreams facility, how it works in principle, and how it should be used. This unit is very useful for learners and readers who need basic information on iostreams. For the learners and readers who require deeper understanding, the unit also gives an overview of the iostream architecture, its components, and class hierarchy. Also explains the concepts of the formatted and unformatted I/O operations.

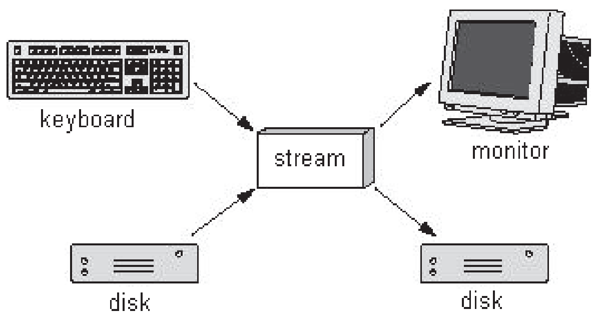
File I/O facilities of C++ are implemented through a component header file fstream of standard library. This fstream library predefines a set of operations for handling file related input/output activities. It provides its attributes to the file stream classes for input, output and both input and output respectively. These streams are necessary to be linked with the disk files for input and output purposes.

12.2 Overview on Streams

The C++ I/O system like the C language I/O system operates through streams. You should already know that a stream is logical device that either produces or consumes information. A stream is linked to a physical device by the C++ I/O system. All streams behave in the same manner, even if the actual physical devices they are linked to differ. Because all streams act the same, the I/O system presents the programmer with a consistent interface.

Until now, you have used cout to write data to the screen and cin to read data from the keyboard without specifying any format. In C++ as part of the programming language, no statements are defined for writing to a screen or to a file and for reading data into a program. However, C++ includes some libraries to facilitate I/O. These libraries contain some classes called **stream**.

The iostream classes view the flow of data from your program to the screen or a file. The class cout is a stream of output stream ostream. The reason cout is called a stream is that output sent to it flows like a stream of bytes. If things are sent to cout, they fall in line, one after to the other are dropped into the stream. Then they are displayed on the screen in that order. The name cout, pronounced see-out, stands for console output. Input is analogous to output. The class cin is a stream of input stream istream. In this case, we have data flowing in from the input stream. The main objective of streams is to encapsulate the problems of getting the data to and from the external storage (e.g. a hard-drive) or the screen. Once a stream is created, your program works with the stream and the stream hides the details. Hence, a stream can basically be represented as a source or destination of characters of indefinite length. In C++ , streams are used to perform input and output operations using the keyboard and to display information on the monitor of the computer. The following figure illustrates the functions of input output streams:



Functions of input output streams

12.2.1 Classes for File Stream Operations

To perform file processing, the I/O system of C++ provides various classes that define the file handling methods. These classes are ifstream for input from a file, ofstream for output to a file and fstream for both input from and output to a file. These classes are derived from istream ostream and from the corresponding iostream class. These classes defines the file handling methods and are designed to manage the disk files. These classes are declared inside a header file fstream.h derived from iostream.h, a header file that manages all console input and output operations. The following figure shows the hierarchy of stream I/O classes.

The ios Class

This general input/output stream class that contains basic facilities used by all the other input and output classes. This file is a base class for istream and ostream classes that are also the basic classes of other classes. It provides the basic support for formatted and unformatted I/O operations.

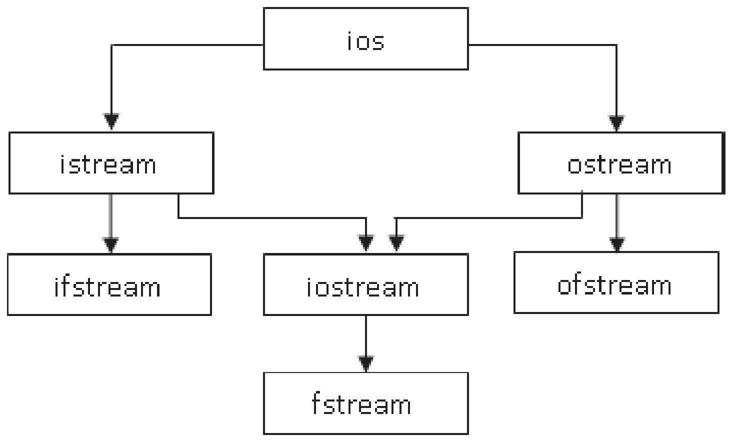


Fig 12.2 Stream class hierarchy

The istream Class

This class supports stream input operations. The stream class ifstream that defines member functions for file input is derived from istream and inherits all the attributes of its parent class istream. The istream class contains an overloaded extraction operators (>>) and member functions such as get(), getline(), read(), seekg() and tellg() for data input.

The ostream Class:

This class supports stream output operations. The stream class ofstream that provides output operations inherits all the attributes of its parent class ostream. The ostream class contains overloaded insertion operators (<<) and member functions such as put(), seekp(), tellp() and write() for data output.

Both istream and ostream classes are derived from **ios** and dedicated to input and output respectively. Their member functions perform both formatted and unformatted operations.

The ifstream Class

This class provides input operations. It contains open() with default input mode. This class is derived from istream class and inherits all the attributes of its parent class istream such as the functions; get(), getline(), read(), seekg() and tellg().

The ofstream Class

This class provides output operations. It contains open() with default output mode. This class is derived from ostream class and inherits all the attributes of its parent class ostream such as the functions; put(), seekg(), tellg() and write().

The iostream Class

The iostream class supports both stream-input and stream-output operations. It is derived through multiple inheritance from both the istream and ostream classes, hence contains all the input and output functions.

The fstream Class

The fstream class is derived from its base class of iostream, hence inherits all the functions from istream and ostream classes through its base class iostream. This class provides support for both input and output operations. It contains an open() with default input mode.

The fstream.h

The fstream.h header file is used for both reading and writing a stream of objects on a file. This file includes the definitions for the stream classes ifstream for input from a file, ofstream for output to a file and fstream for both input from and output to a file.

This is derived class of iostream and automatically includes the header file iostream.h when used with an include directive.

Syntax and Example is given by:

#include<fstream.h>

12.2.2 Declaring File Streams

In C++ programs, as a variable is declared by using a variable name preceded by a built-in data type, a file stream object is also declared in the same way. A file stream object name is declared by a suitable name preferably using the part of input or output stream class name preceded by ifstream or ofstream class to declare an input or output stream object. Syntax is given by:

Syntax:

ifstream stream-object; // declaring input stream object

ofstream stream-object; // declaring output stream object

Example:

- ifstream infile; // input object declaration
- ofstream outfile; // output object declaration

12.3 Unformatted I/O Operations

Unformatted Input/Output is the most basic form of input/output. Unformatted input/output transfers the internal binary representation of the data directly between memory and the file.

12.3.1 Input Stream

The input stream does read operation through keyboard. It uses cin as object. The cin statement uses >>(extraction operator) before variable name. The entered data is separated by space, tab or enter. More than one variable can be used in cin statement to input data. Such operation is known as cascaded input operations.

***Note:-**While entering string, blank spaces are not allowed.

Syntax for using the standard input stream is cin followed by the operator >> followed by the variable that stores the data extracted from the stream.

For example:

```
int prog;

cin>>prog;
```

In the example above, the variable prog is declared as an integer type variable. The next statement is the cin statement. The cin statement waits for input from the user"s keyboard that is then stored in the integer variable prog. The input stream cin is waiting for the user to input a value, before proceeding for processing or storing the value. This duration is dependent on the user pressing the RETURN key on the keyboard. It is also possible to request input for more than one variable in a single input stream statement.

A single cin statement is as follows:

```
cin>>x>>y;
```

It is also written as:

```
1. cin>>x;
2. cin>>y;
```

In both of the above cases, two values are input by the user, one value for the variable x and another value for the variable y.

12.3.2 Output Stream

The output stream manages output of the stream. It displays contents of the variables on the screen. It uses << (insertion operator) before variable name. It uses the cout object to perform console write operation.

Example to illustrate the use of Input and Output streams

```
#include <iostream. h>
using namespace std;
void main()
{
int a,b;
cout << "Enter the value of a: ";
cin >> a;
b=a+10;
cout << "Value of b is: " << b;
```

The following is the output of above program:

```
Enter the value of a : 10
value of b is : 20
```

12.3.3 Unformatted Console I/O Functions

get() and put() function

Single Character input and output operations in C++ can be done by using two important functions. They are : get() and put() functions.

The get() function is used to read a character and put() function is used to display a character on a screen. The get() function has two syntaxes

```
1.get(char *);
2.get(void);
```

If syntax1 is used the get() function assign the read data to its argument whereas if syntax 2 is used the get() function returns the data read. The data is assigned to the variable on the left hand side of assignment operator.

The put() function is used to display a character on screen. The syntax of put() is as follows:

```
cout.put('A');
cout.put(x);
```

Example Program to illustrate the use of get ().

```
#include <iostream.h>
#include <fstream.h>
using namespace std;
 int main()
{
char str[80];
cout << "Enter your name: ";
cin.get(str, 79);
cout << str << '\n';
return 0;
}
```

getline() function

The getline() function reads the string including spaces .The cin function does not allow the string to enter blank spaces. The input reading is terminated when user presses the enter key. The new line character is accepted but not saved and is replaced with null character.

```
cin.getline(variable,size);
```

Example Program to illustrate the use of getline( )

```
#include <iostream.h>
int main ()
{
char mybuffer [100];
cout << "What's your name? ";
cin.getline (mybuffer,100);
cout << "Hello " << mybuffer << ".\n";
cout << "Which is your favourite team? ";
cin.getline (mybuffer,100);
cout << mybuffer;
return 0;
}
```

read() and write() function

The read() function is used to read text through the keyboard .When we see read statement, it is necessary to enter character equal to the number of size specified .The syntax of this function is given by:

cin.read(variable,size);

The write() function is used to display the string on the screen.Its format is same as getline() but the function is exactly opposite. cout.write() displays only specified number of characters given in the second argument,though actual string may be more in length. These functions are also described in next unit. The syntax of this funciton is given by:

cout.write(variable,size);

peek() and ignore() function

peek() function

This function returns the succeeding characters without extraction.

cin.peek()==’#’;

ignore() function

The member function ignore() has two arguments, maximum number of characters to avoid and the termination character.

cin.ignore(1,’#’) ;

This statement ignores 1 character till # character is found.

putback() function

The putback function replaces the given character into the input stream.

cin.putback(‘*’) ;

The putback() function sends back the given symbol into input stream. It replaces the previous character with the new one specified in the putback() function.

Example program to illustrate the use of peck ( ), ignore ( ) and putback ().

```
#include <iostream.h>
using namespace std;
int main()
{
char ch;
cout << ”enter a phrase: ”;
while ( cin.get(ch) )
{
if (ch == ’!’)
cin.putback(‘$’);
else
cout << ch;
while (cin.peek() == ’#’)
cin.ignore(1,’#’);
}
return 0;
}
```

gcount() function

This function returns the number of unformatted characters extracted from input stream.

Example Program is to illustrate the use of gcout ();

```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;
int main()
{
const int SIZE = 80;
char buffer[ SIZE ];
cout <<“Enter a sentence:” << endl;
cin.read( buffer, 20 );
cout.write( buffer, cin.gcount() );
cout << endl;
return 0;
}
```

12.4 Formatted I/O Operations

When a C++ program that includes the iostream classes starts, the following objects are created and initialized. Each object has operators and methods that can be used to facilitate the input/output process. Instead, a set of functions and streams are provided for formatted I/O. C++ provides three types of console I/O functions. These are following:

- ios class function
- Manipulators
- User Defined output functions

12.4.1 IOS Class

The classes derived from ios are special I/O with high level formatting operations. The iostream class is automatically loaded in the program by the compiler. The IOS class functions are:

- width() : To set the required field width. The output will be displayed in given width.
- precision() : To set number of decimal point for a float value.
- fill() : To set a character to fill in the blank space of a field.
- setf() : To set various flags for formatting output.
- unsetf() : To remove the flag setting.

Using width( ), precision( ), and fill( )

In addition to the formatting flags, there are three member functions defined by the **ios** class that set these format parameters: the field width, the precision and the fill character, respectively. By default, when a value is output, it occupies only as much space as the number of characters it takes to display it. However, you can specify a minimum field width by using the **width()** function. Its prototype is:

streamsize width(streamsize w);

Here w becomes the field width, and the previous field width is returned. The **streamsize** type is defined by <**iostream**> as some form of integer. In some implementations, each time an output is performed, the field width returns to its default setting, so it might be necessary to set the minimum field width before each output statement. After you set a minimum field width, when a value uses less than the specified width, the field is padded with the current fill character (the space, by default) so that the field width is reached. However, keep in mind that if the size of the output value exceeds the minimum field width, the field will be overrun. No value is truncated.

By default, six digits of precision are used. You can set this number by using the **precision( )** function. Its prototype is :

streamsize precision(streamsize p);

Here the precision is set to p and the old value is returned.

By default, when a field needs to be filled, it is filled with spaces. You can specify the fill character by using the fill( ) function. Its prototype is :

char fill(char ch);

After a call to fill(), ch becomes the new fill character, and the old one is returned.

Example program to illustrates the format functions :

```
#include <iostream.h>
using namespace std;
int main( ) {
cout.width(10);                  // set minimum field width
cout << “hello “<<“\n”;          // right justify be default
cout.fill(‘%’);                  // set fill character
cout.width(10);                  // set width
cout << “hello”<<“\n”;           // right justify by default
cout.setf(ios::left);            // left justify
cout << “hello”<<“\n”;           // output left justified
cout.width(10);                  // set width
cout.precision(10);              //set 10 digits of precision
cout << 123.234567 << “\n”;
cout.width(10);                  // set width
cout.precision(6);               // set 6 digits of precision
```

```
        cout << 123.234567 << "\n";
        return 0;
        }
```

The following is the output of the above program:

```
hello
%%%%%hello
hello%%%%%
123.234567
123.235%%%
```

12.5 Managing Output with manipulators

Formatting output is important in the development of output screens, which can be easily read and understood. C++ offers the learners and programmers several input/output manipulators. You will learn here, what a manipulator is, and how is used to manage the output of the program. Manipulators are operators used in C++ for formatting output. The data is manipulated by the programmer's choice of display. To format the output in C++ program you have to use a combination of escape sequences and output manipulators. Many of these formatting tools require you to include the iomanip header file. There are numerous manipulators available in C++.

Escape sequences	
\n	new line
\t	Tab
\\	Backslash
\a	Alarm

Manipulators	
Flush	flushes the output buffer
Endl	flushes the output buffer and inserts a new line
Oct	sets the output base to octal
dec	sets the output base to decimal
hex	sets the output base to hexadecimal

Parameterized Manipulators	
setbase(base)	sets the output base (10=decimal, 8=octal, 16 = hex)
setw(width)	sets minimum output field width
setfill(char)	sets the fill character to be used when width is set.
setprecision(n)	sets the precision for floating point numbers.
setiosflags(flag)	sets one or more of the ios flags shown below
resetiosflags(flag)	resets one or more ios flags

Some of the more commonly used manipulators such as endl, setw, setfill and setprecision manipulator are explained here below:

endl manipulator:

This manipulator has the same functionality as the 'n' newline character.

For Example:

```
cout << "VMOU" << endl;
cout << "Vardhman Mahaveer Open Univerity Kota";
```

It gives the following output:

```
VMOU
Vardhman Mahaveer Open Univerity Kota
```

setw manipulator:

This manipulator sets the minimum field width on output. The syntax is:

setw(x)

Here setw causes the number or string that follows it to be printed within a field of x characters wide and x is the argument set in setw manipulator. The header file that must be included while using setw manipulator is #include <iomanip.h>.

Example to illustrate the use of setw ( ) .

```
#include <iostream.h>
using namespace std;
#include <iomanip.h>
void main( )
{
int x1=12345,x2= 23456, x3=7892;
cout << setw(8) << "VMOU" << setw(20) <<"Values"<< endl
<< setw(8) << "E1234567" << setw(20)<< x1 << endl
<< setw(8) << "S1234567" << setw(20)<< x2 << endl
<< setw(8) << "A1234567" << setw(20)<< x3 << endl;
}
```

The following is the output of the above example :

```
VMOU                    Values
E1234567                12345
S1234567                23456
A1234567                 7892
```

setfill manipulator:

This is used after setw manipulator. If a value does not entirely fill a field, then the character specified in the setfill argument of the manipulator is used for filling the fields.

Example to illustrate the use of settfil ( ).

```
#include <iostream. h>

using namespace std;

#include <iomanip.h>

void main()

{

cout << setw(10) << setfill('$') << 50 << 33 << endl;

}
```

The following is the output of the above program :

```
$ $ $ $ $ $ $ $ 5 0 3
```

This is because the setw sets 10 for the width of the field and the number 50 has only 2 positions in it. So the remaining 8 positions are filled with $ symbol which is specified in the setfill argument.

12.5.4 setprecision manipulator:

The setprecision manipulator is used with floating point numbers. It is used to set the number of digits printed to the right of the decimal point. This may be used in two forms:

- fixed
- scientific

These two forms are used when the keywords fixed or scientific are appropriately used before the setprecision manipulator. The keyword fixed before the setprecision manipulator prints the floating point number in fixed notation. The keyword scientific, before the setprecision manipulator, prints the floating point number in scientific notation.

For Example:

```
#include <iostream.h>

#include <iomanip.h>

using namespace std;

void main( )

{

float x = 0.1;

cout << fixed << setprecision(3) << x << endl;

cout << scientific << x << endl;

}
```

The following is the output of the above example:

```
0.100
1.000e.001
```

The first cout statement contains fixed notation and the setprecision contains argument 3. This means that three digits after the decimal point and in fixed notation will output the first cout statement as 0.100. The second cout produces the output in scientific notation. The default value is used since no setprecision value is provided.

Example :

Program to illustrate several of the I/O manipulators

```
#include <iostream.h>
#include <iomanip.h>
using namespace std;
int main( ) {
cout << hex << 100 << endl;
cout << oct << 10 << endl;
cout << setfill('X') << setw(10);
cout << 100 << " hi " << endl;
return 0;
}
```

The following is the output of the above program.

```
64
13
XXXXXXX144 hi
```

12.6 Summary

You have noticed that I/O in C++ is implemented with streams. Abstractly, a **stream** can be thought of as a sequence of bytes of infinite length that is used as a buffer to hold data that is waiting to be processed. Typically you deal with two different types of streams. **Input streams** are used to hold input from a data producer, such as a keyboard, a file, or a network. For example, the user may press a key on the keyboard while the program is currently not expecting any input. Rather than ignore the users keypress, the data is put into an input stream, where it will wait until the program is ready for it. Conversely, **output streams** are used to hold output for a particular data consumer, such as a monitor, a file, or a printer. When writing data to an output device, the device may not be ready to accept that data yet. For example, the printer may still be warming up when the program writes data to its output stream. The data will sit in the output stream until the printer begins consuming it. Some devices, such as files and networks, are capable of being both input and output sources.

There are three types of file stream objects used in C++. Each object has operators and methods that can be used to facilitate the file input/output process.

- **ifstream** – input file stream object facilitates reading files

- **ofstream** – output file stream objects facilitates writing to files.
- **fstream** – file stream objects supports both read and write operations on files.

The state of the stream objects can be changed by using manipulators. These can set the formatting and display characteristics and various other attributes of the stream objects. File I/O can be accomplished by using the fstream classes, which derive from the stream classes.

I/O manipulators are special I/O format functions that can occur within an I/O statement, instead of separate from it. The standard manipulators are discussed here.

12.7 Self Assessment Questions

1. What are streams? Explain the hierarchy of stream I/O classes.
2. Differentiate between get() and getline() functions with suitable example.
3. What is the difference between unformatted I/O and formatted I/O?
4. Write advantages and disadvantages of formatted I/O.
5. What do you mean by I/O manipulators? Explain in brief.
6. Define file stream objects. How many types of file stream objects used in C++?
7. Define the following manipulators:

   (i) setw (ii) setprecision (iii) setfill (iv) endl
8. Explain the following:

   (i) gcount() (ii) put() (iii) peek() (iv) ignore()

Unit-13 : Working with Files

Structure of Unit :

13.0 Objectives

After going through this unit you should be able to:

– Understand and use of the ifstream, ofstream and fstream class objects.

– Identify and describe the use of opening and closing files.

– Describe the detecting end of files and file pointers.

– Use of sequential I/O with files-read and write member functions.

– State the significance of Error Handling during file input output.

13.1 Introduction

You have learnt in the previous unit, most computer programs work with files. This is because, files help in storing data or information permanently. The word processor creates document files. Database programs create various types of files which store the information. The compilers read source files and generate executable files. Thus, here, we see it is the files which are mostly worked with, inside programs. Therefore, we say a file itself is a **bunch of bytes** stored on some storage devices like magnetic tape, or magnetic disks etc. The file I/O facilities are implemented in C++ by using a header of standard library. That header file is **fstream.h**. It is C++ programming language uses the concept of streams to perform input and output operations using the keyboard and to display information on the monitor of the computer.

The fstream library is use to predefine the set of operations for handling file related input and output. It defines certain classes which help one perform file input and output.

For example: ifstream class ties a file to the program for input purpose, ofstream class ties a file to the program for output purpose , and fstream class ties a file to the program for both input and output. The file manipulation and related operations using streams are the topics we are going to discuss in this unit.

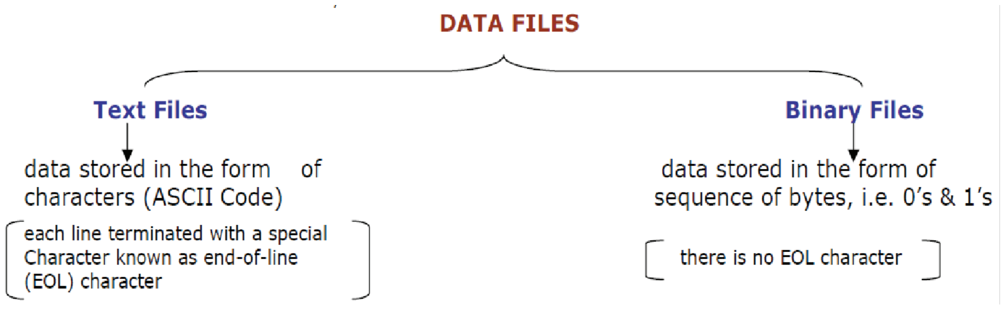
13.2 File I/O Streams

To store and retrieve data or information on a file you should use the following four components:

- files
- file stream objects
- file stream methods
- file modes

13.2.1 Files & Their Types

In C++, a file, at its lowest level, is interpreted simply as **a sequence of or stream of bytes**. Generally, files in terms of computers are of two types. One that stores instruction for the computer, i.e. a program files and the second that stores data, i.e. data file. These files are shown here.


DATA FILES
Text Files
data stored in the form of characters (ASCII Code)
each line terminated with a special Character known as end-of-line (EOL) character
Binary Files
data stored in the form of sequence of bytes, i.e. 0's & 1's
there is no EOL character

13.2.2 File Stream Objects

You have learnt in previous unit that a stream is a general name given to the flow of data. Different streams are used to represent different kind of data flow. Each stream is associated with a particular class of **fstream.h** header file of the standard library in C++.

There are three types of file stream objects used in C++. Each object has operators and methods that can be used to facilitate the file input/output process. These streams are given below:

Type of stream	Associated Class
input stream (Read mode)	ifstream
output stream (Write mode)	ofstream
input/output stream (R/W mode)	fstream

13.3 Opening and Closing a File

Before opening a file, you will create a file stream object of a particular class (i.e. ifstream, ofstream or fstream) depending upon the type of operation. For example, in order to open a file as an input file i.e. data will be read from it and no other operation would take place, you will create a file stream object of ifstream type. Similarly, in order to open an output file (on which no operation can take place except writing only) you shall create a file stream object of ofstream type.

A file can be opened in C++ by using two methods:

(i) By using the **constructor** function of the stream class to be used

(useful when we use only one file in the stream)

for example: ofstream outfile(“marks.dat”);

(ii) By using **open()** function of the stream class to be used

(useful in case of multiple files)

for example: ofstream outfile;

outfile.open(“marks.dat”);

The first method is preferred when a single file is used with a stream, however, for managing multiple files with the same stream, the second method is preferred. Let us discuss each of the methods one by one.

Firstly you should know the syntax and examples of these two methods that are as follows:

By using Constructor	By using member function Open ( )

Syntax
filestream object;("filename",mode);
 	
Example
ifstream fin ("abc.txt");
	Syntax
	filestream'object;
	object.open("filename", mode);
	Example
	ifstream fin :
	fin.open("abc.txt");

Note:- (a) Modes are optional and given at the end.

(b) Filename must follow the convention of 8.3 and it’s extension can be anyone.

13.3.1 Opening Files using Constructors

As you know that a constructor of a class initializes an object of its class when it (the object) is being created. Same way the constructor of the stream classes are used to initialize file stream objects with the file names passed to them. Let us consider a situation wherein you desire to open a file called “message.dat” containing the following text:

Vardhman Mahaveer Open University Kota

Formerly: Kota Open University Kota

You will open this file by using the default constructor of the ofstream class by the following statement :

ofstream myfile(“message.dat”);

The above declaration means: open an output stream called myfile and initialize it with the name “message.dat”. The “message.dat” is the name of the file lying on the floppy or hard disk. Thus whetever is written in myfile stream in the program, will actually be written into the file “message.dat”. Since a simple text is to be written into a file, we can use the operator << to write the data. Here we consider an example program for this purpose. The following program creates the required file:

Example program to create/writing a file

```
//This program creates a file called “message.dat”
//include<fstream.h>            //required for I/O
void main()
{
ofstream myfile(“message.dat”);         // create file for output
if(!myfile) {                           //checks if the file is opened or not
cout<<“Cannot open this file..”;
return 1;
}
myfile<< “Vardhman Mahaveer Open University Kota\n”;          // send text to file
myfile<< “Formerly: Kota Open University Kota”;
myfile.close();                         // close the file
}
```

It may be noted that in above program, **getfrom operator <<** is appropriately overloaded; we used it to write text to the file. Moreover the function close() has been used to close the file at the end of the program.

We can open the create file for input by using the default constructor of the ifstream class by the following statement:

```
ifstream yourfile(“message.dat”);
```

The above declaration means: open an input stream called yourfile and initialize it with the name “message.dat”. Thus the “message.dat” will be known as yourfile in the program. Since it is a simple text file, we can use the operator >> to read the data from the file. However there is a limitation of the operator >> to read a string containing blanks and white space characters. To remove this you will use get() and getline() functions as input member functions of the ifstream class for text files in this unit Next.

13.3.2 Opening Files using Open() function

There may be some situations requiring a program to open more than one file. The strategy for opening multiple files depends upon how they will be used. If the situation requires simultaneous processing of two files, then you need to create a separate stream for each file. However, if the situation demands sequential processing of files (i.e. processing them one by one), then you can open a single stream and associate it with each file in turn. To use this approach, declare a stream object without initializing it, and then use a second statement to associate the stream with a file. For example:

```
ifstream afile;              //create an input stream
afile.open(“vmou.dat”);      // associate afile stream with file vmou.dat
      .                      //process vmou.dat
```

```
afile.close();          //terminate association with vmou.dat
afile.open("kou.dat");  //associate afile stream with file kou.dat
    .
    .                   //process kou.dat
afile.close();          //terminate association
```

The above code lets you handle reading two files in succession. It is noted that the first file is closed before opening the second one. This is necessary because a stream can be connected to only one file at a time.

13.3.3 The File Access Modes

The file mode describes how a file is to be used: to read from it, to write to it, to append it, and so on. When you associate a stream with a file, either by initializing a file stream object with a file name or by using the **open()** method, you can provide a second argument specifying the file mode, as mentioned below:

```
stream_object.open("filename",(filemode));
```

The second argument of **open()**, the filemode, is of type **int**, and you can choose one from several constants defined in the **ios** class. For example, we can set the file mode so that the output is appended to the end of a file rather than writing over the current contents of the file. With an fstream object, the mode is used to determine whether the file is to be used for input or output. The following table shows the various filemodes:

File mode constants	Meaning	Associated Class
ios::in	open for reading or input mode (default for ifstream)	Ifstream
ios::out	open for writing or output mode (default for ofstream)	Ofstream
ios::app	Start writing at end of file (APPend)	Ofstream
ios:ate	Start reading or writing at end of file (AT End)	ofstream ifstream
ios::trunk	Truncate file to zero length if it exists (TRUNCate)	Ofstream
ios::nocreate	Error when opening if file does not already exist	Ofstream
ios::noreplace	Error when opening for output if file already exist, unless ate or app is set	Ofstream
ios::binary	Open file in binary (not text) mode	ofstream ifstream

The **fstream** class does not provide a mode by default and, therefore, one must specify the mode explicitly when using an object of the **fstream** class. You can combine two or more file mode constants using the C++ bitwise OR operator(|).

For example : the following statements:

```
ofstream fout;
fout.open("vmou", ios::app|ios::nocreate);
```

will open a file in the append mode if the file exists and will abandon the opening operation if the file does not exist.

To open a binary file, you need to specify ios::binary alongwith the file mode. For exmpale:

```
fout.open("vmou", ios::app|ios::binary);
```

Or `fout.open("kou", ios::out|ios::noncreate|ios::binary);`

13.3.4 Closing a File

As already mentioned, a file is closed by disconnecting it with the stream it is associated with. A file can be **closed** in two ways:

(i) If the file has been opened using constructor, it gets closed automatically as soon as the stream objects go out of scope. This calls the destructor, which closes the file.

(ii) By using **close()** function of the class

The all types of the files can be closed using **close( )** member function. It takes the following syntax:

```
Stream_object.close( );
```

Example:

```
fin.close( );    // here fin is an object of istream class
```

The complete program:

```
//Write a program that receives the roll numbers and marks of the students and store these details into
// a file called marks.dat
#include<fstream.h>
void main()
{
ofstream myfile;
myfile.open("marks.dat", ios::out);
char ans= 'y';
int rollno;
float marks;
while(ans== 'y'||ans= ='Y')
{
cout<< "\n Please enter roll number";
cin>>rollno;
cout<< "\n Please enter marks";
cin>>marks;
myfile<<rollno<<'\n'<<marks<<'\n';
cout<< "\n Want to insert more records?(y/n)…";
cin>>ans;  }
myfile.close();
}
```

After running this program, there is no output to the screen. To see the output you have to go **DOS Shell** option under File menu and then upon the DOS prompt, type the following command:

```
TYPE marks.dat
```

and press enter key, it will show the file contents. To get back to your program screen in Turbo C++, type EXIT and press enter.

13.4 Detecting End-of-File(EOF)

As far as files are concerned, till now you have been reading and writing a certain number of characters or objects. While reading a file, a situation can arise when you do not know the number of objects to be read from the file i.e. you do not know where the file is going to end? A simple method of detecting end of file(eof) is discussed here.

A file pointer in a data file keeps on moving or changing its position with every read (get-file pointer) or write(put-file pointer) operation. The syntax of eof() function is given by:

Syntax:

filestream_object.eof( );

Following are the methods to find whether the file pointer has reached to end of the file or not.

Detecting EOF with eof()	**Detecting EOF without eof()**

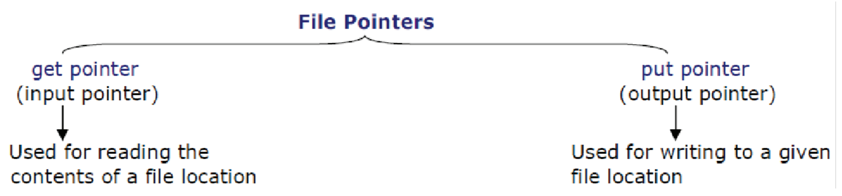
#include <fsteam.h> // Using header file
void main()
{ ifstream aFile("StName.txt"); // opening a file
char ch;
while(!aFile.eof())   *// checking for EOF*
{ aFile.get(ch); // reading file
cout<<ch; // displaying read data
}
aFile.close();
}
	#include <fstream.h> // Using header file
	void main()
	{ ifstream aFile("StName.txt"); // opening a file
	char ch;
	while(aFile) *// checking for EOF*
	{ aFile.get(ch); // reading file
	cout<<ch; // displaying read data
	}
	aFile.close();
	}

In the first code of above table: aFile.eof() returns true if aFile is at the EOF or false otherwise. And therefore we are using 'while (!aFile.eof())' so that until and unless aFile is at EOF while loop will iterate. Here note the '!' operator.

In the second column of above table: aFile returns false if aFile is at the EOF or true otherwise. And therefore we are using 'while (aFile)' so that until and unless aFile is at EOF while loop will iterate.

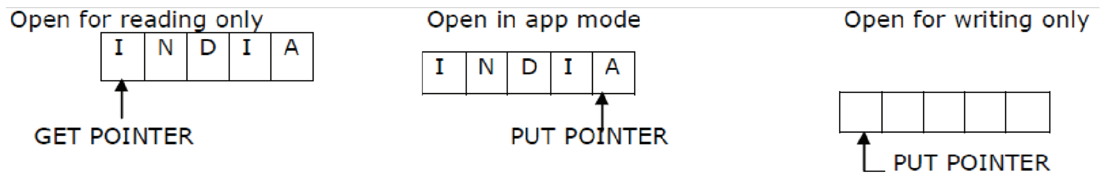
13.4.1 File Pointers and Manipulation

All I/O streams objects have, at least, one internal stream pointer: ifstream, like istream, has a pointer known as the **get pointer** that points to the element to be read in the next input operation. ofstream, like ostream, has a pointer known as the **put pointer** that points to the location where the next element has to be written. Finally, fstream, inherits both, the get and the put pointers, from iostream (which is itself derived from both istream and ostream). Thus you can say that each file has two pointers associated with it which are called file pointers. One of them is the input pointer, also known as get pointer; and the other one is called output pointer or the put pointer. These file pointers are explained in detail.


File Pointers
get pointer
(input pointer)
Used for reading the contents of a file location
put pointer
(output pointer)
Used for writing to a given file location

Note:-

Each time an input or output operation takes place, the concerned pointer is automatically advanced.



How can the file pointers be moved in a file?

The file stream classes support some predefined functions that navigate the position of the file pointers. The relevant functions are: seekg(), seekp(), tellg(), and tellp(). Some of the important functions are discussed here.

Member function name	**Explanation**

seekg( )	Used to move **reading** pointer forward and backward
	Syntax
	fileobject.seekg(no_of_bytes,mode);
	Example:
	(a) fout.seekg(50,**ios::cur**); // move 50 bytes forward from current position
	(b) fout.seekg(50,**ios::beg**); // move 50 bytes forward from current beginning
	(c) fout.seekg(50,**ios::end**); // move 50 bytes forward from end .

seekp( )	Used to move **writing** pointer forward and backward
	Syntax
	fileobject.seekp(no_of_bytes,mode);
	Example:
	(a) fout.seekp(50,**ios::cur**); // move 50 bytes forward from current position
	(b) fout.seekp(50,**ios::beg**); // move 50 bytes forward from current beginning
	(c) fout.seekp(50,**ios::end**); // move 50 bytes forward from end .

tellp( )	It return the distance of **writing** pointer from the beginning in bytes Syntax Fileobject.tellp( ); Example: long n = fout.tellp( );
tellg( )	It return the distance of **reading** pointer from the beginning in bytes Syntax Fileobject.tellg( ); Example: long n = fout.tellg( );

13.5 Sequential I/O Operations

The file stream classes supports a number of member functions for performing input and output operations in file handling. One pair of functions contains **get()** and **put()** functions that designed for handling a single character at a time. The function **getline()** lets you handle multiple characters at a time. Another pair of functions **read()** and **write()** functions designed to read and write blocks of binary data. These functions are known as member functions for input output. Here you will learn one by one of them.

13.5.1 The get(), put() and getline() functions

get(): The get() function is byte-oriented. That is used to read a byte of data from a specified file. This member function has various forms such as get() with no arguments, get() with character reference argument and get() with three arguments. But most commonly used form is discussed here.

```
istream & get(char &ch);
```

The get() function reads a single character from the associated stream and puts that value in **ch**. It returns a reference to the stream.

Take a look at the following example that illustrates the use of get() function :

Example Program to display contents of a file using get() function with one argument

```
#include<fstream.h>
#include<conio.h>
int main()
{
clrscr();
char ch;
ifstream ifile;     // create input stream
ifile.open("marks.dat", ios::in);  // open file
if(!ifile)  // if ifile stores zero i.e. false value
{
cout<<"cannot open file\n";
return 1;
}
while(ifile)    // ifile will be 0 when eof is reached
{
ifile.get(ch); // reading a character
cout<<ch;   // displaying the read character
}
ifile.close();
return 0;
}
```

The following is the output of the above program.

```
101

78

102

86

103

79
```

As stated earlier, when the end-of-file is reached, the stream associated with the file becomes false. Thus, when ifile will reach the end-of-file, it will be false and cause to stop the while loop.

In addition to the above, there is another form of get() function takes three arguments; a character array, a size limit and a delimiter having value '\n'. This function reads characters from input stream upto one less than the size of the characters and terminates as soon the delimiter is encountered. A null character is inserted to terminate the input in the character array used as a buffer by the program. The delimiter is not stored in the character array though, remains in the input stream. The syntax of this function is as follows:

Syntax:

```
istream & get(char* buf, stream-size N, char delim);
```

Thus function reads character into a character array pointed to by **buf** (a user defined name) until either **N** characters are read or the specified delimiter (any specified character) is found. If no **delim** character is specified, by default a new line character is inserted that acts as a delimeter.

Example:

```
1. char.length[50];
   cin.get (length, 50, $);
2. cin.get (length, 50,@);
3. cin.get (length, 50);
```

In this example, the call to **cin.get()** does not specify a delimiter character, so the default **'\n'** will automatically be used.

put(): This function is also byte-oriented. The put() member function also operates on characters. It writes a single character to the associated stream. This function is commonly used with get() member function where get() reads a character and put() writes a character. The syntax of this function is given by:

```
ostream & put (char ch);
```

The put() function writes the value of **ch** to the stream and returns a reference to the stream.

Example program to write a single character at a time to a disk file from the entered string through keyboard using put() function.

```
#include<fstream.h>

#include<conio.h>

void main()
```

```
{
clrscr();
int i;
char ch, string[60];
cout<<"Please enter string:";
cin>>string;
ofstream outfile("string.cha");     // by default output mode due to ofstream specification
for( i=0;i<strlen(string);i++)
outfile.put(string[i]);
}
```

The following is the output of the above program.

```
Please enter string: VMOU is the best Open University
```

getline():

This function is a member of each input stream class that is used to perform input operations. The getline() function reads the new line character but does not store it in the character array. Instead, the function inserts a null character after the line in the character array.

This function can be invoked by using the object **cin** where the function call invokes this function which reads character input into the variable line. The reading is terminated as soon as either a new line character (\n) is found or size-1 characters have been read.

Syntax of this function is given by :

```
istream & getline (char*buf, streamsize N);
istream & getline (char*buf, streamsize N, char delim);
```

For example:

```
1. char city[25];
   cin.getline(city,25);
2. max=50;
   infile.getline(buffer, Max);
```

Example program to read the text using getline() function.

```
#include<fstream.h>
#include<conio.h>
void main()
{
clrscr();
const int Max=20;
char city[Max];
cout<<"City Name:";
cin.getline(city,Max);
cout<<"Enter City Name ="<<city<<endl;
cout<<"Another City Name:";
cin.getline(city,Max);
cout<<"Entered City Name ="<<city<<endl;
}
```

The following is the output of the above program.

```
City Name:  Kota
Entered City Name = Kota
Another City Name: Jaipur
Entered City Name = Jaipur
```

13.5.2 Writing and Reading Data in Binary Format

To write and read data in binary format two member functions are available in C++ language. They are following : read( ) and write( ) functions. These two functions are used to perform unformatted I/O operations. These functions handle the data in binary form hence, also known as binary input and output functions. The write() function writes data to a file to binary format and the read () function reads data from the file. The syntax of these functions are as follows:

Syntax for write ();

```
        Fileobject,write( (char *) & object, sizeof(object) );
```

Syntax for read ();

```
        Fileobject,read ( (char *) & object, size of(object) );
```

Example:

```
FILE.read((char *)&OBJ, sizeof(OBJ));
FILE.write((char *)&OBJ, sizeof(OBJ));
```

Example Program to that illustrates the use of write( ) member function

```
#include<fstream.h>
#include<iostream.h>
using namespace std;
        struct student
        {
        int roll ;
        char name[30];
        char address[60];
        };
```

```
int main()
            {
            students;
              ofstream fout;
              fout.open("student.dat");
               cout<<"\n Enter Roll Number :";
             cin>>s.roll;
              cout<<"\n Enter Name :";
             cin>>s.name;
             cout<<"\n Enter address :";
             cin>>s.address;
                fout.write((char *)&s,sizeof(student));
               fout.close();
             return 0;
            }
```

Example Program that illustrates the use of read( ) member function

```
            #include<fstream.h>
            #include<iostream.h>
            #include<conio.h>
            using namespace std;
            struct student
             {
             int roll ;
             char name[30];
             char address[60];
            };
int main()
             {
            student s;
             ifstream fin;
             fin.open("student.dat");
             fin.read((char *)&s,sizeof(student));
             cout<<"\n Roll Number :"<<s.roll;
```

```
  cout<<"\n Name       :"<<s.name;
 cout<<"\n Address    :"<<s.address;
  fin.close();
getch();
 return 0;
}
```

13.5.3 Writing and Reading Class Objects in a File

The functions write() and read() can also be used for writing and reading class objects. These functions handle the entire structure of an object as a single unit, using the computer's internal representation of data. For instance, the function write() copies a class object from memory byte by byte with no conversion. But one thing you must remember that only the data members are written to the disk file and not the member functions. The length of an object is obtained by **sizeof** operator and it represents the sum total of lengths of all data members of the object.

Example Program to illustrate Writing Class Object in a File

```
            #include<fstream.h>
            #include<iostream.h>
            using namespace std;
            class student
             {
            int roll ;
            char name[30];
            char address[60];
public:
            void read_data( );   // member function prototype
            void write_data( );  // member function prototype
            };
void student::read_data( )      // member function defintion
             {
            cout<<"\n Enter Roll :";
             cin>>roll;
            cout<<"\n Student name :";
            cin>>name;
             cout<<"\n Enter Address :";
            cin>>address;
            }
            void student:: write_data()
            {
            cout<<"\n Roll :"<<roll;
             cout<<"\n Name :"<<name;
            cout<<"\n Address :"<<address;
            }
            int main()
            {
              student s;
            ofstream fout;
              fout.open("student.dat");
    s.read_data();  // member function call to get data from KBD
    fout.write((char *)&s,sizeof(student)); // write object in file
    fout.close();
     return 0; }
```

Example Program to illustrate reading class object from a binary file

```
#include<fstream.h>
#include<iostream.h>
#include<conio.h>
using namespace std;
class student
 {
      int roll ;
      char name[30];
      char address[60];
    public:
         void read_data( );   // member function prototype
         void write_data( );  // member function prototype
        };
void student::read_data( )      // member function defintion
 {
   cout<<"\n Enter Roll :";
   cin>>roll;
```

```
    cout<<"\n Student name :";
     cin>>name;
    cout<<"\n Enter Address :";
     cin>>address;
}
void student:: write_data()
 {
    cout<<"\n Roll :"<<roll;
    cout<<"\n Name :"<<name;
    cout<<"\n Address :"<<address;
  }
int main()
 {
     student s;
    ifstream fin;
    fin.open("student.dat");
    fin.read((char *)&s,sizeof(student));
    s.write_data();
    fin.close();
    getch();
    return 0;
     }
```

13.6 Error Handling During File I/O

Sometimes visible or invisible errors may occur during the file operations. For instance, any of the errors may occur such as a file being opened for reading might not exist, a specified file name may be an invalid file name, an invalid file operation may be attempted or perhaps no more room might exist on the disk. To check for such errors and to handle file operations smoothly, C++ stream classes inherit a **'stream state'** member form the class **ios.** The stream state member functions provide the information on the status of a file that is currently being used. The provided information like end-of-file has been reached or file open failure and so on. These members use bit fields to store the status of the error condition.

There are several Boolean functions that indicate whether an error has occurred and what the state of the file stream is. These functions return the setting of several of the file's internal state flags. The following is a list of error handling functions and their meaning.

Function	Meaning
file.good();	File stream is good; no unusual condition or error has occurred
file.eof();	An end-of-file condition has been encountered. Warning: modern ANSI/ISO C++ IOstream libraries don't set the flag that this function returns.
file.fail();	An error has occurred; includes fatal errors covered by bad();
file.bad();	A fatal error has occurred.

Generally, a fatal error is one that can't be corrected by retrying the operation. For example, a disk error during a read or write. A non-fatal error is one that may be correctable by retrying the operation, usually with some argument changed.

For example, if you want to create a file only if it doesn't already exist, you might first try to open it in ios::in mode to see if it does. A true from file.fail() is the desired result; it tells you there is no file that you'd be overwriting. You would then open the file in ios::out mode, after a file.clear() to clear the error flag.

A word of caution: when the internal flags are set, they remain set until they are explicitly cleared. Closing the file doesn't clear the flags, only this function does: file.clear();

13.7 Command-Line Parameters

Many operating systems, such as DOS and UNIX, enable the user to pass parameters to your program when the program starts. These are called command-line options, and are typically separated by spaces on the command line. For example:

```
SomeProgram Param1 Param2 Param3
```

These parameters are not passed to main() directly. Instead, every program's main() function is passed two parameters. The first is an integer count of the number of arguments on the command line. The program name itself is counted, so every program has at least one parameter. The example command line shown previously has four. (The name SomeProgram plus the three parameters make a total of four command-line parameters/arguments.)

The second parameter passed to main() is an array of pointers to character strings. Because an array name is a constant pointer to the first element of the array, you can declare this argument to be a pointer to a pointer to char, a pointer to an array of char, or an array of arrays of char.

Typically, the first argument is called argc (argument count), but you may call it anything you like.

The second argument is often called argv (argument vector), but again this is just a convention.

It is common to test argc to ensure you've received the expected number of arguments, and to use argv to access the strings themselves. Note that argv[0] is the name of the program, and argv[1] is the first parameter to the program, represented as a string. If your program takes two numbers as arguments, you will need to translate these numbers to strings.

Here in this unit you will see how to use the standard library conversions.

Example program illustrates how to use the command-line arguments/parameters.

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

```
int main(int argc, char **argv)
{
cout << "Received " << argc << " arguments...\n";
for (int i=0; i<argc; i++)
cout << "argument " << i << ": " << argv[i] << endl;
return 0;
}
```

The following is the output of above program:

Test Program Learn Yourself C++ in 20 Days

```
Received 7 arguments...
argument 0: TestProgram.exe
argument 1: Learn
argument 2: Yourself
argument 3: C++
argument 4: in
argument 5: 20
argument 6: Days
```

The function main() declares two arguments: argc is an integer that contains the count of command-line parameters, and argv is a pointer to the array of strings. Each string in the array pointed to by argv is a command-line parameter. Here It is noted that argv could just as easily have been declared as char *argv[] or char argv[][]. It is a matter of programming style how you declare argv; even though this program declared it as a pointer to a pointer, array offsets were still used to access the individual strings. On line 4, argc is used to print the number of command-line parameters: seven in all, counting the program name itself. On lines 5 and 6, each of the command-line parameter is printed, passing the null-terminated strings to cout by indexing into the array of strings.

13.8 Summary

In this unit, you have learnt about file input output facilities of C++ are implemented through a component header file fstream of standard library. The fstream library predefines a set of operations for handling file related input output. This unit also enables you to know how a file is created, data is written to and retrieved from a file. In addition, you have also seen how to perform the basic operations on text/binary files are: reading/writing, reading and manipulation of data stored on these files. Both types of file need to be open and close. Here the opening and closing of files with both of the ways including various file modes have been discussed.

The file pointers and their manipulation are also explained with the several input output functions. The eof() function determines the end-of-file by returning true for end of file otherwise returning false. The concept of sequential I/O with files is also described with the useful binary input output functions. C++ supports features for writing to and reading from the disk structure or class objects directly. The writing and reading data in binary format using write() and read() functions have also been included. Sometimes during file operations, errors may also creep in. The several error handling functions supported by **ios** are summarized. Finally the command line parameters are also discussed.

13.9 Self Assessment Questions

1. Fill in the blanks:

   (a) A ____________ is a data flow from a source to a sink.

   (b) ifstream is an ___________ stream class.

   (c) The function_____ writes a character at a time in a file.

   (d) The function ______reads a line at a time from a file.

   (e) The term eof is an acronym of ___________

2. State True or False of the following:

   (a) The open() function associates a file with a device.

   (b) fstream class supports both input and output on a file.

   (c) The get() function stops at a blank white getting a string from a file.

   (d) The eof() function returns zero when it detects an end of file.

   (e) The function read() is used to read an object form a file.

3. What are the file modes? Explain in brief.

4. Explain the following member functions:

   (i) eof() (ii) write() (iii) read() (iv) close()

5. How are binary files different text files in C++?

6. What is the difference between get() and getline() functions?

7. What is a Pointer? Explain the following functions:

   (i) seekg() (ii) seekp() (iii) tellg() (iv) tellp()

8. Write a C++ program that prints a text file on the printer.

9. Write a function in C++ to count the number of lowercase alphabets present in a text file "BOOK.TXT".

10. Following is the structure of each record in a data file named "COLONY.DAT"

```
struct COLONY
{ char Colony_Code[10] ;
char Colony_Name[10]
int No_of_People ;
} ;
```

Write a function in C++ to update the file with a new value of No_of_People. The value of Colony_Code and No_of_People are read during the execution of the program.

Unit-14 : Basics of Exception Handling & Templates

Structure of Unit:

14.0 Objectives

After going through this unit you should be able to:

- Understand about the Exception.
- Understand Exception handling Mechanism & its purpose
- Understand the throwing mechanism
- Understand the catching mechanism when exception occurs
- Understand the templets and its uses.

14.1 Introduction to a Exception Handling

We know that it is very rare that a program works correctly, first time. The two most common types of bugs are logical errors and syntactional errors. A logic error is a bug in a program that causes it to operate incorrectly, but not to terminate abnormally. A syntactical error refers to an error in the syntax of a characters or tokens that is intended to be written in a particular programming language. We often come across some peculiar (unusable) other than these errors, they are known as exceptions. C++ exception handling provides a type safe, integrated approach, for coping with unusual problems.

Exception handling is a mechanism that allows two separately developed program components to

communicate when a program anomaly, called an exception, is encountered during the execution of the program. Some exceptional condition are given below :

- Division by zero.
- Array index out of range.
- Unable to allocate memory.

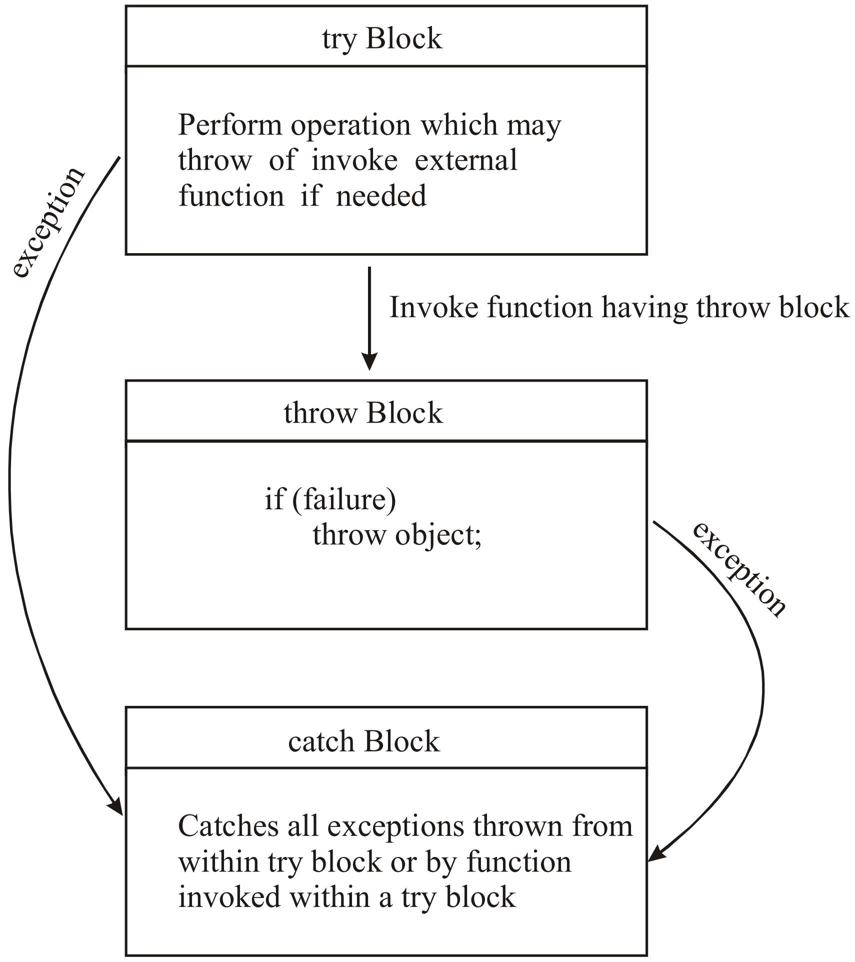
- Arithmetic overflow such as the result exceeding the range
- Unexpected arguments
- Array index overflow under flow
- Free store exhausted
- File not found

Exception are if two kinds; namely synchronous exceptions and asymchronous exceptions. The exception which occur during program execution due to some fault in input data or technique that is not suitable to handle the current class of data are known as synchronous exception, such as "out-of-range index" and "overflow". The exception caused by events or faults unrelated to the program or beyond the control of the program are dcalled asymchronous exceptions, such as 'keyboard interrupts" and "disk failure".

In this chapter we first look at how to raise, or throw, an exception at the location where the program anomaly is encountered. We then look at how to associate handlers, or catch clauses, with a set of program statements using a try block, and we look at how exceptions are handled by catch clauses.

14.2 Exception Handling Mechanism

When program encounters an abnormal situation for which it is not designed, the user may transfer control to some other part of the program that is designed to deal with the problem. This is done by throwing an exception. The exception-handling mechanism uses three blocks :try, throw, and catch.


try Block
Perform operation which may throw of invoke external function if needed
Invoke function having throw block
throw Block
if (failure)
throw object;
catch Block
Catches all exceptions thrown from within try block or by function invoked within a try block
exception
exception

Exception Handling Model

The keyword **try** is used to block of statements (surrounded by braces) which may generate exceptions. This block of statement is known as try-block. When as exception is detected, it is thrown using a **throw** statement in the try block. A **catch** block defined by the keyword catch catches the exception 'thrown' by the throw statement in the try block and handle it appropriately.

The general form of these two block are :

```
try
{
        ...
        ...
        throw exception( )
        ...
        ...
}
catch (exception e)
{
        ...
        ...
}
```

When the try block throws as exception, the program control leaves the try block and enters the catch statement of the catch block. If the type of object thrown matches the argument type in the catch statement, then catch block is executed for handling the exception. If the argument does not match, the program will be aborted with the help of the abort () function which is invoked by default. When no exception detected and thrown, the control goes to the statment immediately after the catch block. That is the catch block is skipped.

Example program to illustrate the simple try-catch mechanism.

```
# include <iostream.h>
#include<iomanip.h>
#include <conio.h>
void main ()
{
int a, b, c;
cout << "Enter the values of a and b : " ;
cin >> a >> b;
try
{
 if (b = = 0)
throw b;
c = a/b;
cout << "Value of a/b = " << c << endl;
}
catch (int i)
cout << "Exception: divide by zero caught ..." << endl;
}
getch ();
}
```

The following is the output of the above program:

```
Enter the values of a and b : 10 S
Value of a/b = 2
Enter the values of a and b : 5 0
Exception : divide by zero caught...
```

Most often, exception are thrown by functions that are invoked from within the try blocks. The point at which the throw is executed is called the throw point. Once an exception is thrown to the catch block, control cannot return to the throw point.

14.3 Throwing Mechanism

When an exception that is desired to be handled is detected, it is thrown using the throw statement in one of the following forms:

```
throw (exception);
throw exception;
throw;
//used or rethrowing an exception
```

The operand object exception may be of any type, including constants. It is also possible to throw objects not intended for error handling. When exception is thrown, it will be caught by the catch statement associated with the try block. That is, the control exists the current try block, and is transferred to the catch block after that try block. The throw point can be in a deeply nested scope within a try block or in a deeply nested function call. In any case, control is transferred to the catch statement.

14.4 Catching Mechanism

The code for handling exceptions is included in a catch block. A catch block looks like a function definition and is of the following form:

```
catch (type argument)
{
        // Statements for handling exceptions
}
```

The type indicates the type of exception that a catch block handles and argument is the argument's name. The exception-handling code should be placed between the two braces. The catch statement catches an exception whose type matches with the type of catch argument. When it is caught, the code in the catch block is executed.

If the argument in the catch definition is given a name, then the argument can be used in the exception handling code. After executing the handler, the control goes to the statement immediately following the catch block. Due to a mismatch, if an exception is not caught, an abnormal program termination may occur or catch block is simply skipped.

Sometimes, it happens that a program segment has more than one condition to throw an exception. In such a situation, we can associate more than one catch statement with a try, as shown below:

```
try {
        // try block;
}
catch (type-1 argument-1){
        //catch block-1
}
catch (type-2 argument-2){
        //catch block-2
}
```

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

Example program to illustrate multiple catch statements execution

```
#include <iostream.h>
void test (int a)
{
        try     {
                if (a = = 1)
                        throw a;
                else    if (a = = 0)
                throw 'a';
                else if (a = = -1)
                        throw 1. 0;
                cout << "n End of try-block\n";
        }
        catch (char c)          //catch 1
        {
                cout <<"Caught a character\n";
        }
        catch (int m)              //catch 2
        {
                cout<<"Caught an integer\n";
        }
        catch(double d)            //catch 3
        {
                cout <<"Caught a double \n";
        }
        cout <<"n End of try-catch block\n";
}
void main()
{       cout <<"Testing multiple catches\n";
        cout <<"a == 1\n";
        test(1);            \
        cout <<"a = =0\n";
        test(0);
        cout <<"a ==-1\n";
        test(-1);
        cout <<"a ==2\n";
        test(2);
}
```

The following is the output of the above program.

```
Testing Multiple Catches
a== 1
Caught an integer
End of try-catch block
a== 0
Caught a character
End of try-catch block
a == -1
```

```
CAught a double
End of try-catch block
a ==5
End of try-block
End of try-catch block
```

When the try block does not throw any exception and then it completes normal execution.

14.5 Rethrowing Exception

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

```
throw;
```

This causes the current exception to be thrown to the next enclosing try/catch sequence and is caught by a catch statement listed after that enclosing try block. The following program demonstrates how an exception is re-thrown and caught.

Example : Program to illustrate re-throwing of exception

```
#include < iostream.h>
void divide (int x, double y)
{
        cout <<"Inside function\n";
        try     {
                if(y == 0.0)
                        throw y;
                else
                        cout <<"Division = "<< x/y<<"\n";
        }
        catch(double)   {
                cout <<"Caught double inside function\n";
                throw;
        }
        cout <<"END OF FUNCTION\ n";
}
void main()
{
        cout<<"Inside main\n";
        try     {
                divide (10.5, 2.0);
                divide (20.5, 0.0);
          }
          catch (double)   {
                cout<<"Caught doubling inside main\n";
          }
          cout <<"End of main\n";
}
```

The following is the output of the above program:

```
Inside main
Inside function
Division - 5
  END OF FUNCTION
Inside function
Caught double inside function
Caught doubling inside main
End of main
```

When an exception is re-thrown, it will not be caught by the same catch statement or any other catch in that group. Rather, it will be caught by an appropriate catch in the outer try/catch sequence only. A catch handler itself may detect and throw an exception. Here again, the exception thrown will not be caught by any catch statements in that group. It will be passed on to the next outer try/catch sequence for processing.

14.6 Exception Objects

The exception declaration of a catch clause can be either a type declaration or an object declaration. When should the exception declaration in a catch clause declare an object? An object should be declared when it is necessary to obtain the value or manipulate the exception object created by the throw expression. If we design our exception classes to store information in the exception object when the exception is thrown and if the exception declaration of the catch clause declares an object, the statements within the catch clause can use this object to refer to the information stored by the throw expression.

14.7 Introduction to Templates

Templates is one of the feature, which enable us to define generic classes and functions. These generic classes and functions are not tied to specific implementation types. A template can be considered as a kind of macro.

Function templates and **class templates** enable programmers to specify, with a single code segment, an entire range of related (overloaded) functions called **function-template specializations**or an entire range of related classes called **class-template specializations**. This technique is called **generic programming**.

We might write a single function template for an array-sort function, then have C++ generate separate function-template specializations that will sort int arrays, float arrays, string arrays and so on.

We might write a single class template for a stack class, then have C++ generate separate class-template specializations, such as a stack-of-int class, a stack-of-float class, a stack-of-string class and so on. A class tamplate defines a parametrized type. A parametrized type is a data type defined in terms of other data type. A function template define an algorithm. Analgorithm is a generic recipe for accomplishing a task independent of the particular data types used for its implementation.

It is noted that the distinction between templates and template specializations: Function templates and class 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 colors.

14.8 Function Templates

Overloaded functions normally perform similar or identical operations on different types of data. If the operations are identical for each type, they can be expressed more compactly and conveniently using function templates. Initially, the programmer writes a single function-template definition. Based on the argument types provided explicitly or inferred from calls to this function, the compiler generates separate object-code functions (i.e., function-template specializations) to handle each function call appropriately. Function templates provide a compact solution, like macros, but enable full type checking.

All **function-template definitions** begin with keyword **template** followed by a list of **template parameters** to the function template enclosed in **angle brackets** (< and >). The general format of a function template is :

```
template< typename T >

or

template< class ElementType >

or

template < class T>

return type functionname (arguments of type T)

{

        //.......


        // Body of function with type T

        //.......

      }
```

We must use the template argument (parameter) T as and when necessary in the function body and in its argument list.

Example

```
template < class T>

void swap (T&x, T&y)

     {

        T temp =x
```

```
            x = y;
            y= temp;
        }
```

14.9 Class Templates

A class template definition (or declaration) is always preceeded by a template clause. The general format of a class template is:

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

Example

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

```
 {
   T sum = 0;
   for (int i =0; i< size; i++)
       sum + = this –> v[i] *y –v[i];
       return sum;
      }
};
```

14.10 Summary

Exception handling wires error handling directly into the programming language and sometimes even the operating system. An exception is an object that is “thrown” from the site of the error and can be “caught” by an appropriate exception handler designed to handle that particular type of error. It’s as if exception handling is a different, parallel path of execution that can be taken when things go wrong. And because it uses a separate execution path, it doesn’t need to interfere with your normally-executing code. This makes that code simpler to write since you aren’t constantly forced to check for errors. In addition, a thrown exception is unlike an error value that’s returned from a function or a flag that’s set by a function in order to indicate an error condition – these can be ignored. An exception cannot be ignored so it’s guaranteed to be dealt with at some point. A template is a one of the feature added to C++. A template can be used to create a family of classes or function.

14.11 Self Assessment Questions

1. What is an exception?
2. How is an exception handled in C++?
3. When should a program throw an exception?
4. When is a catch (....) hander is used?
5. What should be placed inside a try and catch block?
6. What is generic programming?
7. Distinguished between the term class template and function template.

Appendix A : References

- K R Venugopal, Rajkumar, T Ravishankar, Mastering C++, Tata McGraw Hill.
- E. Balagurusamy, Object Oriented Programming with C++, Tata McGraw Hill.
- Let us C++, Yashwant Kanetkar, BPB Publications.
- Jagadev, Rath & Dehuri, Object-Oriented Programming Using C++, Prentice Hall India.
- Herbert Schildt, The Complete Reference C++, Tata McGraw Hill.
- Robert Lafore, Object Oriented Programming in Turbo C++, Galgotia Publications.
- Eric S. Roberts and Julie Zelenski, Programming Abstraction in C, Addison-Wesley.
- Brain Overland, C++ Without Fear, Second Edition, Prentice Hall.
- Scot Robert Ladd C++ Components and Algorithms, BPB Publications.
- B. Chandra, Object-Oriented Programming Using C++, Alpha Science International.
- Jaspreet Singh & Pinkiparampreet Kaur Object Oriented Programming Using C++, Technical Publication Pune.
- Ira Pohl, Object-Oriented Programming Using C++, 2/E, Pearson Education India
- Poornachandra Sarang, Object Oriented Programming with C++, Prentice Hall India.