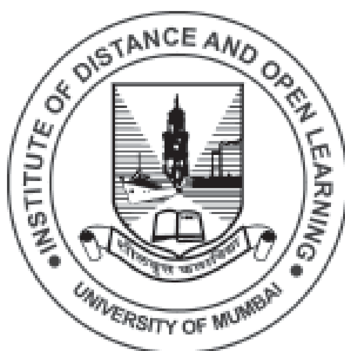


INSTITUTE OF DISTANCE AND OPEN LEARNING
University of Mumbai

F.Y.B.Sc. (I.T)

SEMESTER - II

OBJECT ORIENTED PROGRAMMING

SUBJECT CODE : USIT 201

© UNIVERSITY OF MUMBAI

Prof. Suhas Pednekar
Vice Chancellor
University of Mumbai

Prof. Ravindra D. Kulkarni
Pro Vice-Chancellor,
Univeristy of Mumbai

Prof. Prakash Mahanwar
Director,
IDOL, University of Mumbai

Programe Co-ordinator : **Shri Mandar Bhanushe**
Head, Faculty of Science and Technology IDOL
University of Mumbai - 400098

Course Co-ordinator : **Gouri S. Sawant**
Assistant Professor, B.Sc. IT, IDOL
University of Mumbai - 400098

Course Writers : **Abhijit Kale**
Assistant Prof. VPM's B.N.Bandodkar college of Science

: **Vipul Chavan**
Assistant Prof. VPM's B.N.Bandodkar college of Science

: **Tejas Jadhav**
Assistant Prof. VPM's B.N.Bandodkar college of Science

: **Vishal Khandekar [Visiting faculty]**
K.J. Somaiya Institute of Management,
Mumbai K.J.Somaiya

: **Inumarthi V Srinivas**
Assistant Prof. K.J. Somaiya Institute of Management,
Mumbai

April, 2021 Print - 1

Published by : **Director**
Institute of Distance and Open Learning,
University of Mumbai,
Vidyanagari, Mumbai - 400 098.

DTP Composed by : Mumbai University Press
University of Mumbai, Vidyanagari,
Santacruz (E.), Mumbai - 400 098.

Printed by :

CONTENTS

Chapter No.	Title	Page No.
	Unit 1	
1.	Objects Oriented Methodology	1
2.	Principles of Oops	10
	Unit 2	
3.	Classes and Objects	17
4.	Constructors and Destructors	58
	Unit 3	
5.	Polymorphism	64
6.	Virtual Functions	80
	Unit 4	
7.	Inheritance	89
8.	Exception Handling	114
	Unit 5	
9.	Templates	128
10.	Working with Files	137

OBJECT ORIENTED PROGRAMMING

F.Y.B.Sc. (I.T)
SEMESTER - II

SYLLABUS

Unit	Details	Lectures
I	**Object Oriented Methodology:** Introduction, Advantages and Disadvantages of Procedure Oriented Languages, what is Object Oriented? What is Object Oriented Development? Object Oriented Themes, Benefits and Application of OOPS. **Principles of OOPS:** OOPS Paradigm, Basic Concepts of OOPS: Objects, Classes, Data Abstraction and Data Encapsulation, Inheritance, Polymorphism, Dynamic Binding, Message Passing	12
II	**Classes and Objects:** Simple classes (Class specification, class members accessing), Defining member functions, passing object as an argument, Returning object from functions, friend classes, Pointer to object, Array of pointer to object. **Constructors and Destructors:** Introduction, Default Constructor, Parameterized Constructor and examples, Destructors	12
III	**Polymorphism:** Concept of function overloading, overloaded operators, overloading unary and binary operators, overloading comparison operator, overloading arithmetic assignment operator, Data Conversion between objects and basic types, **Virtual Functions:** Introduction and need, Pure Virtual Functions, Static Functions, this Pointer, abstract classes, virtual destructors.	12
IV	**Program development using Inheritance:** Introduction, understanding inheritance, Advantages provided by inheritance, choosing the access specifier, Derived class declaration, derived class constructors, class hierarchies, multiple inheritance, multilevel inheritance, containership, hybrid inheritance. **Exception Handling:** Introduction, Exception Handling Mechanism, Concept of throw & catch with example	12
V	**Templates:** Introduction, Function Template and examples, Class Template and examples. **Working with Files:** Introduction, File Operations, Various File Modes, File Pointer and their Manipulation	12

Books and References:					
Sr. No.	Title	Author/s	Publisher	Edition	Year
1.	Object Oriented Analysis and Design	Timothy Budd	TMH	3rd	2012
2.	Mastering C++	K R Venugopal, Rajkumar Buyya, T Ravishankar	Tata McGraw Hill	2nd Edition	2011
3.	C++ for beginners	B. M. Hirwani	SPD		2013
4.	Effective Modern C++	Scott Meyers	SPD		
5.	Object Oriented Programming with C++	E. Balagurusamy	Tata McGraw Hill	4th	-
6.	Learning Python	Mark Lutz	O' Reilly	5th	2013
7.	Mastering Object Oriented Python	Steven F. Lott	Pact Publishing		2014

UNIT – I

1

OBJECTS ORIENTED METHODOLOGY

Chapter Structure :

1.0 Program and Programming
1.1 Programming Languages
1.2 Procedure Oriented Programming
1.3 Need of Object Oriented Programming
1.4 Object Oriented Programming
1.5 Comparison of Procedural and Object Oriented Approach
1.6 Benefits of OOP
1.7 Advantages of OOPs
1.8 Object Oriented Languages
1.9 Applications of OOPS
1.10 Question

1.0 Program and Programming

- Components of any computer system are hardware and software.
- Both hardware and software have their own sets of functionalities which can be interdependent or independent of each other.
- A computer system is designed to produce the desired results by making the functionalities of both the hardware and the software to converge.
- The hardware is what we can see, touch and feel e.g. keyboard, mouse, visual display units like monitors, printers etc.
- The software can be modified / replaced with lesser effort, time and money.
- Computer is essentially a data processing machine which requires two kinds of inputs for its operations and these are: data and instructions.
- The hardware of a computer can not produce the desired results unless it is given the requisite instructions and data by the user(s).
- Do you know what is software all about? The software deals with the instructions.
- The examples of software are Microsoft Office, Microsoft Windows 7, Red Hat Linux, Railways Reservation System, Microsoft Internet Explorer, Google Search Engine etc.
- A program as an independent entity or as part of a software is intended to instruct the hardware to carry out specific task(s) to the satisfaction of the user(s) by providing specific outcomes. So how do you define a program?

- A program can be defined to be a set of instructions written in a programming language which are given in a fixed sequence to the hardware of a specific computer and executed by its hardware to produce predetermined and expected outcomes.
- The instructions in a program are written mostly in natural languages (e.g. English, Hindi, French, German, and Chinese etc.) following the syntax (form) and semantics (meaning) of the programming language chosen for writing the program.
- There are a variety of programming languages available for writing the programs e.g. BASIC, C, C++, Java, Prolog, Lisp, HTML, PHP etc. The sequence of instructions is very important because if the sequence is not correct, the expected results cannot be achieved by the program.
- Now, let us see what does programming mean to us? The meaning of the term programming (or computer programming) has been changing rapidly since the idea of a first program was envisaged.
- Initially the computers were used to solve the mathematical problems with the help of calculations.
- Hartee in 1950 suggested that the process of preparing a calculation for a machine can be broken down into two parts, ‘programming’ and ‘coding’. He described programming as the process of drawing up the schedule of the sequence of individual operations required to carry out the calculation.
- In 1958, Booth proposed that the process of organizing a calculation can be divided into two parts, a) the mathematical formulation, and b) the actual programming. With the passage of time, the definition of programming has kept on evolving and at present programming is considered to be the process of writing programs and may include activities as diverse as designing, writing, testing, debugging and maintaining the code of a program.

In normal conversation, programming is described as the process of instructing the computer to do something desired and useful for the user with the help of a programming language.

1.1 Programming Languages

- The programming languages are created by, we, human beings.
- These languages are used to communicate instructions to the machines especially computers so that the programs can control the behaviour of the hardware of the machines to get desired results.
- Basically, the hardware of the computers understands only the language of the hardware which is called the machine language.
- The hardware is unable to understand and decipher any program written in any other programming language.

- Every type of a CPU has its own machine language.
- Therefore, in order to make the hardware of a computer understand the instructions contained in a program written in any other programming language, a mechanism called ‘translator’ is required.
- This translator converts the program written in programming languages other than the native machine language of the CPU (hardware) into the native machine language of a particular CPU on which this program is intended to be executed.
- Every programming language must have its own translator for the programs written in it to be executed or run on the computer hardware.
- Various types of translators available can be categorized into assemblers, interpreters or compilers.
- The primitive or the first generation of programming languages were called machine languages and the symbols like ‘0’ and ‘1’ were used to write programs under this category of programming languages.
- The second generation of programming languages were called the assembly languages and mainly used mnemonics to construct a program.
- The third generation of programming languages was called high level languages as these programming languages were independent of the CPU of the hardware being used and the instructions written in the programs were just like the instructions given in natural languages.
- The third generation languages are known as 3GL languages.
- The current generation of the programming languages are called the fourth generation languages or the 4GLs. These languages represent the class of programming languages that are closest to the human (natural) languages.
- Based on the intended use of domain of use, the programming languages are broadly classified as imperative programming languages where imperative sentences are used in a program to issue commands in terms of instructions; and declarative programming languages where declarative instructions are used in a program to assert the desired result.
- More common paradigm classifies these languages into imperative, functional, logic programming and object-oriented languages.

Following Table presents the summary of main features of these programming paradigms.

Paradigm	Key Concepts	Key Concepts	Program Execution	Result
Functional	Function	Collection of functions	Evaluation of functions	Value of the main function
Imperative	Command (instruction)	Sequence of commands	Execution of commands	Final state of computer memory
Logic	Predicate	Logic formulas: axioms & a theorem	Logic proving of the theorem	Failure or Success of proving
Object-oriented	Object	Collection of classes of objects	Exchange of messages between the objects	Final state of the objects

1.2 Procedure Oriented Programming

- Conventional programming language using high level language such as Cobol and C, is commonly known as Procedure Oriented Programming (POP).
- In Procedure Oriented Programming, the problem is viewed as a sequence of a thing to be done.
- The primary focus is on functions.
- Procedure Oriented Programming basically consists of writing a list of instructions for the computer to follow, and organizing these functions into groups known as functions.
- In a multi-function program, many important data items are placed as global so that they may be accessed by all the functions.
- Each function may have its own local data.

Some characteristics of Procedure Oriented Programming are:

1. Large programs are divided into smaller programs known as functions.
2. Most of the functions share global data.
3. Data move openly around the system from function to function.
4. Functions transform data from one form to another.
5. Employs top-down approach in program design.

1.3 Need of Object Oriented Programming

- Developments in software technology continue to be dynamic. New tools and techniques are announced in quick succession.
- This has forced the software engineers and industry to continuously look for new approaches to software design and development, and they are becoming more and more critical in view of the increasing complexity of software system as well as the highly competitive nature of the industry.
- This rapid advance appears to have created a situation of crises within the industry.

The following issues needs to be solved:

- To represent real life entities of problems in system design.
- To design system with open interface.
- To ensure reusability and extensibility of modules.
- To develop modules that is tolerant to any changes in future
- To improve software productivity and decrease cost.
- To improve the quality of the software.
- To manage time schedule.
- To industrialize the software development process.

To overcome these problems and major motivation factor in the invention of object oriented approach is to remove some of the flows encountered in the procedure oriented approach.

1.4 Object Oriented Programming

Object Oriented Programming is the most recent concept among programming model. The motivating factor in the invention of object oriented approach is to remove some of the flows encountered in the procedural approach. OOPS treats data as a critical element in the program development and does not allow it to flow freely around the system. It binds data more closely to the functions that operate on it, and protects it from accidental modification from outside functions.

Some of the striking features of Object Oriented Programming are:

- Importance on data rather than procedure.
- Programs are divided into what are known as objects.

Data structures are designed as such that they characterize the objects.

Introduction to OOPs

- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- Follow bottom-up approach in program design.

1.5 Comparison of Procedural and Object Oriented Approach

There are two different approaches to write a program, i.e., Procedure Oriented Programming and Object Oriented Programming. Basic aim of these methods is nothing but to make programming efficient. We can write the program using any of the way but there are notable differences between both approaches.

Sr. No	Procedure Oriented Programming	Object Oriented Programming
	POP is divided into small parts called as functions	In OOP, program is divided into parts called objects
	In POP, Importance is not given to data but to functions as well as sequence of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a real world.
	POP follows top-down approach.	OOP follows bottom-up approach.
	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
	In POP, data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
	In POP, most function uses global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
	POP does not have any proper way for hiding data so it is less secure.	OOP provides data hiding so provides more security.
	In POP, overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
	Example of POP are: C, VB, FORTRAN, Pascal.	Example of OOP are: C++, JAVA, VB.NET, C#.NET.

1.6 Benefits of OOP

1) As OOP is closer to the real world phenomena, hence, it is easier to map real world problems onto a solution in OOP.
2) The objects in OOP have the state and behaviour that is similar to the real world objects.
3) It is more suitable for large projects.
4) The projects executed using OOP techniques are more reliable.
5) It provides the bases for increased testability (automated testing) and hence higher quality.
6) Abstraction techniques are used to hide the unnecessary details and focus is only on the relevant part of the problem and solution.
7) Encapsulation helps in concentrating the structure as well as the behaviour of various objects in OOP in a single enclosure.
8) The enclosure is also used to hide the information and to allow strictly controlled access to the structure as well as the behaviour of the objects.
9) OOP divides the problems into collection of objects to provide services for solving a particular problem.
10) Object oriented systems are easier to upgrade/modify.
11) The concepts like inheritance and polymorphism provide the extensibility of the OOP languages.
12) The concepts of OOP also enhance the reusability of the code written.
13) Software complexity can be better managed.
14) The use of the concept of message passing for communication among the objects makes the interface description with external system much simpler.
15) The maintainability of the programs or the software is increased manifold. If designed correctly, any tier of the application can be replaced by another provided the replaced tier implements the correct interface(s). The application will still work properly.

1.7 Advantages of OOPs

1. Code reusability in terms of inheritance.
2. Object-oriented system can be easily upgraded from one platform to another.
3. Complex projects can be easily divided into small code functions.
4. The principle of abstraction and encapsulation enables a programmer to build secure programs.
5. Software complexity decreases.
6. Principle of data hiding helps programmer to design and develop safe programs.
7. Rapid development of software can be done is short span of time.
8. More than one instance of same class can exist together without any interference.

1.8 Object Oriented Languages

Some of the most popular Object-oriented Programming languages are :

a) C++

b) Ruby

c) Java.

d) Delphi

e) smalltalk

f) Charm++

g) Eiffle.

h) Simula.

1.9 Applications of OOPS

Object concept helps to translate our thoughts to a program. It provides a way of solving a problem in the same way as a human being perceives a real world problem and finds out the solution. It is possible to construct large reusable components using object-oriented techniques. Development of reusable components is rapidly growing in commercial software industries. If there is complexity in software development, object-oriented programming is the best

paradigm to solve the problem. The following areas make the use of OOP:

1. Image Processing
2. Pattern Recognition
3. Computer Assisted Concurrent Engineering
4. Computer Aided Design and Manufacturing
5. Computer Aided Teaching
6. Intelligent Systems
7. Database Management Systems
8. Web-based Applications
9. Distributed Computing and Applications
10. Component-based Applications
11. Business Process Re-engineering
12. Enterprise Resource Planning
13. Data security and management
14. Mobile Computing
15. Data Warehouse and Data Mining
16. Parallel Computing

1.10 Questions

Q1. Why does hardware not provide flexibility of operations to the user ?

Q2. How is a program related to software?

Q3. What is the signature of a method?

Q4. Differentiate between information hiding and encapsulation.

Q5. What is the difference between object-oriented and object-based programming languages?

Q6. What is the need of Object Oriented Programming ?

Q7. Differentiate Procedure Oriented Programming & Object Oriented Programming

Q8. List and explain benefits of OOP .

Q9. List and explain application of OOPS

*****

2

PRINCIPLES OF OOPS

Chapter Structure :

2.0 Object Oriented Programming Paradigm
2.1 Characteristics of OOPS / Concepts of OOPS
- 2.1.0 Class
- 2.1.1 Object
- 2.1.2 Polymorphism
- 2.1.3 Inheritance
- 2.1.4 Reusability
- 2.1.5 Data abstraction and encapsulation
- 2.1.6 Dynamic Binding
- 2.1.6 Message Passing

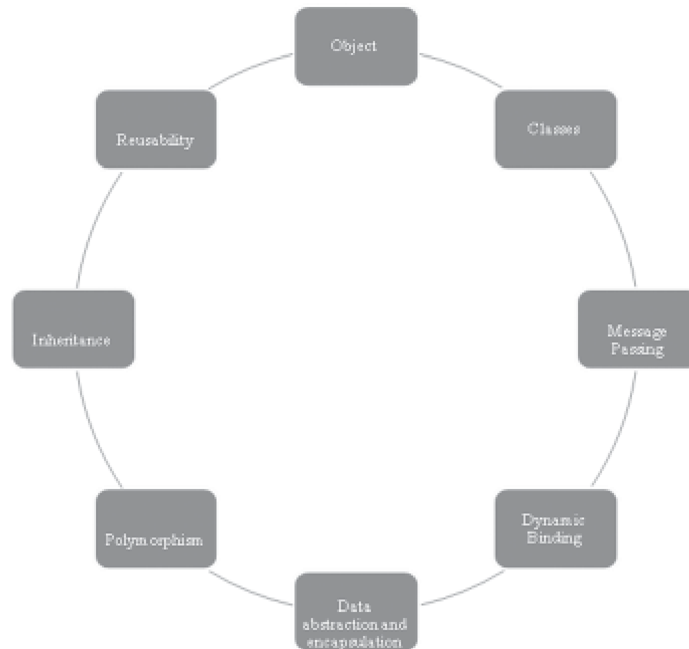
2.2 Summary
2.3 Further Readings
2.4 Web Reference
2.5 Questions

2.0 Object Oriented Programming Paradigm

- Simula was the first programming language developed in the mid-1960s to support the object-oriented programming paradigm followed by Smalltalk in the mid-1970s that is known to be the first 'pure' object-oriented language.
- Eiffel, Java, C++, Object Pascal, Visual Basic, C# etc are the other OOP languages that came into existence later on, all having different complexities of syntax and dynamic semantics.
- The main motive of the developers of programming languages over the years has always been to create such programming languages that are close to human (i.e. natural) languages.
- The way we perceive and interact with the things in our day-to-day lives, the representation of programming constructs should closely match the same. Hence came into existence the concept of 'objects' and 'object-oriented programming paradigm'.
- Every object has certain defining properties which distinguish it not only from different types of other objects but from the similar types of objects too.
- Every object has certain functions associated with it and all similar types of objects are supposed to support these. Although, Some of these functions can be the same as associated with different types of objects.

- Example with Explanation : In case of a ball pen, one of the functions associated with each type of ball pen object, is to write and another associated function is the provision to hold it in hands conveniently. All the ball pen objects support both these functions. Incidentally, all the tooth brush objects also support the function of holding them in the hands conveniently but, in addition, support other functions like brushing the teeth too.
- This concept of objects borrowed from the real world has been the basis of the object-oriented programming (OOP) paradigm and this paradigm is a direct consequence of an effort to have a programming language closely matching the human behaviour.
- This OOP paradigm is all about creating program(s) dealing with objects where these objects interact with one another to achieve the overall objectives of the program. Every object in the programs has certain defining properties called attributes (or instance variables) possessing supporting values for each of the attributes and some associated functions (normally called methods or operations). As in the real world objects, no two objects in a program can have the same values of all the attributes.
- At times, instead of dealing with individual objects, it is convenient to talk collectively about a group of similar objects where all the objects of this group will have the same set of attributes and methods.
- In the object-oriented programming paradigm parlance, this collection of objects corresponding to a particular group is known as a class.
- All programs under this paradigm contain a description of the structure (corresponding to attributes) and behaviour (corresponding to methods) of so called classes.
- In a program, various objects are created from these classes. The process of creation of an object from a class is called ‘instantiation’ and the object created is known as an instance of the class.
- Every object created will have a ‘state’ associated with the description of the structure in the class from which it has been instantiated.
- The state of an object is defined by the set of values assigned to its corresponding attributes (and stored in the memory) of the object.
- A class is defined to be a template or a prototype so that a collection of attributes and methods can be described within it and this definition can be used for creating different objects within a program.
- It is this concept of encapsulating the data and methods within the objects that provides the programmers with flexibility within the OOP paradigm because an object can be extended or modified without making changes to its external interface or other classes/objects in the program.
- Various classes may exhibit features like inheritance and polymorphism of methods.

2.1 Characteristics of OOPS / Concepts of OOPS



Characteristics of OOPs are :

- Object
- Classes
- Polymorphism
- Inheritance
- Reusability
- Data abstraction and encapsulation
- Dynamic Binding
- Message Passing

2.1.0 Class

- A group of objects that share common properties for data part and some program part are collectively called as class.
- In C ++ a class is a new data type that contains member variables and member functions that operate on the variables.
- The most remarkable feature of C + + is a class. The class binds together data and methods which work on data. The class is an abstract data type (ADI) so creation of class simply creates a template.
- The class is a keyword.
- The general syntax of creating a class in C++ is given below:

```
class class name
{
public :
        data & function;
private :
        data & function;
protected :
        data & function;
};
```

Following this keyword class, class_name represents name of the class. The class name must obey rules of writing identifier as class_name is nothing but an identifier. The class is opened by opening brace} and closed by closing brace}. The class definition/declaration must end with semicolon. Inside the class we define the data members and member functions. They may be defined either public, private or jn protected mode.

There are three different types of mode :

1. public
2. private
3. protected.

2.1.1 Object

- Objects are the basic run time entities in an Object Oriented System.
- They may represent a person, a bank account, a table of data or any item that the program has to handle.
- They may also represent user-defined data such as vector, time, and lists.
- When a program is executed, the objects interact by sending messages to one another.
- Creating objects is similar to declaring variables.

Syntax:

<Class Name> <Object Name>;

For Example:

Student s1;

Where Student is a class name and s1 is its object.

2.1.2 Polymorphism

- Poly means many. Morphism means forms.
- Polymorphism feature enables classes to provide different implementation of methods having the same name.
- There are two types of polymorphism:
    1. Compile time (Overloading)
    2. Run time (Overriding)

2.1.3 Inheritance

- Inheritance is the process by which objects of one class acquire the properties of objects of another class.
- It supports the concept of hierarchical classification.
- The existing class is called as base class and a new class which is created on the basis of base class is called as derived class.
- There are 5 different types of inheritance, i.e.:
    - Single level
    - Multilevel
    - Multiple
    - Hierarchical
    - Hybrid

2.1.4 Reusability

- The concept of inheritance provides an important feature to the object-oriented languagereusability.
- A programmer can take an existing class and, without modifying it, and additional features and capabilities to it.
- This is done by deriving a new class from an existing class.

2.1.5 Data Abstraction and Encapsulation

- The wrapping up of data and functions into a single unit is known as encapsulation.
- Data encapsulation is the most striking feature of a class.
- Abstraction refers to the act of representing essential features without including the background details or explanations.
- Encapsulation is the hiding of information in order to ensure that data structures and operators are used as intended and to make the usage model more obvious to the developer.

- C++ provides the ability to define classes and functions as its primary encapsulation mechanisms.

2.1.6 Dynamic Binding

- Binding means linking.
- It is linking of function definition to a function call.
    1. If linking of function call to function definition i.e., a place where control has to be transferred is done at compile time, it is known as static binding.
    2. When linking is delayed till run time or done during the execution of the program then this type of linking is known as dynamic binding. Which function will be called in response to a function call is find out when program executes.

2.1.7 Message Passing

- In C++ objects communicate each other by passing messages to each other.
- A message contains the name of the member function and arguments to pass. In message passing shown below :

  object. method (parameters);

- Message passing here means object calling the method and passing parameters.
- Message passing is nothing but the calling the method of the class and sending parameters.
- The method is turn executes in response to a message.

2.2 Summary

- The programs are the means through which we can make the computers produce the useful desired outputs.
- Out of a variety of programming paradigms being used by practitioners as well as the researchers, the structured and the object-oriented programming paradigm and corresponding structured and object-oriented programming have been in focus for quite some time now.
- In this unit, you studied that the structured programming languages initially helped in coping with the inherent complexity of the softwares of those times but later on, were found wanting in the handing the same as far as the software of present days are concerned.
- you were introduced to the concepts of object-oriented programming paradigm and it was illustrated as to how this paradigm is closer to natural human thinking.
- It was followed by the illustrations of some more concepts of object-oriented programming like classes, objects, message passing, interface, associations, inheritance and polymorphism.
- In the end, you saw what the various benefits of OOPs are and how can these help in producing good quality software.

2.3 Further Readings

- The *C++ Programming Language* by Bjarne Stroustrup, Addison-Wesley, 3rd edition, 1997.
- *C++: The Complete Reference*, Herbert Schildt, 4th Edition, Mc Graw Hill.
- Object Oriented Analysis and Design , Author- Timothy Budd , Publisher -TMH edition - 3rd , Year-2012
- C++ for beginners , Author- B.M Hirwani , Publisher -SPD ,Year-2013

2.4 Web Reference

www.w3school.com
www.tutorialspoint.com
www.javatpoint.com

2.5 Questions

Q1. Discuss programming paradigms in detail.

Q2. Explain Characteristics of OOPs

Q3. What is class and Objects ? Explain with example.

Q4. Explain the terms a) Dynamic Binding b) Message Passing

*****

UNIT – II

3

CLASSES AND OBJECTS

Chapter Structure :

3.0 Basics of C++
3.0.0 C++ Character Set
3.0.1 White Spaces Characters
3.0.2 Tokens:
3.0.3 Identifiers
3.0.4 Keywords
3.0.5 Simple C++ Programs
3.0.6 Difference Between C AND C++
3.0.7 Data Types in C++
3.0.8 Variables
3.0.9 Literals or Constants
3.0.10 Operators in C++
3.0.11 Control Structure in C++
3.0.12 Iterative or looping statement
3.0.13 Breaking statement
- 3.1 C++ Classes and Objects
- 3.2 C++ Class Definitions
- 3.3 Define C++ Objects
- 3.4 Accessing the Data Members
- 3.5 Classes and Objects in Detail
- 3.6 Defining Class and Declaring Objects
- 3.7 Declaring Objects
- 3.8 Accessing data members and member functions
- 3.9 Member Functions in Classes
- 3.10 Passing and Returning Objects in C++
- 3.11 Returning Object (method) as argument
- 3.12 Friend Function
- 3.13 Pointer to C++ Classes
- 3.14 C++ Array of Pointers

3.0.14 Questions

3.0 Basics of C++

C++ is an object oriented programming (OOP) language. It was developed at AT&T Bell Laboratories in the early 1979s by Bjarne Stroustrup. Its initial name was C with classes, but later on in 1983 it was renamed as C++. It is a deviation from traditional procedural languages in the sense that it follows object oriented programming (OOP) approach which is quite suitable for managing large and complex programs. C++ language is an extension to C language and supports classes, inheritance, function overloading and operator overloading which were not supported by C language. In any language, there are some fundamentals you need to learn before you begin to write even the most elementary programs. This chapter includes these fundamentals; basic program constraints, variables, and Input/output formats. C++ is a superset of C language. It contains the syntax and features of C language. It contains the same control statements; the same scope and storage class rules; and even the arithmetic, logical, bitwise operators and the data types are identical. C and C++ both the languages start with main function. The object oriented feature in C++ is helpful in developing the large programs with clarity, extensibility and easy to maintain the software after sale to customers. It is helpful to map the real-world problem properly. C++ has replaced C programming language and is the basic building block of current programming languages such as Java, C# and Dot.Net etc.

3.0.0 C++ Character Set

Character set is a set of valid characters that a language can recognise. The character set of C++ is consisting of letters, digits, and special characters.
The C++ has the following character set:

Letters (Alphabets)	A-Z, a-z
Digits	0-9
Special Characters !, <	#, <=, >=, @ ,+, -, *, /, ^, \, ( ), [ ], { }, =, >. ,,, “, $, ;, :, % , &, ?, _

There are 62 letters and digits character set in C++ (26 Capital Letters + 26 Small Letters + 10 Digits) as shown above. Further, C++ is a case sensitive language, i.e. the letter A and a, are distinct in C++ object oriented programming language. There are 29, punctuation and special character set in C++ and is used for various purposes during programming.

3.0.1 White Spaces Characters

A character that is used to produce blank space when printed in C++ is called white space character. These are spaces, tabs, new-lines, and comments.

3.0.2 Tokens

A token is a group of characters that logically combine together. The programmer can write a program by using tokens. C++ uses the following types of tokens:

- Keywords
- Identifiers
- Literals
- Punctuators
- Operators

3.0.3 Identifiers

A symbolic name is generally known as an identifier. Valid identifiers are a sequence of one or more letters, digits or underscore characters (_). Neither spaces nor punctuation marks or symbols can be part of an identifier. Only letters, digits and single underscore characters are valid. In addition, variable identifiers always have to begin with a letter. In no case can they begin with a digit. Another rule for declaring identifiers is that they cannot match any keyword of the C++ programming language. The rules for the formation of identifiers can be summarised as:

A symbolic name is generally known as an identifier. Valid identifiers are a sequence of one or more letters, digits or underscore characters (_). Neither spaces nor punctuation marks or symbols can be part of an identifier. Only letters, digits and single underscore characters are valid. In addition, variable identifiers always have to begin with a letter. In no case can they begin with a digit. Another rule for declaring identifiers is that they cannot match any keyword of the C++ programming language. The rules for the formation of identifiers can be summarised as:

3.0.4 Keywords

There are some reserved words in C++ which have predefined meaning to complier called keywords. These are also known as reserved words and are always written or typed in lower cases. There are following keywords in C++ object oriented language:

List of Keywords:

asm	case	class	void
double	extern	friend	default
new	protected	return	inline
switch	try	unsigned	sizeof
auto	catch	const	volatile
else	float	goto	delete
operator	public	short	int
template	typedef	virtual	static
break	char	continue	while
enum	for	if	do
private	register	signed	long
this	union	struct	

3.0.5 SIMPLE C++ PROGRAMS

1)

```
#include <iostream.h>          // Section: 1- The include Directive
int main ()                    // Section: 3 - Main function definition
{                              // Section: 4 - Declaration of an object
cout << "Hello World!";
return 0;
}
```

2)

```
#include <iostream.h>
# include<conio.h>
main()
{
char name [15];
clrscr();
cout << "Enter your name:";
cin >> name;
cout<<"Your name is: " <<name;
return0;
}
```

3)

```
#include <iostream.h>
int main ()
{
// declaring variables:
int a, b;
int result;
// process:
a = 5;
b = 2;
a = a + 1;
result = a - b;
// print out the result:
cout << result;
// terminate the program:
return 0;
}
```

3.0.6 Difference Between C AND C++

Following are some differences between C and C ++ :
C++ is regarded as an intermediate-level language. It comprises a combination of both high-level and low-level language features. C++ is an extension to C Programming language. The difference between the two languages can be summarised as follows:

- The variable declaration in C, must occur at the top of the function block and it must be declared before any executable statement. In C++ variables can be declared anywhere in the program.
- In C++ we can change the scope of a variable by using scope resolution operator. There is no such facility in C language.
- C Language follows the top-down approach while C++ follows both top-down and bottom-up design approach.
- C is a procedure language and C++ is an object oriented language.
- C allows a maximum of 32 characters in an identifier name whereas C++ allows no limit on identifier length.
- C++ is an extension to C language and allows declaration of class, while C language does not allow this feature.
- C++ allows inheritance and polymorphism while C language does not.

3.0.7 Data Types in C++

C++ defines several data types which can be used under different programming situations like an int data type can be used to represent whole numbers as age of a person, roll number etc. or float data type can be used to represent salary of person, interest rate etc. The basic data types are as shown as follows :

1. Built-in Type
   - (a) Integral Type
     - (i) int
     - (ii) char
   - (b) Floating Types
     - (i) float
     - (ii) double
   - (c) void
   - (d) bool
   - (e) wchar_

2. User Defined Data Type
   - (a) class
   - (b) struct
   - (c) union
   - (d) enumeration

3. Derived Data Types
   - (a) array
   - (b) function
   - (c) pointer
   - (d) reference

3.0.8 Variables

A variable is a named location in memory that is used to hold a value that can be modified in the program by the instruction. All variables must be declared before they can be used. They must be declared in the beginning of the function or block (except the global variables). The general form of variable declaration is :

data type variable [list];

Here list denotes more than one variable separated by commas;

Example :

```
int a;
float b,c;
char p,q;
```

Here a is a variable of type int, b and c are variable of type float, and p, q are variables of type char, int, float and char are data types used in C. The rule for writing variables are same as for writing identifiers as a variables is nothing but an identifier. C++ allows you to declare variables anywhere in the program. That is unlike C it is not necessary to declare all the variables in the beginning of the program. You can declare wherever you want it to be declares i.e., right on the place where you want to use it. An advantage of this is that sometimes lots of variables are declared in the advance in the beginning of the program and many of them are unreferenced. Declaring variables at the place where they are actually required is handy. You do not need to declare all the variables earlier prior to their use.

On the other hand, it is burdensome to look for all the variable declared in the program as variable declaration will be scattered everywhere in the whole program.
C++ also allows you to initialize variable dynamically at the place of use. That is you can write anywhere in the program like this.

```
int x = 23;
float sal = 2345;
char name[ ] = “Hari”;
```

This is known as “Dynamic Initialization” of the variables.

3.0.9 Literals or Constants

A number which does not charge its value during execution of a program is known as a constant or literals. Any attempt to change the value of a constant will result in an error message. A keyword const is added to the declaration of an identifier to make that identifier constant. A constant in C++ can be of any of the basic data types. Let us consider the following C++ expression:

```
const float Pi = 3.1215;
```

The above declaration means that Pi is a constant of float types having a value: 3.1415.

Examples of some valid constant declarations are:

```
const int rate = 50;
const float Pi = 3.1415;
const char ch = „A ;
```

3.0.10 Operators in C++

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provide the following types of operators "

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

This chapter will examine the arithmetic, relational, logical, bitwise, assignment and other operators one by one.

Arithmetic Operators

There are following arithmetic operators supported by C++ language "
Assume variable A holds 10 and variable B holds 20, then –

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator, increases integer value by one	A++ will give 11
--	Decrement operator, decreases integer value by one	A-- will give 9

Relational Operators

There are following relational operators supported by C++ language

Assume variable A holds 10 and variable B holds 20, then –

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Logical Operators

There are following logical operators supported by C++ language.

Assume variable A holds 1 and variable B holds 0, then –

<table>
<tr><th>Operator</th><th>Description</th><th>Example</th></tr>
<tr><td>&&</td><td>Called Logical AND operator. If both the operands are non-zero, then condition becomes true.</td><td>(A && B) is false.</td></tr>
<tr><td>||</td><td>Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.</td><td>(A || B) is true.</td></tr>
<tr><td>!</td><td>Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.</td><td>!(A && B) is true.</td></tr>
</table>

Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ are as follows –

<table>
<tr><th>p</th><th>q</th><th>p & q</th><th>p | q</th><th>p ^ q</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
</table>

Assume if A = 60; and B = 13; now in binary format they will be as follows “

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

The Bitwise operators supported by C++ language are listed in the following table. Assume variable A holds 60 and variable B holds 13, then –

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of ‘flipping’ bits.	(~A) will give -61 which is 1100 0011 in 2’s complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Assignment Operators

There are following assignment operators supported by C++ language –

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Misc Operators

The following table lists some other operators that C++ supports.

Sr.No	Operator & Description
1	**size of** sizeof operator returns the size of a variable. For example, sizeof(a), where 'a' is integer, and will return 4.
2	**Condition ? X : Y** Conditional operator (?). If Condition is true then it returns value of X otherwise returns value of Y.
3	**,** Comma operator causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list.
4	**. (dot) and -> (arrow)** Member operators are used to reference individual members of classes, structures, and unions.
5	**Cast** Casting operators convert one data type to another. For example, int(2.2000) would return 2.
6	**&** Pointer operator & returns the address of a variable. For example &a; will give actual address of the variable.
7	***** Pointer operator * is pointer to a variable. For example *var; will pointer to a variable var.

3.0.11 Control Structure in C++

C++ program is usually not limited to a linear sequence of instructions but it may bifurcate, repeat code or may have to take decisions during the process of coding. For that purpose, C++ provides control structures which are used to control the flow of program. Before we discuss control structures, let us first discuss a new concept: the compound-statement or block, which is very much needed to understand well the flow of control in a program. A block is a group of statements which are separated by semicolons (;) like all C++ statements, but grouped together in a block enclosed in braces: { }: for example: { statement1; statement2; statement3; ... } represents a compound statement or block. In C++ object oriented programming, the control structure can be classified into following three categories: Selection or conditional statement; Iterating or looping statement; Breaking statement; Let us discuss the above control statement and their types in the following section.

Selection or conditional statement

In this type of statement, the execution of a block depends on the next condition. If the condition evaluates to true, then one set of statement is executed, otherwise another set of statements is executed. C++ provides following types of selection statements: If; If-else; Nested if; Switch conditional

a) if statement:

The syntax of if statement is

```
If (expression)
{
(Body of if)
Statements;
}
```

Where, expression is the condition that is being evaluated. If this condition is true, statement is executed. If it is false, statement is ignored (not executed) and the program continues right after this conditional structure.

```
# include <iostream.h>
main()
{ int a, b;
a=10;
b=20;
if (a<b)
cout <<"a is less than b";
}

Output: a is less than b.
```

b) if-else statement:

The syntax of if –else statement is

```
If (expression)
{
(Body of if)
Statements 1;
}
else
{
(Body of else}
Statement 2
}

# include <iostream.h>
main()
{ int a, b;
a=10;
b=20;
if (a<b)
cout <<"a is less than b";
}
else
{
cout<< "b is less than a"
}
```

Output: a is less than b.

c) Switch statement:

Switch statement is used for multiple branch selection. The syntax of switch statement is

```
switch (expression)
{
case exp 1:
First case body;
Break;
case exp 2:
Second case body;
Break;
case exp 3:
Third case body;
Break;
default:
default case body;
        }
```

```
# include <iostream.h>
# include <conio.h>
int main()
{
clrscr();
int d_o_w;
cout <<"Enter number of week s day (1-7)";
cin>>d_o_w;
switch(d_o_w)
{
case 1: cout<<"/n Sunday";
break;
case 2: cout<<"/n Monday";
break;
case 3: cout<<"/n Tuesday";
break;
case 4: cout<<"/n Wednesday";
break;
case 5: cout<<"/n Thursday";
break;
case 6: cout<<"/n Friday";
break;
case 7: cout<<"/n Saturday";
break;
default: cout<<"/n Wrong number of day";
}
return 0;
}
```

Output:

```
Enter number of week s dat (1-7):

7

Saturday
```

3.0.12 Iterative or looping statement

In C++ , programming language looping statement is used to repeat a set of instructions until certain condition is fulfilled. The iteration statements are also called loops or looping statement. C++ allows following four kinds of iterative loops:

- for loop
- while loop
- do-while loop and
- nested loops

This loop is specially designed to perform a repetitive action with a counter which is initialized and increased on each iteration

```
#include <iostream.h>
int main ()
{
for (int n=10; n>0; n—)
{
cout << n << ", ";
}
cout << "****!\n";
return 0;
}
```

Output: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, ****!

```
// Program by using while loop
#include <iostream.h>
using namespace std;
int main ()
{
int n;
cout << "Enter the starting number : ";
cin >> n;
while (n>0)
{
cout << n << ", ";
-n;
}
cout << "****!\n";
return 0;
          }
```

Enter the starting number: 8
8, 7, 6, 5, 4, 3, 2, 1, ****!

```
#include <iostream.h>
using namespace std;
int main ()
{
unsigned long n;
do
{
cout << "Enter number (0 to end): ";
cin >> n;
cout << "You entered: " << n << "\n";
}
while (n != 0);
return 0;
}
```

Output: Enter number (0 to end): 12345
You entered: 12345
Enter number (0 to end): 160277
You entered: 160277
Enter number (0 to end): 0
You entered: 0

3.0.13 Breaking statement

Using break, we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. In this section, we will discuss following breaking statements:

- break statement
- continue statement
- goto statement and
- exit statement

```
// break loop example
#include <iostream.h>
int main ()
{
int n;
for (n=10; n>0; n—)
{
cout << n << ", ";
if (n==3)
{
cout << "countdown aborted!";
break;
}
}
return 0;
}
```

Output:
10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!

```
// continue loop example
#include <iostream.h>
int main ()
{
for (int n=10; n>0; n—) {
if (n==5) continue;
cout << n << ", ";
}
cout << "****!\n";
return 0;
}
```

Output:

10, 9, 8, 7, 6, 4, 3, 2, 1, ****!

```
// goto loop example
#include <iostream.h>
using namespace std;
int main ()
{
int n=10;
loop:
cout << n << ", ";
n—;
if (n>0) goto loop;
cout << "****!\n";
return 0;
}
```

Output: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, ****!

```
// Program for exit statement
#include <iostream.h>
#include <conio.h>
void main ()
{
clrscr();
int i, number;
i = 1;
while(i<5)
{
cout <<"Enter the number:";
cin>>number;
if number>5
```

```
{
cout <<"The number is greater than five or equal to" <<end1;
exit();
}
cout << "The number is: "<<number<<end1;
i++
}
getch();
}
```

Output:

```
Enter the number: 2
The number is: 2
Enter the number: 3
The number is: 3
Enter the number: 9
The number is greater than or equal to 9
```

3.1 C++ Classes and Objects

The main purpose of C++ programming is to add object orientation to the C programming language and classes are the central feature of C++ that supports object-oriented programming and are often called user-defined types.

A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class.

3.2 C++ Class Definitions

When we define a class, we define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object. A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword **class** as follows –

```
class Box
{
  public:
    double length;   // Length of a box
    double breadth;  // Breadth of a box
    double height;   // Height of a box
};
```

The keyword **public** determines the access attributes of the members of the class that follows it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as **private** or **protected** which we will discuss in a sub-section.

3.3 Define C++ Objects

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box “

Box Box12;   // Declare Box12 of type Box
Box Box11;   // Declare Box11 of type Box
Both of the objects Box1 and Box2 will have their own copy of data members.

3.4 Accessing the Data Members

The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try the following example to make the things clear –

```
#include <iostream.h>
class Box
{
  public:
    double length;   // Length of a box
    double breadth;  // Breadth of a box
    double height;   // Height of a box
};
```

```
int main()
{
  Box Box1;        // Declare Box1 of type Box
  Box Box2;        // Declare Box2 of type Box
  double volume = 0.0;     // Store the volume of a box here

  // box 1 specification
  Box1.height = 5.0;
  Box1.length = 6.0;
  Box1.breadth = 7.0;

  // box 2 specification
  Box2.height = 10.0;
  Box2.length = 12.0;
  Box2.breadth = 13.0;

  // volume of box 1
  volume = Box1.height * Box1.length * Box1.breadth;
  cout << "Volume of Box1 : " << volume <<endl;

  // volume of box 2
  volume = Box2.height * Box2.length * Box2.breadth;
  cout << "Volume of Box2 : " << volume <<endl;
  return 0;
}
```

When the above code is compiled and executed, it produces the following result "
Volume of Box1 : 210
Volume of Box2 : 1560
It is important to note that private and protected members can not be accessed directly using direct member access operator (.). We will learn how private and protected members can be accessed.

3.5 Classes and Objects in Detail

So far, you have got very basic idea about C++ Classes and Objects. There are further interesting concepts related to C++ Classes and Objects which we will discuss in various sub-sections listed below –

Concept & Description

Class Member Functions
A member function of a class is a function that has its definition or its prototype within the class definition like any other variable.

Class Access Modifiers
A class member can be defined as public, private or protected. By default members would be assumed as private.

Constructor & Destructor
A class constructor is a special function in a class that is called when a new object of the class is created. A destructor is also a special function which is called when created object is deleted.

Copy Constructor
The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.

Friend Functions
A friend function is permitted full access to private and protected members of a class.

Inline Functions
With an inline function, the compiler tries to expand the code in the body of the function in place of a call to the function.

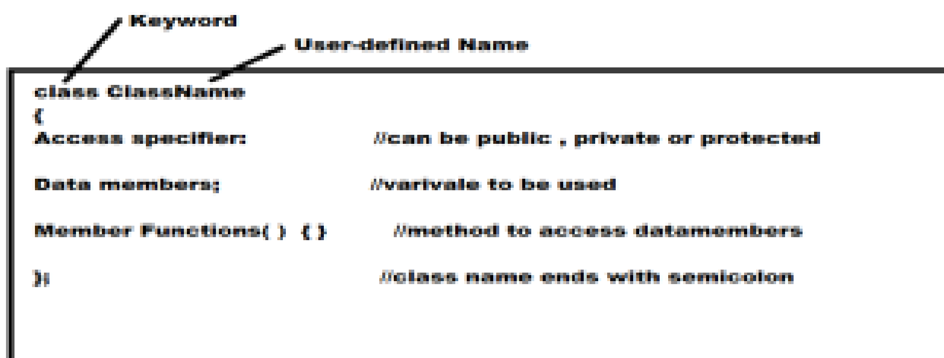
this Pointer
Every object has a special pointer this which points to the object itself.

Pointer to C++ Classes
A pointer to a class is done exactly the same way a pointer to a structure is. In fact a class is really just a structure with functions in it.

Static Members of a Class
Both data members and function members of a class can be declared as static.

3.5 Defining Class and Declaring Objects

A class is defined in C++ using keyword class followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.



3.7 Declaring Objects

When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax:

ClassName ObjectName;

3.8 Accessing data members and member functions

The data members and member functions of class can be accessed using the dot('.') operator with the object. For example if the name of object is *obj* and you want to access the member function with the name *printName()* then you will have to write *obj.printName()* .

Accessing Data Members

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member.
This access control is given by Access modifiers in C++. There are three access modifiers : **public, private and protected**.

Access-control specifiers	**Accessible to**	
	Own class members	**Objects of a class**
private	yes	no
protected	yes	no
public	yes	yes

A class can use all of three visibility/accessibility labels as illustrated below:

```
Class A
{
private:
int x;
void fun1()
{
// This function can refer to data members x, y, z and functions fun1(), fun2()
and fun3()
```

```
}
protected:
int y;
void fun2()
{
//This function can also refer to data members x, y, z and functions fun1(),
fun2() and fun3()
}
public :
int z;
        void fun3()
    {
//This function can also refer to data members x, y, z and functions fun1(),
fun2() and fun3()
}
};
Now, consider the statements
A obja; //obja is an object of class A
int b; // b is an integer variable
```

The above statements define an object obja and an integer variable b. The accessibility of members of the class A is illustrated through the obja as follows:

1. Accessing private members of the class A:
   b=obja.x; //Won t Work: object can not access private data member „x
   obja.fun1(); //Won t Work: object can not access private member function fun1()
   Both the statements are illegal because the private members of the class are not accessible.

2. Accessing protected members of the class A:
   b=obja.y; // Won t Work: object can not access protected data member „y
   obja.fun2(); // Won t Work: object can not access member function fun2()
   Both the statements are also illegal because the protected members of the class are not accessible.

3. Accessing public members of the class A:
   b=obja.c; //OK
   obja.fun3(); //OK
   Both the statements are valid because the public members of the class are accessible.

3.9 Member Functions in Classes

A member function performs an operation required by the class. It is the actual interface to initiate an action of an object belonging to a class. It may be used to read, manipulate, or display the data member. The data member of a class must be declared within the body of the class, while the member functions of a class can be defined in two places:

- ♦ Inside class definition
- ♦ Outside class definition

The syntax of a member function definition changes depending on whether it is defined inside or outside the class declaration/definition. However, irrespective of the location of their definition, the member function must perform the same operation. Thus, the code inside the function body would be identical in both the cases. The compiler treats these two definitions in a different manner. Let us see, how we can define the member function inside the class definition.

The syntax for specifying a member function declaration is similar to a normal function definition except that is enclosed within the body of a class. Foe example, we could define the class as follows:

```
class Number
{
int x, y, z;
public:
void get_data(void); //declaration
void maximum(void); //declaration
void minimum(void) //definition
{
int min;
min=x;
if (min>y)
min=y;
if (min>z)
min=z;
cout<<“\n Minimum value is =”<<min<<endl;
}
};
```

if you look at the above declaration of class number you can observe that the member function get_data() and maiximun() are declared, but they are not defined. The only member function which is defined in the class body is minimum(). When a function is defined inside a class, it is treated as a inline function. Thus, member function minimum is a inline function. Generally, only small functions are defined inside the class.

Now let us see how we can define the function outside the class body. Member functions that are declared inside a class have to be defined outside the class. Their definition is very much like the normal function. Can you tell how does a compiler know to which class outside defined function belong? Yes, there should be a mechanism of binding the functions to the class to which they belong. This is done by the scope resolution operator (**::**). It acts as an identity-label. This label tells the compiler which class the function belongs to. The common syntax for member function definition outside the class is as follows:

```
return_type class_name :: function_name(argument declaration)
{
functionbody
}
The scope resolution :: tells the compiler that the function_name belongs to the
class class_name. Let us again consider the class Number.
class Number
{
int x, y, z;
public:
void get_data(void); //declaration
void maximum(void); //declaration.
.
.
};
void Number :: get_data(void)
{
cout<< "\n Enter the value of fist number(x):"<<endl;
cin>>x;
cout<< "\n Enter the value of second number(y):"<<endl;
cin>>y;
cout<< "\n Enter the value of third number(z):"<<endl;
cin>>z;


}
void Number :: maximum(void)
{
int max;
max=x;
if (max<y)
max=y;
if (max<z)
max=z;
cout<<"\n Maximun value is ="<<max<<endl;
}
```

if you look at the above declaration of class Number, you can easily see that the member function get_data() and maiximun() are declared in the class. Thus, it is necessary that you have to define this function. You can also observe in the above snapshot of C++ program identity label (::) which are used in void Number :: get_data(void) and void Number ::maximum(void) tell the compiler the functionget_data() and maximum() belong to the class Number.

Now, let us see the complete C++ program to find out the minimum and maximum of three given integer numbers:

```
#include<iostream.h>
class Number
{
int x, y, z;
public:
void get_data(void); //declaration
void maximum(void); //declaration
void minimum(void) //definition
{
int min;
min=x;
if (min>y)
min=y;
if (min>z)
min=z;
cout<<"\n Minimum value is ="<<min<<endl;
}
};
void Number :: get_data(void)
{
cout<< "\n Enter the value of fist number(x):"<<endl;
cin>>x;
cout<< "\n Enter the value of second number(y):"<<endl;
cin>>y;
cout<< "\n Enter the value of third number(z):"<<endl;
cin>>z;
}
void Number :: maximum(void)
{
int max;
max=x;
if (max<y)
max=y;
if (max<z)
max=z;
```

```
cout<<"\n Maximun value is ="<<max<<endl;
}
void main()
{
Number num;
num.get_data();
num.minimum();
num.maximum();
}

output:
Enter the value of the first number (x):
10
Enter the value of the second number (y):
20
Enter the value of the third number (z):
5
Minimum value is=5
Maximum value is=20
```

3.10 Passing and Returning Objects in C++

In C++ we can pass class's objects as arguments and also return them from a function the same way we pass and return other variables. No special keyword or header file is required to do so.

Passing an Object as argument

To pass an object as an argument we write the object name as the argument while calling the function the same way we do it for other variables.

```
Syntax:

function_name(object_name);
```

Example: In this Example there is a class which has an integer variable 'a' and a function 'add' which takes an object as argument. The function is called by one object and takes another as an argument. Inside the function, the integer value of the argument object is added to that on which the 'add' function is called. In this method, we can pass objects as an argument and alter them.

```
// C++ program to show passing//
of objects to a function
#include<iostream.h>
class Example {
```

```
public:
int a;
// This function will take
// an object as an argument
void add(Example E)
{
a = a + E.a;
}
};
// Driver Code
int main()
{
// Create objects
Example E1, E2;
// Values are initialized for both objects
E1.a = 50;
E2.a = 100;
cout << "Initial Values \n";
cout << "Value of object 1: " << E1.a
<< "\n& object 2: " << E2.a << "\n\n";
// Passing object as an argument
// to function add()
E2.add(E1);
// Changed values after passing
// object as argument
cout << "New values \n";
cout << "Value of object 1: " << E1.a
<< "\n& object 2: " << E2.a
<< "\n\n";
return 0;
}
```

Output:

```
Initial Values
Value of object 1: 50
& object 2: 100

New values
Value of object 1: 50
& object 2: 150
```

3.11 Returning Object (method) as argument

Syntax:

object = return object_name;

Example: In the above example we can see that the add function does not return any value since its return-type is void. In the following program the add function returns an object of type 'Example'(i.e., class name) whose value is stored in E3.

In this example, we can see both the things that are how we can pass the objects as well as return them. When the object E3 calls the add function it passes the other two objects namely E1 & E2 as arguments. Inside the function, another object is declared which calculates the sum of all the three variables and returns it to E3.

This code and the above code is almost the same, the only difference is that this time the add function returns an object whose value is stored in another object of the same class 'Example' E3. Here the value of E1 is displayed by object1, the value of E2 by object2 and value of E3 by object3.

```
// C++ program to show passing
// of objects to a function
#include <iostream.h>
class Example {
public:
int a;
// This function will take
// object as arguments and
// return object
Example add(Example Ea, Example Eb)
{
Example Ec;
Ec.a = 50;
Ec.a = Ec.a + Ea.a + Eb.a;
// returning the object
return Ec;
}
};
```

```
int main()
{
Example E1, E2, E3;
// Values are initialized
// for both objects
E1.a = 50;
E2.a = 100;
E3.a = 0;
cout << "Initial Values \n";
cout << "Value of object 1: " << E1.a
<< ", \nobject 2: " << E2.a
<< ", \nobject 3: " << E3.a << "\n";
// Passing object as an argument
// to function add()
E3 = E3.add(E1, E2);
// Changed values after
// passing object as an argument
cout << "New values \n";
cout << "Value of object 1: " << E1.a << ", \nobject 2: " << E2.a << ", \nobject
3: " << E3.a<< "\n";
return 0;
}
```

Output:

```
Initial Values
Value of object 1: 50,
object 2: 100,
object 3: 0

New values
Value of object 1: 50,
object 2: 100,
object 3: 200
```

3.12 Friend Function

As we have discussed that the private members cannot be accessed from outside the class. It implies that a non-member function cannot have an access to the private data of a class. Let us suppose, we want a function operate on objects of two different classes. In such situations, C++ provides the friend function which is used to access the private members of a class. Friend function is not a member of any class. So, it is defined without scope resolution operator. The syntax of declaring friend function is given below:

```
class class_name
{
…
…
public:
…
…
friend return type function_name(arguments);
}
```

Let us consider the complete C++ program to find out sum of n given numbers to understand the concept of friend function

```
#include<iostream.h>
#define MAX_SIZE 100
class Sum
{
int num[MAX_SIZE];
int n;
public:
void get_number(void);
friend int add(void);
};
void Sum :: get_number(void)
{
cout<< "\n Enter the total number(n):"<<endl;
cin>>n;
cout<< "\n Enter the number:"<<endl;
for (int i=0;i<n; i++)
cin>>num[i];
}
int add(void)
{
Sum s;
```

```
int temp=0;
s.get_number();
for (int i=0;i<s.n; i++)
temp+=s.num[i];
return temp;
}
void main()
{
int res;
res=add();
cout<<"The sum of n value is="<<res<<endl;
}
```

If you look at the above program, you can easily see that the function add is declared as a friend function of class Sum. The add function accesses the private data, adds the numbers of array and returns value to the main function where it is called upon. Furthermore, you can also see that friend function add() is defined without scope resolution operator(::), because it does not belong to a class.

Now, let us consider a situation in which we want to operate on objects of two different classes. In such a situation, friend functions can be used to bridge the two classes

```
#include<iostream.h>
class Two; //forward declaration like function prototype
class One
{
int a;
public:
void get_a(void);
friend int min(One, Two);
};
class Two
{
int b;
public:
void get_b(void);
friend int min(One, Two);
};
void One :: get_a(void)
{
cout<<"Enter the value of a:"<<endl;
cin>>a;
}
```

```
void Two :: get_b(void)
{
cout<<"Enter the value of b:"<<endl;
cin>>b;
}
int min (One o, Two t)
{
if(o.a<t.b)
return o.a;
else
return t.b;
}
void main()
{
One one;
Two two;
int minvalue;
one.get_a();
two.get_b();
minvalue=min(one,two);
cout<<"Minimum="<<minvalue<<endl;
}
```

You can observe that the above program contains two classes named one and two. The function min() is declared in the both the classes with the keyword friend. An object of each class has been passed as an argument to the function min (). Being a friend function, it can access the private members of both classes through these arguments.

Now, let us note some special properties possessed by friend function:

(i) A friend function is not in the scope of the class to which it has been declared as friend.
(ii) A friend function cannot be called using the object of that class. It can be invoked like a normal function without the use of any object.
(iii) Unlike member functions, it can not access the members directly. However, it can use the object and dot membership operator with each member name to access both private and public members.
(iv) It can be declared either in the public or the private part of a class without affecting its meaning.
(v) Generally, it has got objects as arguments.

3.13 Pointer to C++ Classes

A pointer to a C++ class is done exactly the same way as a pointer to a structure and to access members of a pointer to a class you use the member access operator **->** operator, just as you do with pointers to structures. Also as with all pointers, you must initialize the pointer before using it.
Let us try the following example to understand the concept of pointer to a class –

```
#include <iostream.h>
 class Box {
   public:
      // Constructor definition
      Box(double l = 2.0, double b = 2.0, double h = 2.0) {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
      }
      double Volume() {
         return length * breadth * height;
      }
      private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};
int main(void) {
   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2
   Box *ptrBox;                // Declare pointer to a class.
   // Save the address of first object
   ptrBox = &Box1;
   // Now try to access a member using member access operator
   cout << "Volume of Box1: " << ptrBox->Volume() << endl;
   // Save the address of second object
   ptrBox = &Box2;

   // Now try to access a member using member access operator
   cout << "Volume of Box2: " << ptrBox->Volume() << endl;

   return 0;
}
```

Output :

Constructor called.
Constructor called.
Volume of Box1: 5.94
Volume of Box2: 102

3.14 C++ Array of Pointers

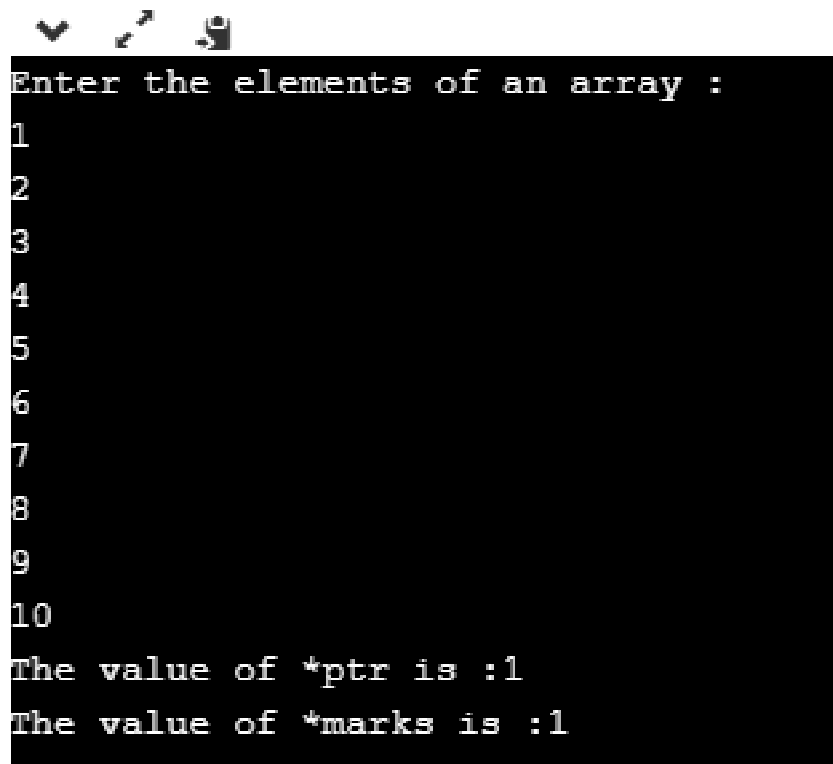
Array and pointers are closely related to each other. In C++, the name of an array is considered às a pointer, i.e., the name of an array contains the address of an element. C++ considers the array name as the address of the first element. For example, if we create an array, i.e., marks which hold the 20 values of integer type, then marks will contain the address of first element, i.e., marks[0]. Therefore, we can say that array name (marks) is a pointer which is holding the address of the first element of an array.

Let's understand this scenario through an example.

```
1. #include <iostream.h>
2. int main()
3. {
4. int *ptr;  // integer pointer declaration
5. int marks[10]; // marks array declaration
6. std::cout << ”Enter the elements of an array :” << std::endl;
7. for(int i=0;i<10;i++)
8. {
9. cin>>marks[i];
10. }
11. ptr=marks; // both marks and ptr pointing to the same element..
12. std::cout << ”The value of *ptr is :” <<*ptr<< std::endl;
13. std::cout << ”The value of *marks is :” <<*marks<<std::endl;
14. }
```

In the above code, we declare an integer pointer and an array of integer type. We assign the address of marks to the ptr by using the statement ptr=marks; it means that both the variables 'marks' and 'ptr' point to the same element, i.e., marks[0]. When we try to print the values of *ptr and *marks, then it comes out to be same. Hence, it is proved that the array name stores the address of the first element of an array.

Output :



Array of Pointers

An array of pointers is an array that consists of variables of pointer type, which means that the variable is a pointer addressing to some other element. Suppose we create an array of pointer holding 5 integer pointers; then its declaration would look like:

1. **int** *ptr[5]; // array of 5 integer pointer.

In the above declaration, we declare an array of pointer named as ptr, and it allocates 5 integer pointers in memory.

The element of an array of a pointer can also be initialized by assigning the address of some other element. Let's observe this case through an example.

1. **int** a; // variable declaration.

2. ptr[2] = &a;

In the above code, we are assigning the address of 'a' variable to the third element of an array 'ptr'.

We can also retrieve the value of 'a' be dereferencing the pointer.

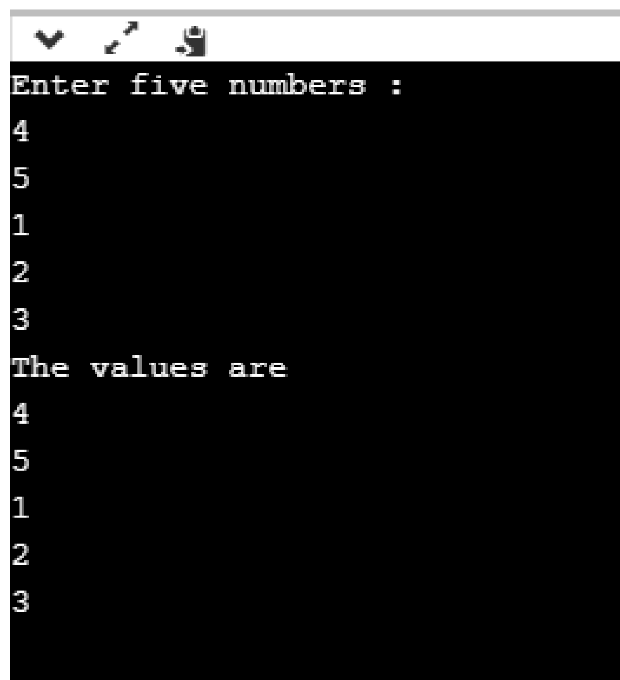
1. *ptr[2];

Let's understand through an example.

```
1.  #include <iostream.h>
2.  int main()
3.  {
4.     int ptr1[5]; // integer array declaration
5.     int *ptr2[5]; // integer array of pointer declaration
6.     std::cout << "Enter five numbers :" << std::endl;
7.     for(int i=0;i<5;i++)
8.     {
9.        std::cin >> ptr1[i];
10.    }
11.    for(int i=0;i<5;i++)
12.    {
13.       ptr2[i]=&ptr1[i];
14.    }
15.    // printing the values of ptr1 array
16.    std::cout << "The values are" << std::endl;
17.    for(int i=0;i<5;i++)
18.    {
19.       std::cout << *ptr2[i] << std::endl;
20.    }
21.    }
```

In the above code, we declare an array of integer type and an array of integer pointers. We have defined the 'for' loop, which iterates through the elements of an array 'ptr1', and on each iteration, the address of element of ptr1 at index 'i' gets stored in the ptr2 at index 'i'.

Output :



Till now, we have learnt the array of pointers to an integer. Now, we will see how to create the array of pointers to strings.

Array of Pointer to Strings

An array of pointer to strings is an array of character pointers that holds the address of the first character of a string or we can say the base address of a string.

The following are the differences between an array of pointers to string and two-dimensional array of characters:

- An array of pointers to string is more efficient than the two-dimensional array of characters in case of memory consumption because an array of pointer to strings consumes less memory than the two-dimensional array of characters to store the strings.
- In an array of pointers, the manipulation of strings is comparatively easier than in the case of 2d array. We can also easily change the position of the strings by using the pointers.

Let's see how to declare the array of pointers to string.

First, we declare the array of pointer to string:

```
1. char *names[5] = {“john”,
2.                  ”Peter”,
3.                  ”Marco”,
4.                  ”Devin”,
5.                  ”Ronan”};
```

In the above code, we declared an array of pointer names as 'names' of size 5. In the above case, we have done the initialization at the time of declaration, so we do not need to mention the size of the array of a pointer. The above code can be re-written as:

```
1. char *names[ ] = {“john”,
2.                  ”Peter”,
3.                  ”Marco”,
4.                  ”Devin”,
5.                  ”Ronan”};
```

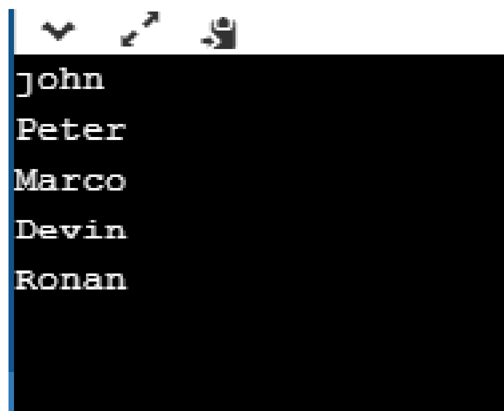
In the above case, each element of the 'names' array is a string literal, and each string literal would hold the base address of the first character of a string. For example, names[0] contains the base address of “john”, names[1] contains the base address of “Peter”, and so on. It is not guaranteed that all the string literals will be stored in the contiguous memory location, but the characters of a string literal are stored in a contiguous memory location.

Let's create a simple example.

```
1. #include <iostream.h>
2. int main()
3. {
4.   char *names[5] = {“john”,
5.                    ”Peter”,
```

```
6.                  "Marco",
7.                  "Devin",
8.                  "Ronan"};
9.  for(int i=0;i<5;i++)
10.    {
11.       std::cout << names[i] << std::endl;
12.    }
13.    return 0;
14. }
```

In the above code, we have declared an array of char pointer holding 5 string literals, and the first character of each string is holding the base address of the string.



Output

3.0.14 Questions

Q.1. Write a C+ + program to find out the sum of n numbers.
Q.2. What are classes? Explain purpose of classes?
Q.3. Explain the term control structure in C++?
Q.4. Explain the function of operators in C++?
Q.5. What do you mean by keyword in C++?
Q.6. What do you mean by identifier in C++?
Q.7. What do you mean by data types in C++?
Q.8. What is the difference between variable and constant in C++ programming language?
Q.9. What do you mean by global variable and local variable in C++?
Q.10. Explain the difference between C and C++?
Q.11. What is the scope resolution operator?
Q.12. What is the purpose of the member function?
Q.13. What is friend class? Write a program to illustrate the concept of friend class.
Q.14. Why is friend function needed ?

*****

4

CONSTRUCTORS AND DESTRUCTORS

Chapter Structure :

4.1 C++ Class Constructor and Destructor
4.2 Features of the Constructor
4.3 Constructor with Parameters
4.4 Destructor
4.5 Features of Destructor
4.6 Summary
4.7 Reference
4.8 Questions

4.1 C++ Class Constructor and Destructor

constructor is a special member function whose name is same as the name of its class in which it is declared and defined. The purpose of the constructor is to initialize the objects of the class. The constructor is called so because it is used to construct the objects of the class.

Example of constructor and later illustrate its various features :

```
#include < iostream.h >
#include < conio.h >
class demo
{
public:
demo()
{
cout< <"Hello from constructor \n";
}
};
void main()
}
clrscr( );
demo d;
getch( );

OUTPUT :

Hello from constructor
```

4.2 Features of the Constructor

1. The constructors are always declared in the public section. If declared in the private section then objects are can only be created inside the member functions but serve no purpose.
2. They are invoked automatically when objects of the class are created. The declaration demo d;creates an object d which automatically calls the constructor of the class and prints Hello from constructor.
3. They do not have any return type not even void so they cannot return any value.
4. Constructors cannot be inherited, but they can be called from the constructors of derived class.
5. Constructors are used to construct the object of the class.
6. The constructor with no argument is known as default constructor of the class. The default constructor for the class demo will be demo : demo( )
7. Constructors which take arguments like a function takes are known as parameterized constructor.
8. There is no limit of the number of constructors declared in a class but they all must conform to rules of function overloading.
9. Constructor can have default arguments.
10. Addresses of constructors cannot be taken.
11. Constructors cannot be virtual.
12. Constructor make implicit calls to operators new and delete in case memory allocation and de- allocation is to be performed.

4.3 Constructor with Parameters

Constructor are similar to functions but they have the name as class name so similar to functions which takes argument we can have constructor which can take arguments. The constructor which takes parameters is known as parameterized constructor. Again depending upon type of arguments and number of arguments they may be overloaded. An example of this is given below:

```
#include <iostream.h>
#include <conio.h>
class demo
{
int a,b;
public :
demo( )
{
a=b=0;
cout<<"Zero argument constructor called\n";
show( );
```

```
}
demo(int x, int y)
{
a=x;
b=y;
cout<<"Two argument constructor called\n";
show( );
}
demo(int x)
{
a=b=x;
cout<<"One argument constructor called\n";
show( );
}
void show( )
{
cout<<"a="<<a<<"\tb="<<b<<endl;
}
};
void main( )
{
clrscr( );
demo d1;
demo d2(10,20);
demo d3(30);
getch( );
}
```

OUTPUT :

```
Zero argument constructors called
a=0 b=0
Two argument constructor called
a=10 b=20
One argument constructor called
a=30 b=30
```

4.4 Destructor

A destructor is a member function of the class whose name is same as the name of the class but the preceded with tilde sign (~). The purpose of destructor is to destroy the object when it is no longer needed or goes out of scope. As a very small example of destructor see the program given below :

```
#include <iostream.h>
#include <conio.h>
class demo
{
public :
demo( )
{
cout<<"Constructor called\n";
}
~demo( )
{
cout<<"Destructor called"<<endl;
}
};
void main( )
{
clrscr( );
demo d;
getch( );
}
```

OUTPUT :

```
Constructor called
Destructor called
```

4.5 Features of Destructor

1. The name is same as of class but proceeded with a ~ sign.
2. Destructor is automatically called as soon as an object goes out of scope.
3. Destructor is used to destroy the objects.
4. Once a destructor is called for a object, the object will no longer be available for the future reference.
5. Destructor can be used for housekeeping work such as closing the file, de-allocating the dynamically allocated memory etc. Closing a file in destructor

is a good idea as user might forget to close the file associated with object. But as the object goes out of scope destructor will be called and all code written in destructor executes which will always result in closing the file and no data loss may be there. When new is used for allocation of memory in the constructor we must always use delete in the destructor to be allocate the memory.

6. Similar to constructor there is no return type for destructor and that's why they cannot return any value.
7. There is no explicit or implicit category for a destructor. They are always called implicitly by the compiler.
8. Destructor can never take any arguments.
9. Destructor can be virtual.

4.6 Summary

- In this unit, we have seen and discussed the concept of class as well as object which are the basic components used in C++ programming.
- Class declaration and object creation have been discussed and illustrated with examples. The members viz., data members and function members of a class, defining member functions were explained and elaborated.
- We have seen the way to pass objects as arguments to the functions with call by value and call by reference. Furthermore, static members, Friend function and Friend class are also discussed with examples.
- A class constructor is a class method having the same name as the class name. The constructor does not have a return type. A constructor may take zero or more parameters. If we don't apply a constructor, the compiler provides a no-argument default constructor. It is important to understand that if we write our own constructor, the compiler does not provide the default constructor.
- A class may define one or more constructor. It is up to us to decide which constructor to call during object creation by passing an appropriate parameter list to the constructor. We may set the default value for the constructor parameter.
- A class destructor is a class method having the same name as the class name and is prefixed with tilde (~) sign. The destructor does not return anything and does not accept any argument. A class definition may contain one and only one destructor.
- The runtime calls the destructor, if available, during the object creation. A destructor is typically used for freeing the resources allocated in the class constructor.

FURTHER READINGS :-

- B. Stroustrup, *The C++ Programming Language*, third edition, Pearson/ Addison-wesley Publication,1997.
- K. R. Venu Gopal, Raj Kumar Buyya, T Ravishankar, *Mastering C++*, Tata-McGraw-Hill Publishing Company Limited, New Delhi.
- E. Balagurusamy, *Object Oriented Programming with C++*, Tata Mc-Graw-Hill Publishing Company Limited, New Delhi.

4.7 References

www.w3school.com

www.tutorialspoint.com

www.javatpoint.com

www.programiz.com

4.8 Questions

Q1. Write a program in C++ to demonstrate the use of destructor.

Q2. Explain the role of destructor in C++ in memory management.

Q3. Define parameterized constructor by taking a C++ program.

Q4. What do you mean by destructor? Explain with example.

Q5. Explain the application of constructors and destructors in C++ programming.

Q6. What do you mean by default constructor? Explain with example.

Q7. What do you mean by constructor in C++ programming?

*****

UNIT – III

5

POLYMORPHISM

Chapter Structure :

5.0 Polymorphism
5.1 Types of Polymorphism
5.2 Function Overloading
5.3 Operator Overloading
5.4 Binary Operators Overloading in C++
5.5 Unary Operators Overloading in C++
5.6 Comparison / Relational Operators Overloading in C++
5.7 Assignment Operators Overloading in C++
5.8 Conversion Types in C++
5.9 Questions

5.0 Polymorphism

Polymorphism is the ability to use an operator or method in different ways. Polymorphism gives different meanings or functions to the operators or methods. Poly refers many that signify the many uses of these operators and methods. A single method usage or an operator functioning in many ways can be called polymorphism. Polymorphism refers to codes, operations or objects that behave differently in different contexts. "Polymorphism is a mechanism that allows you to implement a function in different ways."

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

- 5+ 20 //The above refers to integer addition.
- The same + operator can be used with different meanings with strings:
- "Tech" + "nical"
- The same + operator can be also used for floating point addition: 2.15 + 10.20

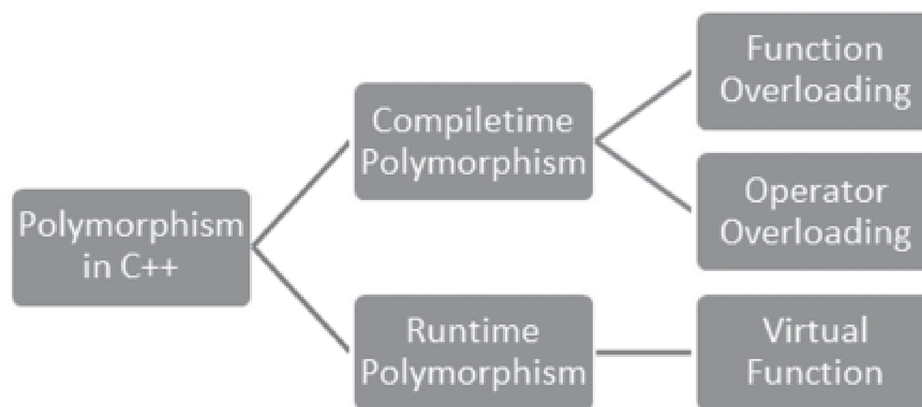
We saw above that a single operator „+ behaves differently in different contexts such as integer, string or float referring the concept of polymorphism. The above concept leads to operator overloading. When the exiting operator or function operates on new data type it is overloaded. C++ also permits the use of different functions with the same name. Such functions have different argument list. The difference can be in terms of number or type of arguments or both. It refers as function overloading.

So, we conclude that the concept of operator overloading and function overloading is a branch of polymorphism. Both the concepts have been discussed in unit 2 in detail.

5.1 Types of Polymorphism

C++ provides three different types of polymorphism:

- Function overloading
- Operator overloading
- Virtual functions



5.2 Function Overloading

Function overloading refers to using the same thing for different purposes. C++ permits the use of different functions with the same name. However such functions essentially have different argument list. The difference can be in terms of number or type of arguments or both. These functions can perform a variety of different tasks. This process of using two or more functions with the same name but differing in the signature is called function overloading. It is only through these differences that the compiler knows which function to call in any given situations.

Consider the following function :

```
int add (int a, int b)
{
return a + b;
}
```

This trivial function adds two integers. However, what if we also need to add two floating point numbers? This function is not at all suitable, as any floating point parameters would be converted to integers, causing the floating point arguments to lose their fractional values.

One way to work around this issue is to define multiple functions with slightly different names:

```
int add (int a, int b)
{
return a + b;
}
Double  addition (double p, double q)
{
return p + q;
}
```

How Function Overloading is Achieved

1) Overloading functions that differ in terms of number of parameters

```
#include<iostream.h>
// function prototype
int func(int i);
int func(int i, int j);
void main(void)
{
cout<<func(15); //func(int i)is called
cout<<func(15,15); //func(int i, int j) is called
}
int func(int i)
{
return i;
}
int func(int i, int j)
{
return i+j;
}
```

2) Overloading functions that differ in terms of types of parameters

```
#include<iostream.h>
//function prototypes
int func(int i);
double func(double i);
void main(void)
{
cout<<func(15); //func(int i) is called
cout<<func(15.155); //func(double i) is called
}
int func(int i)
{
```

```
return i;
}
double func(double i)
{
return i;
}
```

5.3 Operator Overloading

We already know that a function can be overloaded (same function name having multiple bodies). The concept of overloading a function can be applied to operators as well. For example, in C++ we can multiply two variables of user-defined data type with the same syntax that is applied to the basic data type. This means that C++ has the ability to provide the operators with a special meaning for data type. The mechanism which provides this special meaning to operators is called operator overloading. The operator overloading feature of C++ is one of the methods of realizing polymorphism. Here, poly refers to many or multiple and morphism refers to actions, i.e. performing many actions with a single operator. Thus operator overloading enables us to make the standard operators, like +, -, * etc, to work with the objects of our own data types. So what we do is, write a function which redefines a particular operator so that it performs a specific operation when it is used with the object of a class. Operator overloading is very exciting feature of C++. The concept of operator overloading can also be applied to data conversion. It enhances the power of extensibility of C++. Thus operator overloading concepts are applied to the following two principle areas:

- Extending capability of operators to operate on user defined data, and
- Data conversion

General rules for Operator Overloading :

- Only built-in operators can be overloaded. If some operators are not present in C++, we cannot overload them.
- The arity of the operators cannot be changed
- The precedence of the operators remains same.
- The overloaded operator cannot hold the default parameters except function call operator "()".
- We cannot overload operators for built-in data types. At least one user defined data types must be there.
- The assignment "=", subscript "[]", function call "()" and arrow operator "->" these operators must be defined as member functions, not the friend functions.
- Some operators like assignment "=", address "&" and comma "," are by default overloaded.

5.4 Binary Operators Overloading in C++

When overloading binary operator using operator function as class member function, the left side operand must be an object of the class and right side operand may be either a built-in type or user defined type. The other method using friend function will be discussed later on. We present numerous program of overloading binary operators.

The binary operators take two arguments and following are the examples of Binary operators. You use binary operators very frequently like addition (+) operator, subtraction (-) operator and division (/) operator.

Following example explains how addition (+) operator can be overloaded. Similar way, you can overload subtraction (-) and division (/) operators.

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

class Box {
  double length;      // Length of a box
  double breadth;     // Breadth of a box
  double height;      // Height of a box

  public:

  double getVolume(void) {
    return length * breadth * height;
  }

  void setLength( double len ) {
    length = len;
  }

  void setBreadth( double bre ) {
    breadth = bre;
  }

  void setHeight( double hei ) {
    height = hei;
  }

  // Overload + operator to add two Box objects.
  Box operator+(const Box& b) {
    Box box;
```

```
      box.length = this->length + b.length;
      box.breadth = this->breadth + b.breadth;
      box.height = this->height + b.height;
      return box;
    }
};

// Main function for the program
int main() {
  Box Box1;                // Declare Box1 of type Box
  Box Box2;                // Declare Box2 of type Box
  Box Box3;                // Declare Box3 of type Box
  double volume = 0.0;     // Store the volume of a box here

  // box 1 specification
  Box1.setLength(6.0);
  Box1.setBreadth(7.0);
  Box1.setHeight(5.0);

  // box 2 specification
  Box2.setLength(12.0);
  Box2.setBreadth(13.0);
  Box2.setHeight(10.0);

  // volume of box 1
  volume = Box1.getVolume();
  cout << "Volume of Box1 : " << volume <<endl;
   // volume of box 2
  volume = Box2.getVolume();
  cout << "Volume of Box2 : " << volume <<endl;

  // Add two object as follows:
  Box3 = Box1 + Box2;

  // volume of box 3
  volume = Box3.getVolume();
  cout << "Volume of Box3 : " << volume <<endl;

  return 0;
}
```

Output :-

Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400

5.5 Unary Operators Overloading in C++

Similar to overloading binary operator we can overload unary “, pre and post ++, pre and post – and unary +. In case of overloading binary operators using member function of class left operand is responsible for calling the operator function and right operand was send as argument.

In case of unary operator only one operand is there and this operand itself calls the overloaded operator function. Nothing is send as argument to the function. Its general syntax is (defined with the class)

The unary operators operate on a single operand and following are the examples of Unary operators H

- The increment (++) and decrement (—) operators.
- The unary minus (-) operator.
- The logical not (!) operator.

The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj—.

Following example explain how minus (-) operator can be overloaded for prefix as well as postfix usage.

```
#include <iostream>
using namespace std;
class Distance {
  private:
    int feet;          // 0 to infinite
    int inches;          // 0 to 12

  public:
    // required constructors
    Distance() {
      feet = 0;
      inches = 0;
    }
    Distance(int f, int i) {
      feet = f;
      inches = i;
    }

    // method to display distance
    void displayDistance() {
      cout << "F: " << feet << " I:" << inches <<endl;
```

```
    }

    // overloaded minus (-) operator
    Distance operator- () {
      feet = -feet;
      inches = -inches;
      return Distance(feet, inches);
    }
};

int main() {
  Distance D1(11, 10), D2(-5, 11);

  -D1;                    // apply negation
  D1.displayDistance();   // display D1

  -D2;                    // apply negation
  D2.displayDistance();   // display D2

  return 0;
}
```

Output :-

F: -11 I:-10

F: 5 I:-11

5.6 Comparison / Relational Operators Overloading in C++

There are various relational operators supported by C++ language like (<, >, <=, >=, ==, etc.) which can be used to compare C++ built-in data types.

You can overload any of these operators, which can be used to compare the objects of a class.

Following example explains how a < operator can be overloaded and similar way you can overload other relational operators.

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

class Distance {
  private:
    int feet;             // 0 to infinite
    int inches;           // 0 to 12
```

```
   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }

      // method to display distance
      void displayDistance() {
         cout << "F: " << feet << " I:" << inches <<endl;
      }

      // overloaded minus (-) operator
      Distance operator- () {
         feet = -feet;
         inches = -inches;
         return Distance(feet, inches);
      }

      // overloaded < operator
      bool operator <(const Distance& d) {
         if(feet < d.feet) {
            return true;
         }
         if(feet == d.feet && inches < d.inches) {
            return true;
         }

         return false;
      }
};

int main() {
   Distance D1(11, 10), D2(5, 11);

   if( D1 < D2 ) {
      cout << "D1 is less than D2 " << endl;
   } else {
      cout << "D2 is less than D1 " << endl;
   }
```

```
   return 0;
 }
```

Output :-
D2 is less than D1

5.7 Assignment Operators Overloading in C++

You can overload the assignment operator (=) just as you can other operators and it can be used to create an object just like the copy constructor.

Following example explains how an assignment operator can be overloaded.

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

class Distance {
  private:
    int feet;          // 0 to infinite
    int inches;          // 0 to 12

  public:
    // required constructors
    Distance() {
      feet = 0;
      inches = 0;
    }
    Distance(int f, int i) {
      feet = f;
      inches = i;
    }
    void operator = (const Distance &D ) {
      feet = D.feet;
      inches = D.inches;
    }

    // method to display distance
    void displayDistance() {
      cout << "F: " << feet << " I:" << inches << endl;
    }
};
```

```
int main() {
  Distance D1(11, 10), D2(5, 11);

  cout << "First Distance : ";
  D1.displayDistance();
  cout << "Second Distance :";
  D2.displayDistance();

  // use assignment operator
  D1 = D2;
  cout << "First Distance :";
  D1.displayDistance();

  return 0;
}
```

Output :-
First Distance : F: 11 I:10
Second Distance :F: 5 I:11
First Distance :F: 5 I:11

5.8 Conversion Types in C++

Many times in programming situations we like to convert one data type into another. Converting data types of a variable into other data type is known as type conversion. We have studied it earlier when we convert int to float, char to int, double to int etc. Compiler also does implicit type conversion. But all we studied involved all built-in types. In this section, we study type conversion with respect to user data type viz. class. We divide the study of type conversion into three parts.
1. Conversion from Built-in types to class type.
2. Conversion from class type to built-in types.
3. Conversion from one class type to another.

1) Conversion from built in type to class type :

A constructor is used to build a matrix object from an int type array. Similarly, we used another constructor to build a string type object from char* type variable. In below examples constructors performed a defacto type conversion from the argument's type to the constructor's class type.

Consider the following constructor :-

```
string :: string(char*a)
{
  length = strlen(a);
  name = new char [len + 1];
  strcpy(name,a);
}
```

This constructor builds a string type object from char* type variable a. The variables length and name are data members of the class string. Once we define the constructor in the class string, it can be used for conversion from char* type to string type.

Example 1:

```
string s1 , s2;
char* name1 = " Good Morning " ;
char* name2 = " Students " ;
s1 = string ( name1 ) ;
s2 = name2 ;
```

Explanation :-

```
s1 = string ( name1 ) ;
```

In the above code snippet, first converts name1 from char* type to string type and then assigns the string type values to the object s1.

The statement below performs the same job by invoking the constructor implicitly.:

```
s2 = name2 ;
```

Example 2:

```
class time
{
  int hours;
  int minutes;
  public :
  void time ( int t )   // II constructor
  {
    hours = t / 60 ;   // t is inputted in minutes
    minutes = t % 60 ;
  }
};
```

Explanation :-

```
time t1;   // object t1 created
int period = 160;
t1 = period;   // int to class type
```

The object t1 is created. The variable period of data type integer is converted into class type time by invoking the constructor. After the conversion, the data member hours of T1 will have value 2 and minutes will have a value of 40 denoting 2 hours and 40 minutes.

Note that constructors are used for the type conversion take a single argument whose type is to be converted.

Note : In both the examples, the left-hand operand of = operator is always a class object. Hence, we can also accomplish this conversion using an overloaded = operator.

2) Conversion from class type to built in type :

The constructor functions do not support conversion from a class to basic type. C++ allows us to define a overload casting operator that convert a class type data to basic type.

The general syntax of an overloaded casting operator function, also referred to as a conversion function, is :

```
operator typename()
{
 // Program statement
}
```

Example :

In the below example, operator double () converts a class object to type double.

```
vector :: operator double ()
{
  double sum = 0 ;
  for ( int i = 0 ; i < size ; i++ )
     sum = sum + v[i] * v[i] ;
  return sqrt ( sum ) ;
};
```

The casting operator should satisfy the following conditions :-

It must be a class member.
It must not specify a return type.

It must not have any arguments. The values inside the function used for conversion belongs to the object that invoked the function. So, function does not need an argument.

3) Conversion from one class to another class type :

We have just seen data conversion techniques from a basic to class type and a class to basic type. But sometimes we would like to convert one class data type to another class type.

Example :

obj1 = obj2 ;   // obj1 and obj2 are objects of different classes
Explanation :-

obj1 is an object of class one and obj2 is an object of class two. The class two type data is converted value is assigned to the obj1. Since the conversion takes place from class two to class one, two is known as the source and one is known as the destination class.

Such conversion between objects of different classes can be carried out by either a constructor or a conversion function. Which form to use, depends upon where we want the type-conversion function to be located, whether in the source class or in the destination class.
Casting operator function - Operator typename () :-
This function converts the class object of which it is a member to typename. The typename may be a built-in type or a user defined one(another class type). In the case of conversions between objects, typename refers to the destination class. Therefore, when a class needs to be converted, a casting operator function can be used. The conversion takes place in the source and the result is given to the destination class.

The following program uses classes and shows how to convert data of one type to another :

```
#include <iostream>
using namespace std;
class stock2 ;
class stock1
{
  int code , item ;
  float price ;
  public :
```

```
    stock1 ( int a , int b , int c )
    {
      code = a ;
      item = b ;
      price = c ;
    };
    void disp ()
    {
      cout << " code " << code << " \n " ;
      cout << " items " << item << " \n " ;
      cout << " price per item Rs. " << price << " \n " ;
    };
    int getcode ()
    { return code; };
    int getitem ()
    { return item ; };
    int getprice ()
    { return price ; };
    operator float ()
    {
      return ( item*price ) ;
    };
};
class stock2
{
   int code ;
   float val ;
   public :
     stock2 ()
     {
       code = 0;
       val = 0 ;
     };
     stock2( int x , float y )
     {
       code = x ;
       val = y ;
     };
     void disp ()
     {
       cout << " code " << code << " \n " ;
       cout << " total value Rs. " << val << " \n " ;
     };
     stock2( stock1 p )
     {
       code = p.getcode() ;
```

```
        val = p.getitem() * p.getprice() ;
      };
};
int main()
{
  stock1 i1 ( 101 , 10 ,125.0 ) ;
  stock2 i2 ;
  float tot_val = i1;
  i2 = i1 ;
  cout << " Stock Details : Stock 1 type " << " \n " ;
  i1.disp ();
  cout << " Stock Value " << " - " ;
  cout << tot_val << " \n " ;
  cout << " Stock Details : Stock 2 type " << " \n " ;
  i2.disp () ;
  return 0 ;
}
```

5.9 Questions

Q1. What are the basic rules for operator overloading in C++?
Q2. What is dynamic binding?
Q3. Describe the concept of polymorphism
Q4. What is function overloading?
Q5. What are the main applications of function overloading?
Q6. Explain the concept of operator overloading?
Q7. What are the limitations of Operator overloading and Functional overloading?

*****

6

VIRTUAL FUNCTIONS

Chapter Structure :

6.0 Virtual Functions
6.1 Rules of Virtual Function
6.2 Limitations of Virtual Functions
6.3 Need of Virtual Function
6.4 Pure Virtual Function
6.5 Static Functions and Member Functions
6.6 this Pointer
6.7 Abstract Classes
6.8 Virtual Destructor in C++
6.9 Summary
6.10 Reference
6.11 Questions

6.0 Virtual Functions

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
- A 'virtual' is a keyword preceding the normal declaration of a function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.

6.1 Rules of Virtual Function

- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- We cannot have a virtual constructor, but we can have a virtual destructor
- Consider the situation when we don't use the virtual keyword.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions

6.2 Limitations of Virtual Functions

- The function call takes slightly longer due to the virtual mechanism, and it also makes it more difficult for the compiler to optimize because it doesn't know exactly which function is going to be called at compile time.
- In a complex system, virtual functions can make it a little more difficult to figure out where a function is being called from.
- Virtual functions will usually not be inlined.
- Size of object increases due to virtual pointer.

```
1.  #include <iostream.h>
2.  {
3.   public:
4.   virtual void display()
5.   {
6.    cout << "Base class is invoked"<<endl;
7.   }
8.  };
9.  class B:public A
10. {
11.  public:
12.  void display()
13.  {
14.   cout << "Derived Class is invoked"<<endl;
15.  }
16. };
17. int main()
18. {
19.  A* a;    //pointer of base class
20.  B b;     //object of derived class
21.  a = &b;
22.  a->display();   //Late Binding occurs
23. }
```

Output:
Derived Class is invoked

6.3 Need of Virtual Function

The first and foremost question which is arises why do we need virtual function? Suppose we do have a list of pointer to objects of a super class in an inheritance hierarchy and we wish to invoke the functions of its derived classes with the help of single list of pointers provided that the functions in super class and sub classes have

the same name and signature. That in turn means we want to achieve run time polymorphism.

6.4 Pure Virtual Function

- A virtual function is not used for performing any task. It only serves as a placeholder.
- When the function has no definition, such function is known as "**do-nothing**" function.
- The "**do-nothing**" function is known as a **pure virtual function**. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.
- The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

Pure virtual function can be defined as:
virtual void display() = 0;

Example :-

```
1.  #include <iostream.h>
2.  class Base
3.  {
4.     public:
5.     virtual void show() = 0;
6.  };
7.  class Derived : public Base
8.  {
9.     public:
10.    void show()
11.    {
12. std::cout << "Derived class is derived from the base class." << std::endl;
13.    }
14. };
15. int main()
16. {
17.    Base *bptr;
18.    //Base b;
19.    Derived d;
20.    bptr = &d;
21.    bptr->show();
22.    return 0;
23.    }
```

Output:

Derived class is derived from the base class.

6.5 Static Functions and Member Functions

Static functions and members can be simply defined as the functions and members which can be accessed through the class name. We do not need to create the instance/ object of the class in order to access the static members of the class. Static members are per class where as non-static members are per instance. An interesting thing is that value of the static members of the class is same for all objects and if static member is publicly defined, any object of the class can modify the value of static members and this value will be updated for all the instances.

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

class ObjectCounter
{

public:
        static int numofobjects;
public:
        ObjectCounter()
        {
                numofobjects ++;
        }
        static void showcount()
        {
                cout<<"Total number of Objects is: "<<numofobjects<<endl;
        }
};

int ObjectCounter ::numofobjects = 0;

int main()
{
        ObjectCounter::showcount();
        ObjectCounter a, b, c;
        cout<<"After instantiating ObjectCounter Class thrice."<<endl;
        ObjectCounter::showcount();
}
```

6.6 this Pointer

Every object in C++ has access to its own address through an important pointer called **this** pointer. The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object. Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.
Let us try the following example to understand the concept of this pointer "

```
#include <iostream>

using namespace std;

class Box {
  public:
    // Constructor definition
    Box(double l = 2.0, double b = 2.0, double h = 2.0) {
      cout <<"Constructor called." << endl;
      length = l;
      breadth = b;
      height = h;
    }
    double Volume() {
      return length * breadth * height;
    }
    int compare(Box box) {
      return this->Volume() > box.Volume();
    }

  private:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
};

int main(void) {
  Box Box1(5.6, 1.2, 5.1);    // Declare box1
  Box Box2(2.5, 4.1, 6.1);    // Declare box2

  if(Box1.compare(Box2)) {
    cout << "Box2 is smaller than Box1" <<endl;
  } else {
    cout << "Box2 is equal to or larger than Box1" <<endl;
  }
```

```
    return 0;
}
```

Output :
Constructor called.
Constructor called.
Box2 is equal to or larger than Box1

6.7 Abstract Classes

Abstract classes act as a container of general concepts from which more specific classes can be inherited. Thus an abstract class is one that is not used to create any object of its own but it solely exists to act as a base class for the other classes that means the abstract class must be a part of some inheritance hierarchy.

An abstract class can further be illuminated through following points:

- An abstract class can not be instantiated that means abstract classes can not have their own instances but their child or derived classes may have their own instances provided the child class itself is not an abstract class.
- Though objects of an abstract class cannot be created, however, one can use pointers and references to abstract class types.
- A class should contain at least one pure virtual function to be called as abstract. Pure virtual functions can be declared with the keyword virtual and =0 syntax at the end of function declaration statement.
- If a class is made abstract by giving a pure virtual function then it must be inherited by a child class of it that provides the implementation of the pure virtual function.
- If a class inherits an abstract class and does not provide the implementation of the pure virtual function then the child class itself should declare the function as pure virtual that means the child class will be an abstract class as well.
- The signature of the function declared as pure virtual in base class must strictly agree with the signature of the function in child class that implements the pure virtual function.

```
//Abstract base class
class Base
{
   public:
   virtual void show() = 0;    // Pure Virtual Function
};

class Derived : public Base
{
```

```
   public:
   void show()
   {
      cout << "Implementation of Virtual Function in Derived class\n";
   }
};

int main()
{
   Base obj;   //Compile Time Error
   Base *b;
   Derived d;
   b = &d;
   b->show();
}
```

Output :-
Implementation of Virtual Function in Derived class

6.8 Virtual Destructor in C++

A **destructor in C**++ is a member function of a class used to free the space occupied by or delete an object of the class that goes out of scope. A destructor has the same name as the name of the constructor function in a class, but the destructor uses a tilde **(~)** sign before its function name.

Virtual Destructor

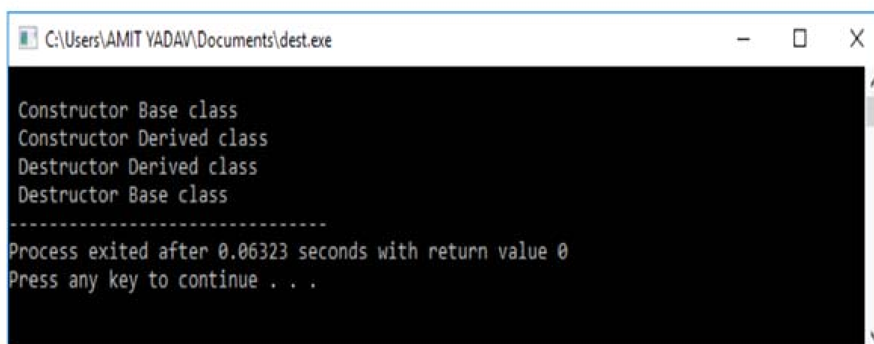
A virtual destructor is used to free up the memory space allocated by the derived class object or instance while deleting instances of the derived class using a base class pointer object. A base or parent class destructor use the **virtual** keyword that ensures both base class and the derived class destructor will be called at run time, but it called the derived class first and then base class to release the space occupied by both destructors.

Why we use virtual destructor in C++?

When an object in the class goes out of scope or the execution of the main() function is about to end, a destructor is automatically called into the program to free up the space occupied by the class' destructor function. When a pointer object of the base class is deleted that points to the derived class, only the parent class destructor is called due to the early bind by the compiler. In this way, it skips calling the derived class' destructor, which leads to memory leaks issue in the program. And when we use virtual keyword preceded by the destructor tilde (~) sign inside the base class, it guarantees that first the derived class' destructor is called. Then the base class' destructor is called to release the space occupied by both destructors in the inheritance class.

```
1.  #include<iostream>
2.  using namespace std;
3.  class Base
4.  {
5.      public:
6.      Base() // Constructor member function.
7.  {
8.      cout << "\n Constructor Base class";  // It prints first.
9.  }
10. virtual ~Base() Define the virtual destructor function to call the Destructor
    Derived function.
11. {
12. cout << "\n Destructor Base class";  /
13. }
14. };
15. // Inheritance concept
16. class Derived: public Base
17. {
18. public:
19. Derived() // Constructor function.
20. {
21. cout << "\n Constructor Derived class" ; / *After print the Constructor Base,
    now it will prints. */
22. }
23.  ~Derived() // Destructor function
24. {
25.     cout << "\n Destructor Derived class"; /*The virtual Base Class?
    Destructor calls it before calling the Base Class Destructor. */
26. }
27. };
28. int main()
29. {
30.     Base *bptr = new Derived; // A pointer object reference the Base class.
31.     delete bptr; // Delete the pointer object.
32. }
```

Output:



6.9 Summary

- Polymorphism simply defines the concept of one name having more than multiple forms.
- It has two types of time compile time and run time. In compile time, an object is bound to its function call at compile time. In run time polymorphism, an appropriate member function is selected while the program is running.
- C++ supports the run time polymorphism with the help of virtual function by using the concept of dynamic binding. Dynamic binding requires use of pointers to objects.
- Run time polymorphism is achieved only when a virtual function is accessed through a pointer to the base class. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class.
- Such virtual functions (equated to zero) are called pure virtual functions. A class containing such pure function is called an abstract class.

FURTHER READINGS :-

- Object Oriented Analysis and Design , Author- Timothy Budd , Publisher - TMH edition - 3rd , Year-2012
- C++ for beginners , Author- B.M Hirwani , Publisher -SPD ,Year-2013
- E. Balagurusamy, *Object Oriented Programming with C++*, Tata Mc-Graw-Hill Publishing Company Limited, New Delhi.

6.10 Reference

www.w3school.com
www.tutorialspoint.com
www.javatpoint.com
www.programiz.com

6.11 Questions

Q1. What are the Rules of Virtual Function ?

Q2. What is virtual function ? Explain with example .

Q3. What are the Limitations of Virtual Functions ?

Q4. What is pure virtual function ? Explain

Q5. Write a note on " this Poinrer" & " Abstract class "

Q6. What is virtual Distructor ? Explain with example.

*****

UNIT – IV

7

INHERITANCE

Unit Structure

7.0 Objectives

7.1 Introduction

7.2 Single Inheritance

7.3 Multilevel Inheritance

7.4 Multiple Inheritance

7.5 Hierarchical Inheritance

7.6 Hybrid Inheritance

7.7 Constructors in Derived Classes

7.8 Summary

7.9 Refernce

7.10 Questions

7.0 Objectives

This chapter would make you understand the following concepts:

- Reusability
- Inheritance
- Single inheritance
- Multiple inheritance
- Multilevel inheritance
- Hybrid inheritance
- Hierarchical inheritance
- Defining a derived class
- Defining derived class constructors

7.1 Introduction

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

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

The derived class inherits some or all of the traits from the base class. A class can ^ inherit properties from more than one class or from more than one level. A derived el&| with only one base class, is called *single inheritance* and one with several base classes 4 called *multiple inheritance*. On the other hand, the traits of one class may be inherited^ more than one class. This process is known as *hierarchical inheritance*. The mechanism *$* deriving a class from another 'derived class' is known as *multilevel inheritance*. Figure 8.1 shows various forms of inheritance that could be used for writing extensible programs. Tfo direction of arrow indicates the direction of inheritance. (Some authors show the arrow® opposite direction meaning <<inherited from<<.)

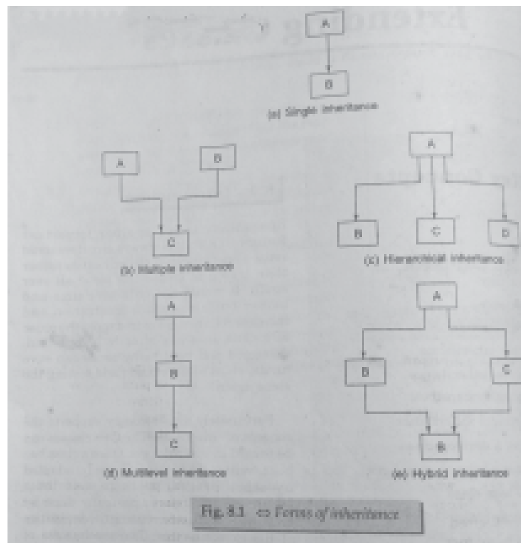


Fig. 8.1 ⇔ Forms of inheritance

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

Examples:

```
class ABC: private XYZ // private derivation
{
members of ABC
};
class ABC: public XYZ         // public derivation
{
members of A8C
};
class ABC: XYZ // private derivation by default
{
members of ABC
};
```

When a base class is privately inherited by a derived class 'public members' of the base class become 'private members' of the derived class and therefore the public members of the base class can only be accessed by the member-functions of the derived class. They are inaccessible to the objects of the derived class. Remember, a

public member of a class can be accessed by its own objects using the dot operator. The result is that no member of the base class is accessible to the objects of the derived class.

On the other hand, when the base class is publicly inherited, 'public members' of the base class become 'public members' of the derived class and therefore they are accessible to the objects of the derived class. In both the cases, the private members are not inherited and therefore, the private members of a base class will never become the members of its derived class.

In inheritance, some of the base class data elements and member functions are 'inherited' into the derived class. We can add our own data and member functions and thus extend the functionality of the base class. Inheritance, when used to modify and extend the capabilities of the existing classes, becomes a very powerful tool for incremental program development.

7.2 Single Inheritance:

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

```
#include <iostream>
using namespace std;
class B
{
int a;
public:
int b;
void get ab();
int get_a(void);
 void show a(void);
};
class D : public B
int c;
public:
void mul(void);
void display(void);
};
//————————————————————————————————
void B :: get_ab(void)
{
a * 5; b = 10;
}
```

```
int B :: get_a()
{
return a;
}
void B :: show_a()
{
cout << <<a = << <<  a << <<\n<<;
}
void D :: mul()
{
c = b * get_a();
}
void D :: display()
{
cout << <<a = << << qet_a() << <<\n<<;
cout << <<b = << << b << <<\n<<;
cout << <<c = << << c << <<\n\n<<;
}
//————————————————————————————————
int main()
{
        D d;
d.get_ab();
d.mul();
d.show_a();
d.display();

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

return 0;
}
```

Program 8.1

Given below is the output of Program 8.1:

```
a = 5
a = 5
b = 10
c = 50

a = 5
b= 20
c = 100
```

The class D is a public derivation of the base class B. Therefore. D inherits all the public members of B and retains their visibility. Thus a public member of the base class B is also a public member of the derived class D. The private members of B cannot be inherited by D. The class D, in effect, will have more members than what it contains at the time of declaration as shown in Fig. 8.2.

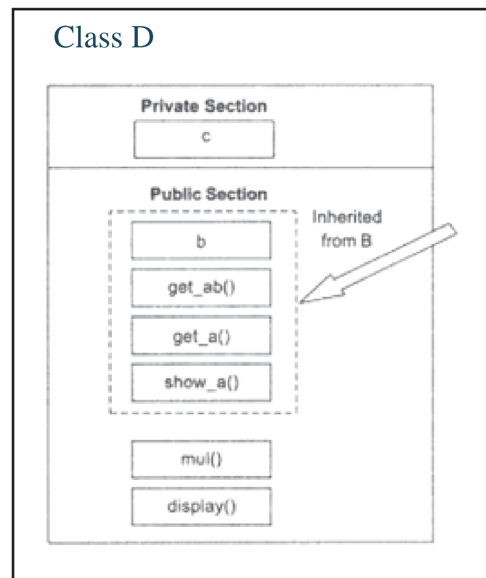


Fig. 8.2 Adding more members to a class (by public derivation)

The program illustrates that the objects of class D have access to all the public members of B. Let us have a look at the functions show_a() and mul():

```
void show a()
{
cout << “a  = “  <<  a  << “\n” <<;
}
void mul ()
{
c = b * get_a();            // c = b * a
}
```

Although the data member a is private in B and cannot be inherited, objects of D are able to access it through an inherited member function of B.

Let us now consider the case of private derivation.

```
class B
{
int a;
public:
int b;
void get_ab();
```

```
void get_a();
void show a();
};
class D : private B        // private derivation
{
int c;
public:
void mul();
void display();
};
```

The membership of the derived class D is shown in Fig. 8.3. In private derivation, the public members of the base class become private members of the derived class. Therefore, the objects of D can not have direct access to the public member functions of B.

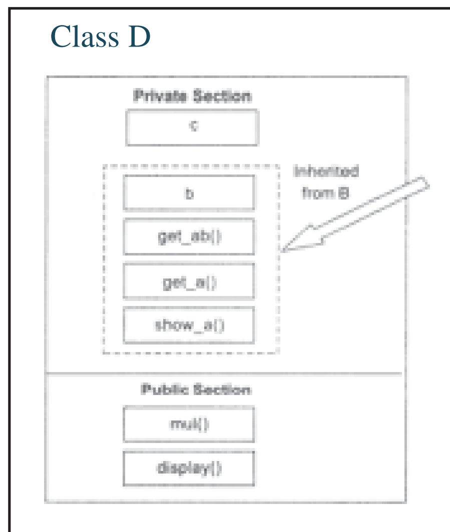


Fig. 8.3 = Adding more members to a class (by private derivation)

The statements such as

```
d.get_ab(); // get_ab() is private
d.get_a();  // so also get_a()
d.show_a();        // and show_a()
```

will not work. However, these functions can be used inside mul() and display() like the normal functions as shown below:

```
void mul()
{
get_ab();
c = b * get a();
void display()
```

```
{
show_a();   // outputs value of 'a'
cout << "b = " << b << "\n"
<< "c = " << c << "\n\n";
}
```

Program 8.2 incorporates these modifications for private derivation. Please compare this with Program 8.1.

SINGLE INHERITANCE : PRIVATE

```
#include <iostream>
using namespace std;
class B
{
int a; // private; not inheritable
public:
int b; // public; ready for inheritance
void get_ab();
int get_a(void);
void show_a(void);
};
class D : private B        // private derivation
{
int c;
public:
void mul(void);
void dlsplay(vold);
};
//———————————————————————————————————————————
void B :: get_ab(void)
{
cout << "Enter values for a and b";
cin >> a >> b;
}
int b :: get_a()
{
        return a;
}
void B :: show_a()
{
        Cout << "a = " << a << "\n";
}
void D :: mul()
{
```

```
        get_ab();
        c = b * get_a();
}
void D :: display()
{
        show_a();
        cout << "b = " << b << "\n"
                << "c = " << c << "\n\n"
}
//————————————————————————————————————————————
int main()
{
        D d;
        // d.get_ab();      WON'T WORK
        d.mul()
        // d.get_a();WON'T WORK
        d.display();
        // d.b = 20; WON'T WORK; b has become private
        d.mul();
        d.display();

        return 0;
}
```

Program 8.2

The output of Program 8.2 would be:

```
Enter values for a and b:5 10
a = 5
b = 10
c = 50
Enter values for a and b:12 20
a = 12
b = 20
c = 240
```

Suppose a base class and a derived class define a function of the same name. What will happen when a derived class object invokes the function?. In such cases, the derived class function supersedes the base class definition. The base class function, will be called only if the derived class does not redefine the function.

Making a Private Member Inheritable

We have just seen how to increase the capabilities of an existing class without modifying it. We have also seen that a private member of a base class cannot be inherited and therefore it is not available for the derived class directly. What do we do if the private data needs to be inherited by a derived class? This can be accomplished by modifying the visibility limit of the private member by making it

public. This would make it accessible to all the other functions of the program, thus taking away the advantage of data hiding.

C++ provides a third visibility modifier, protected, which serve 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 clasees
     Public:
        ........   // visible to all functions
        ........... // in the progrom
};
```

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. It is also ready for further inheritance. 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). Figure 8.4 is the pictorial representation for the two levels of derivation.

Class B

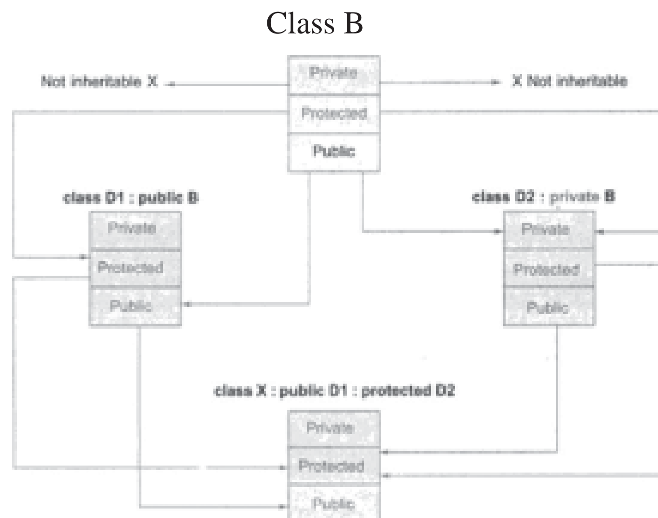


Fig 8.4 Effect of inheritance on the visibility of members

The keywords private, protected, and public may appear in any order and any number of times in the declaration of a class. For example.

```
class beta
{
protected:
        .........
public:
        .........
private:
        .........
public:
        .........
};
```

is a valid class definition.

However, the normal practice is to use them as follows:

```
class beta
{
                ..........              // private by default
                ..........
protected:
        ..........
        ..........
public:
        ..........
}
```

It is also possible to inherit a base class in protected mode (known as protected derivation). In protected derivation, both the public and protected members of the base class become protected members of the derived class. Table 8.1 summarizes how the visibility of base class members undergoes modifications in all the three types of derivation.

Now let us review the access control to the private and protected members of a class. What are the various functions that can have access to these members? They could be:

1. A function that is a friend of the class.
2. A member function of a class that is a friend of the class.
3. A member function of a derived class.

While the friend functions and the member functions of a friend class can have direct access to both the private and protected data, the member functions of a derived class can directly access only the protected data. However, they can access the private data through the member functions of the base class. Figure 8.5 illustrates how the access control mechanism works in various situations. A simplified view of access control to the members of a class is shown in Fig. 8.6.

Base class visibility	Derived class visibility		
	Public derivation	Private derivation	Protected derivation
Private ⟶	Not inherited	Not inherited	Not inherited
Protected ⟶	Protected	Private	Protected
Public ⟶	Public	Private	Protected

class X

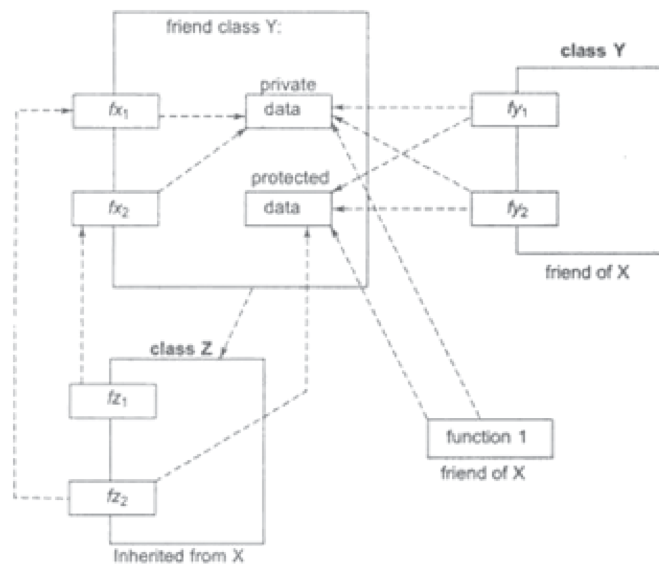


Fig. 8.5 <=> Access mechanism in classes

7.3 Multilevel Inheritance

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

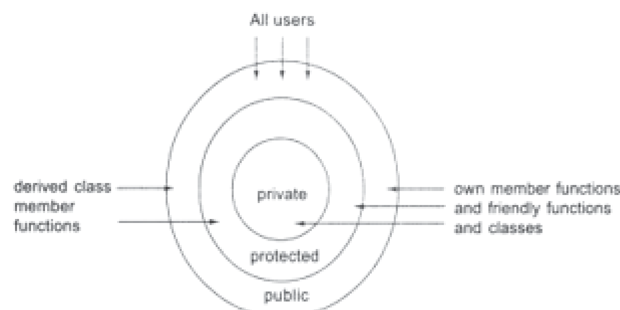


Fig. 8.6 ⇔ *A simple view of access control to the members of a class*

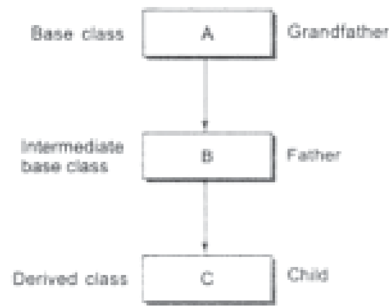


Fig. 8.7 ⇔ Multilevel inheritance

A derived class with multilevel inheritance is declared as follows:

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

This process can be extended to any number of levels.

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

```
class student
{
        protected:
          int roll_number;
        public:
          void getnumber(int);
          void put_number(void);
};
void student :: get number(int a)
{
        roll number = a;
}
void student :: put_number()
{
        cout « "Roll Number: " « roll number « "\n";
}
class test : public student                         // First level
derivation
{
        protected:
          float sub1;
          float sub2;
```

```
        public:
          void getmarks(float, float);
          void put marks(void);
};
void test :: get marks(float x, float y)
{
        Sub1= x;
        sub2 = y;
}
void test :: put marks()
{
        cout « "Marks in SUB1 = " « subl « "\n";
        cout « "Marks in SUB2 = " « sub2 « "\n";
}
class result : public test                          // Second level
derivation
{
          float total;  // private by default
        public:
          void display(void);
};
```

The class result, after inheritance from 'grandfather' through 'father', would contain the following members:

```
private:
        float total;                    // own member
protected:
        Int roll_number;                // inherited from student via test
        float sub1;                     // inherited from test
        float sub2;                     // inherited from test
public:
        void get_number(int);           // from student via test
        void put_number(void);          // from student via test
        void get _marks(float, float);      // from test
        void put_marks(void);               // from test
         void display(void);                // own member
```

The inherited functions put_number() and put _marks() can be used in the definition of display () function:

```
void result :: display(void)
{
        total = sub1 + sub2;
        put_number();
        put_marks();
        cout « 'Total = " « total « "\n";
```

```
}
Here is a simple main() program:
int ma1n()
{
                result student1;                // student 1 created
                student1.get_number(111);
                studentl.get_marks(75.0, 59.5);
                studentl.display();
return 0;
}
```

This will display the result of student1. The complete program is shown in Program 8.3.

MULTILEVEL INHERITANCE

```
#include <iostream>
using  namespace std;
class student
{
                protected:
                  int roll number;
                public: .
                  void getnumber(int);
                  void put_number(void);
};
void student :: get number(int a)
{
                roll_number = a;
}
void student :: put number()
{
                cout « "Roll Number: " « roll number « "\n";
}
class test : public student                     // First level derivation
(
                protected:
                  float sub1;
                  float sub2;
                public:
                  void get_marks(f1oat, float);
                  void put_marks(void);
};
void test :: getmarks(float x, float y)
{
                sub1 = x;
```

```
            sub2 = y;
}
void test :: put_marks()
{
            cout « "Marks in SUB1 = " « subl « "\n";
            cout « "Marks in SUB2 = "« sub2 « "\n";
}
class result : public test                    // Second level derivation
{
             float total;              // private by default
            public:
             void display(void);
};
void result :: display(void)
{
            total = subl + $ub2;
            put_number();
            put_marks();
            cout « "Total = * « total « "\n";
}
int main()
(
            result student1;                 // student 1 created
            student1.get nuntber(111);
             student1.get_marks(75.0, 59.5);
            student1.display();
return 0;
}
```

Program 8.3

Program 8.3 displays the following output:

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

7.4 Multiple Inheritance

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

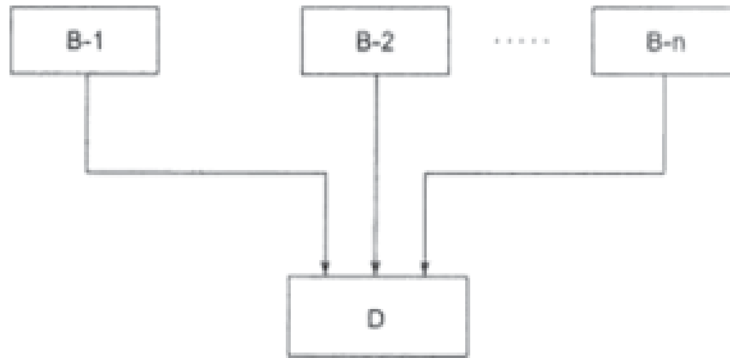


Fig. 8.8 ⇔ *Multiple inheritance*

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

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

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

Example:

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

Classes M and N have been specified as follows:

```
class M
{
protected:
int m;
public:
void get_m(int);
};
void M :: get_m(int x)
{
m = x;
)
class N
```

```
{
protected:
int n;
public:
void getn(int);
};
void N :: get_n(int y)
{
n = y;
}
```

The derived class P, as declared above, would, in effect, contain all the members of M and N in addition to its own members as shown below:

```
class P
{
        protected:
                int m;                                  // from M
int n;                                  // from N
        public:
                void get_m(int);                // from M
void getn(int);                         // from N
void display(void);                     // own member
};
```

The member function displayf) can be defined as follows:

```
void P :: display(void)
{
cout « “m “ « m « “\n”;
cout « “n  “ « n « “\n”;
cout « “m*n =” « m*n « “\n”;
};
```

The main() function which provides the user-interface may be written as follows:

```
main()
{
P p;
p.getm(10);
p.get_n(20);
p.display();
}
```

Program 8.4 shows the entire code illustrating how all the three classes are implemented in multiple inheritance mode.

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

```
class M
{
protected:
int m;
public:
 void get_m(int);
};
class N
{
protected:
 int n;
public:
void get n(int);
};
class P : public M, public N
{
public:
void display(void);
};
void M :: get_m(int x)
{
m = x;
}
void N :: get_n(int y)
{
n = y;
}
void P :: dlsplay(void)
{
cout «     "m = "     « m « *\n";
cout «     "n = "     « n « "\n";
cout «     "m*n = " « m*n « "\n";
}
int main()
{
P p;
p.get_m(10);
p.get_n(20);
p.display();

return 0;
}
```

PROGRAM 8.4

The output of Program 8.4 would be:

m = 10
n = 20
m*n = 200

7.5 Hierarchical Inheritance

We have discussed so far how inheritance can be used to modify a class when it did not satisfy the requirements of a particular problem on hand. Additional members are added through inheritance to extend the capabilities of a class. Another interesting application of inheritance is to use it as a support to the hierarchical design of a program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below that level.
As an example, Fig. 8.9 shows a hierarchical classification of students in a university. Another example could be the classification of accounts in a commercial bank as shown in Fig. 8.10. All the students have certain things in common and, similarly, all the accounts possess certain common features.

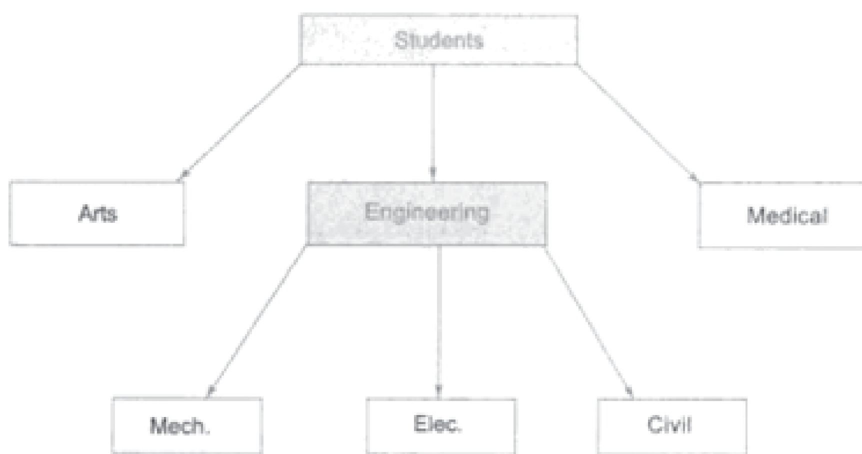


Fig. 8.9 o Hierarchical classification of students

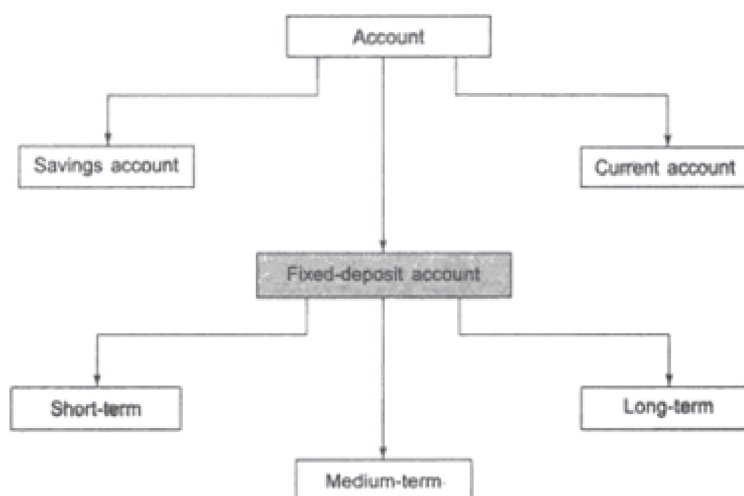


Fig. 8.10 ⇔ Classification of bank accounts

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

7.6 Hybrid Inheritance

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

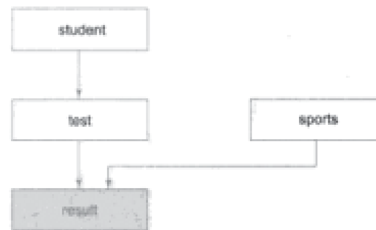


Fig. 8.11 ⇔ Multilevel, multiple inheritance

The sports class might look like:

```
class sports
{
protected:
float score;
public:
void get_score(float);
void put_score(void);
}
```

The result will have both the multilevel and multiple inheritances and its declaration would be as follows:

```
class result : public test, public sports
{
        ..........
        ..........
};
```

Where test itself is a derived class from student. That is

```
class test : public student
{
        .............
        .............
}
```

Program 8.5 illustrates the implementation of both multilevel and multiple inheritance.

```
#include <iostream>
using namespace std;
class student
{
protected:
int roll_number;
public:
void get_number(int a)
{
roll number = a;
                }
                void put_number(void)
                {
                        cout « "Roll No: " « roll number « "\n";
}
};
class test : public student
{
protected:
float part1, part2;
public:
void get_marks (float x, float y)
{
        part1 = x;
        part2 = y;
}
void put marks(void)
{
        cout  « "Marks obtained: " « "\n"
              « "Part1 = "« part1 « "\n"
              «"Part2 = "« part2 « "\n"
}
};
class sports
{
        protected:
float score;
        public:
void get_score(float s)
{
        Score = s;
}
void put_score(void)
{
```

```
        cout « “Sports wt: “ « score « “\n\n”;
}
};
class result : public test, public sports
{
                float total;
        public:
void display(void);
};
void result :: display(void)
{
total = part1 + part2 + score;
put_number();
put_marks();
put_score();
cout « ‘Total Score: “ « total « “\n”;
}
int main()
{
result student_1;
 student_1.get_number(1234);
student_1.get_marks(27.5, 33.0);
student_1.get_score(6.0);
student_1.display();
return 0;
}
```

PROGRAM 8.5

Here is the output of Program 8.5:

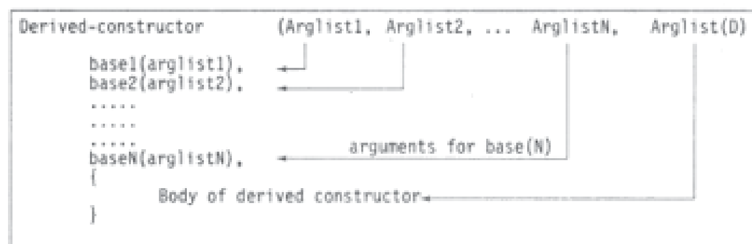
```
Roll No: 1234
Marks obtained:
Part1 = 27.5
Part2 = 33
Sports wt: 6

Total Score: 66.5
```

7.7 Constructors in Derived Classes

As we know, the constructors play an important role in initializing objects. We did not use them earlier in the derived classes for the sake of simplicity. One important thing to note here is that, as long as no base class constructor takes any arguments, the derived class need not have a constructor function. However, if any base class contains a constructor with one or more arguments, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructors. Remember, while applying inheritance we usually create objects using

the derived class. Thus, it makes sense for the derived class to pass arguments to the base class constructor. When both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in the derived class is executed.
In case of multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class. Similarly, in a multilevel inheritance, the constructors will be executed in the order of inheritance.
Since the derived class takes the responsibility of supplying initial values to its base classes, we supply the initial values that are required by all the classes together, when a derived class object is declared. How are they passed to the base class constructors so that they can do their job? C++ supports a special argument passing mechanism for such situations.
The constructor of the derived class receives the entire list of values as its arguments and passes them on to the base constructors in the order in which they are declared in the derived class. The base constructors are called and executed before executing the statements in the body of the derived constructor.
The general form of defining a derived constructor is:



The header line of derived-constructor function contains two parts separated by a colon(:).The header line of derived-constructor function contains two parts separated by a colon!:). The first part provides the declaration of the arguments that are passed to the derived- constructor and the second part lists the function calls to the base constructors.
base1(arglist1), base2(arglist2) ... are function calls to base constructors base1(), base2(), ... and therefore arglist1, arglist2, ... etc. represent the actual parameters that are passed to the base constructors. Arglist1 through ArglistN are the argument declarations for base constructors base through baseN. ArglistD provides the parameters that are necessary to initialize the members of the derived class.

CONSTRUCTORS IN DERIVED CLASS

```
# include <iostream>
using namespace std;
class alpha
{
        int x;
    public:
        alpha(int i)
        {
```

```
                X = i;
                cout << "alpha initialized \n";
                }
                void show_x(void)
                {
                cout << "x = " << x << "\n";
                }
};
class beta
{
                float y;
            public:
                beta(float j)
                {
                        y = j;
                }
                void show_y(void)
                {
                        cout << "y = " << y << "\n";
                }
};
class gamma: public beta, public alpha
{
                int m, n;
            public:
                gamma(int a, float b, int c, int d):alpha(a), beta(b)
                {
                        m = c;
                        n = d;
                        cout << "gamma initialized \n";
                }
                void show_mn(void)
                {
                        cout << "m = " << m << "\n"
                               << "n = " <<n << "\n"
                }
};
int main()
{
            Gamma g(5, 10.75, 20, 30);
            cout << "\n";
            g.show_x();
            g.show_y();
            g.show_mn();

            return 0;
}
```

PROGRAM 8.7
The output of Program 8.7 would be:
beta initialized
alpha initialized
gamma initialized
x = 5
y = 10.75
m = 20
n = 30

7.9 Summary

- The mechanism of deriving a new class from an old class is called inheritance.
- The derived class inherits some or all of the properties of the base class.
- A derived class with only one base class is called single inheritance.
- A class can inherit properties from more than one class which is known as multiple inheritance.
- A class can be derived from another derived class which is known as multilevel inheritance.
- When the properties of one class are inherited by more than one class, it is called hierarchical inheritance.
- A private member of a class cannot be inherited either in public mode or in private mode.
- A protected member inherited in public mode becomes protected, whereas inherited in private mode becomes private in the derived class.
- A public member inherited in public mode becomes public, whereas inherited in private mode becomes private in the derived class.
- The friend functions and the member functions of a friend class can directly access the private and protected data.
- In multilevel inheritance, the constructors are executed in the order of inheritance.

7.9 List of Reference

1. Object Oriented Programming with C++, Fourth Edition, E Balgurusamy

7.10 Model Questions

- What does inheritance means?
- What are the different forms of inheritance? Give an example for each.
- Describe the syntax of the single inheritance in C++.
- When do we use the protected visibility specifier to a class member.
- Describe the syntax of multiple inheritance. When do we use such an Inheritance?
- What is an abstract class?
- In what order are the class constructors called? When a derived class object is created?

*****

8

EXCEPTION HANDLING

Chapter Structure :

8.0 Objective
8.1 Introduction
8.2 Basics of Exception handling.
8.3 Exception handling mechanism
8.4 Throwing Mechanism
8.5 Catching mechanism
8.6 Multiple catch statements
8.7 Catch all exceptions
8.8 Rethrowing an exception
8.9 Summary
8.10 Reference
8.11 Questions

8.0 Objectives:

This chapter would make you understand the following concepts:

- Basics of Exception handling
- exception handling mechanism
- Throwing Mechanism
- Catching mechanism
- Multiple catch statements
- Catch all exceptions
- Rethrowing an exception

8.1 Introduction

We know that it is very rare that program works correctly first time. It might have bugs. The 2 most common types of bugs are logic errors and syntactic errors. The logic errors are due to poor understanding of problem and solution procedure.

We often come across some particular problems other than logic or syntax error. They are known as exceptions. Exception are runtime anomalies or can use your condition that program may encounter while executing. Anomalies might include conditions such as division by zero access to an array outside of its bounds, or running out of memory or disk space. When a program in counters and exceptional condition, it is important that it is identified and dealt with effectively . ANSI C ++ provide built in language features to detect and handle exceptions which are basically runtime errors.

Exception handling was not part of the original C++. it is a new feature added to ANSI C++. Exception handling provides a type safe, integrated approach, for coughing with the unusual predictable problems that arise while executing a program.

8.2 Basics of Exception handling.

Exceptions are of 2 kinds, namely, synchronous exception and asynchronous exceptions. Errors such as "Out of range index" and "Overflow" Belong to the synchronous type of exceptions. The errors that are caused by events beyond the control of the program (Search ask keyboard interrupts) are called asynchronous exceptions for the propose. Exception handling mechanism in C++ Is designed to handle only synchronous exceptions.

The purpose of the exception handling mechanism is to provide means to detect and report exceptional circumstances so that appropriate action can be taken. The mechanism suggests a separate error handling code that performs the following tasks:
1. Find the problem (Hit the exception).
2. Inform that the error has Occurred (Throw the exception).
3. Receive the error information (Catch the exception).
4. Take corrective actions (Handle the exception).
The error handling code basically consists of two segments, one to detect errors and to throw exceptions, and the other to catch the exceptions and to take appropriate actions.

8.3 exception handling mechanism

C++ exception handling mechanism is basically built upon three keywords namely, try throw and catch. The keyword try is to use to preface a block of statements (surrounded by braces) which may generate exceptions. This block of statements is known as try block. When an exception is detected, it is throw using the throw statement in the try block. Attach block Defined by the keyword catch catches the exception thrown by the throw statement in the try block, and handle it appropriately. The relationship is shown in the figure

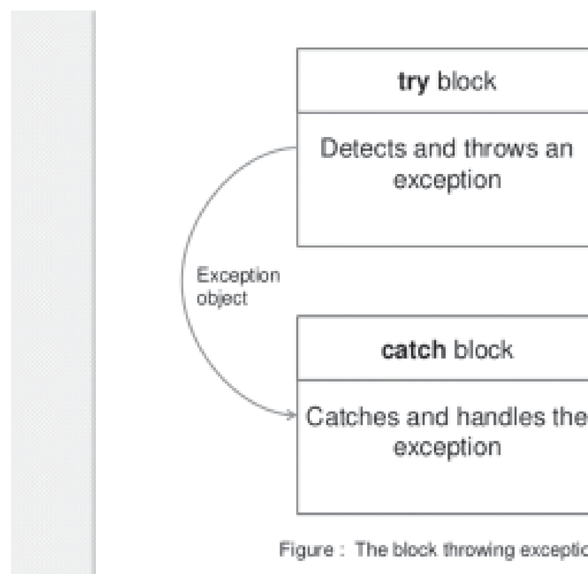


Figure : The block throwing exception

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

```
.....
.....
try
{
    ......
    throw exception;            //Block of statements which_open_mode
    ......                          //Detect and throw the exeptions
    ......
}
catch(type arg)                         //catches exception
{
    ......
    ......                          //Block of statements that
    ......                          //Handles the exceptions
}
......
......
```

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

Try Block Throwing an Exception

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

int main()
{
        int a, b;
        cout << “Enter Values of a and b \n”;
        cin >> a;
        cin >> b;
        int x = a - b;
        try
        {
                if(x != 0)
                {
                cout << “Result(a/x)=” << a/x << “\n”;
```

```
                }
                else                // There is an exceptions
                {
                throw(x);           // Throws int object
                }
        }
        catch(int i)
        {
                cout << "Exception caught: x = " << x << "\n";
        }
        cout << "END";
        return 0;
}
```

The output of Program:
First Run

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

Second Run

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

Program details and catches a division by zero problems. When exception is thrown, the catch blocks the skipped and execution resume with the first line after the catch. In the second Run, the denominator x becomes zero and therefore a division by zero situation occurs. This exception is thrown using the object x. Since the exception object is an int type, the catch statement containing int type argument catches the exception and displays necessary message.
most often coma exceptions are thrown by functions that are invoked from within the try blocks. The point at which the throw is executed is called the throw point. Once an exception is thrown to the catch block, control cannot return to the throw point. This kind of relationship is shown in figure.

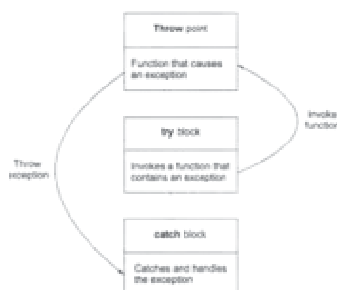


Figure: function invoked by try block throwing exception

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

Invoking function that generate the exception
Invoking function that generates exception

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

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

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

The output of the Program

We are inside the try block

We are inside the function
Result = -3

We are inside the function
Caught the exception

8.4 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 to 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 an exception is thrown, it will be caught by the catch statement associated with the try block. That is the control exits the current try block, and is transfered to the catch block after that try block.

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.

8.5 Catching mechanism

As stated earlier coma code for handling exceptions is included in catch blocks a catch block looks like a function definition and is of the form.

The type indicates the type of exception that catch block handles. The parameter arg is an optional parameter name. Note that the exception handling code is placed

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

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

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

8.6 Multiple catch statements

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

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

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

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

Program shows simple examples where multiple catch statements are used to handle various types of exceptions.

Multiple Catch Statements

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

void test(int x)
{
        try
        {
            if(x==1) throw x;               //int
            else
                if(x==0) throw 'x';             //char
            else
                if(x==-1) throw 1.0;            //double
            cout << "End of the 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 << "End of the try catch system \n\n";
}
int main()
{
        cout "Testing multiple catches \n";
        cout << "x == 1 \n";
        test(1);
        cout << "x == 0 \n";
        test(0);
        cout << "x == -1 \n";
        test(-1);
        cout << "x == 2 \n";
```

```
            test(2);

            return 0;
}
```

The output of the program

```
Testing Multiple Catches
X == 1
Caught an integer
End of the try-catch system

X == 0
Caught a character
End of the try-catch system

X == -1
Caught a double
End of the try-catch system

X == 2
End of try block
End of the try-catch system
```

The program when executed first, in the function test() with x=1 and therefore throws x an int exception. This matches the type of parameter m in catch2 and therefore catch2 handler is executed. Finally, the handler catch3 is executed when a double type exception is thrown. Note that every time only the handler which catches the exception is executed and all other handler is bypassed.

When the try block does not throw any exceptions and it completes the normal execution, control passes to the first statement after the last catch handler associated with that try block.

8.7 Catch all exceptions

In some situations, we may not be able to anticipate all possible types of exceptions and therefore may not be able to design independent catch handler to catch them. In such circumstances, we can force a catch statement to catch all exceptions instead of a certain type alone. This could be achieved by defining the catch statement using ellipses as follows:

Catching all exceptions

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

```
void test(int x)
{
        try
        {
        if(x == 0) throw x;              //int
        if(x == -1) throw 'x';           //char
        if(x == 1) throw 1.0;            //float
        }
        catch(...)                       //catch all
        {
        cout << "Caught an exception \n";
        }
}
int main()
{
        cout << "Testing generic catch \n";
        test(-1);
        test(0);
        test(1);

        return 0;
}
```

The output of the program:

```
Testing Generic catch
Caught an exception
Caught an exception
Caught an exception
```

It may be a good idea to use the catch(...) as a default statement along with other catch handler so that it can catch all exceptions which are not handled explicitly.

8.8 Rethrowing an exception

A handler may decide to rethrow the exception caught without processing it. in such situations, will make simply in walk through without any arguments as shown below:
throw;

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

Rethrowing an exception

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

void divide(double x, double y)
{
        cout << "Inside function \n";
        try
        {
        if(y == 0.0)
            throw y;            //Throwing table
        else
            cout << "Division =" << x/y << "\n";
        }
        catch(double)
        {
        cout << "Caught double inside function \n";
        throw;                  // Rethrowing double
        }
        cout << "End of function \n\n";
}
int main()
{
        cout << "Inside main \n";
        try
        {
        divide(10.5,2.0);
        divide(20.0,0.0);
        }
        catch(double)
        {
        cout << "Caught double inside main \n";
        }
        cout << "End of main \n";
        return 0;
}
```

The output of the program

Inside Main
Inside function
Division = 5.25
End of the function

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

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

Attach handler itself may detect and throw an exception to stop here again; the exception thrown will not be caused by any catch statements in that group. it will be passed on to the next outer try / catch sequence for processing.

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

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

The type-list specifies the type of exceptions that may be thrown. Throwing any other type of exception will cause abnormal program termination. if you wish to prevent a function from throwing any exception, we may do so by making the type-list empty. That is, we must use
throw() ; // empty list
In the function header line.

Program demonstrates how we can restrict a function to throw only certain types and not all.

Testing throw restrictions

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

void test(int x) throw(int, double)
{
        if(x == 0) throw 'x';                    //char
        else
        if(x == 1) throw x;                      //int
```

```
        else
        if(x == -1) throw 1.0;              //double
        cout << "End of function block \n";
}
int main()
{
        try
        {
        cout << "Testing throw restrictions \n";
        cout << "x == 0 \n";
        test(0);
        cout << "x == 1 \n";
        test(1);
        cout << "x == -1 \n";
        test(-1);
        cout << "x == 2 \n";
        test(2);
        }
        catch(char c)
        {
        cout << "Caught a character \n";
        }
        catch(int m)
        {
        cout << "Caught an integer \n";
        }
        catch(double d)
        {
        cout << "Caught a double \n";
        }
        cout << "End of the try-catch system \n\n";
        return 0;
}
```

The output of the program
Testing throw restrictions
X == 0
Caught a character
End of the try-catch system

8.9 Summary

- Exceptions are peculiar problems that a program may encounter at run time.
- Exceptions are of two types: synchronous and asynchronous. C++ provides mechanism for handling synchronous exceptions.
- An exception is typically caused by a faulty statement in a try block. The statement discovers the error and throws it, which is caught by a catch statement.
- The catch statement defines a block of statements to handle the exception appropriately.
- When an exception is not caught, the program is aborted.
- A try block may through an exception directly or invoke a function that throws an exception. Irrespective of location of the throw point, the catch block is placed immediately after the try block.
- We can place two or more catch block together to catch and handle multiple types of exceptions thrown by try block
- It is also possible to make a catch statement to catch all types of exceptions using ellipses as its arguments.
- We may also restrict a function to throw only a set of specified exceptions by adding a throw specification clause to the function definition.

8.10 List of Reference

1. Object Oriented Programming with C++, Fourth Edition, E Balgurusamy

8.11 Model Questions

Q.1. What is an exception?

Q.2. How is an exception handle in C++?

Q.3. What are the advantages of using exception handling mechanism in a program?

Q.4. When should a program through an exception?

Q.5. When is the catch() handler is used?

Q.6. What is an exception specification? When it is used?

Q.7. What should be placed inside a try block?

Q.8. What should be placed inside a catch block?

Q.9. When do we used multiple catch statements?

*****

UNIT – V

9

TEMPLATES

Unit Structrure :

9.0 Objectives
9.1 Introduction
9.2 Function Templates
  9.2.1 Template for a Simple Function
  9.2.2 Function Templates with Multiple Arguments
  9.2.3 Template Function Overloading
9.3 Class Templates
  9.3.1 Class Templates Example
9.4 Summary
9.5 References
9.6 Bibliography
9.7 Questions

9.0 Objectives

The objectives of this chapter are to:

- Explain the concepts of Templates
- Explain the concepts of Function Templates
- Explain the concepts of Class Templates

9.1 Introduction

The C++ programming language has a feature that allows functions and classes to work with generic types. This allows a method or class to operate with a variety of data types without having to rewrite it.

It enables you to define generic classes and functions, allowing you to support generic programming. Generic programming is a technique that uses generic types as parameters in algorithms to enable them to function with a wide range of data types. The basic idea is to transfer the data type as a parameter, eliminating the need to write the same code for different data types.

To support templates, C++ introduces two additional keywords: 'template' and 'typename'. The keyword 'class' can always be used in place of the second keyword. The definition of templates can be implemented in two ways:

- FunctionTemplates
- Class Templates

9.2 Function Templates

A feature template functions similarly to a regular function, with one major exception. A feature template is similar to a function, except that the template may take several different types of arguments. A function template, in other words, describes a set of functions.

What is the best way to declare a feature template?

A function template begins with the keyword template, then template parameters within <>, and finally function declaration.

```
template <class T>
T someFunction(T arg)
{
  // statements.
}
```

T: This is a substitute for a data form that the function uses. It's used in the description of the function. It's just a placeholder, and the compiler will automatically overwrite it with the correct data form.
In a template declaration, the class keyword is used to define a generic type.

A single function template can handle multiple data types at the same time, while a single normal function can only handle one set of data types.
When performing identical operations on two or more types of data, you usually use function overloading to create two functions with the required function declaration. However, using function templates is a safer option since you can accomplish the same task with less code that is easier to maintain.

9.2.1 Template for a Simple Function

The first example illustrates how to write our min-value function as a prototype that can be used for any basic numerical form. This program creates a template version of min() and then calls it with various data types in main() to demonstrate that it works.

Example for simple function template

```
#include <iostream>
using namespace std;
template <typename T>
T minfun(T x, T y)
{
  return (x > y)? y: x;
}
int main()
{
```

```
  cout << minfun <int>(5, 10) << endl;  // Call minfun for int
  cout << minfun <double>(10.0, 8.0) << endl; // call minfun for double
  cout << minfun <char>('b', 'm') << endl;   // call minfun for char
  return 0;
}
```

Output –

5
8.0
b

Explation

- In the preceding programme, the function template minfun() is defined, which takes two T-type arguments, x and y. The letter T denotes that the statement can be of any data form.
- Using a simple conditional operation, the minfun() function returns the smallest of the two arguments.
- Variables of three different data types are declared within the main() function: int, float, and char. The variables are then transferred as regular functions to the minfun() function prototype.
- When an integer is passed to the template function during runtime, the compiler recognises that it must generate a minfun() function to accept the int arguments and does so.
- Similarly, it recognises the argument data types and produces the minfun() function accordingly when floating-point and char data are transferred.
- This approach substituted three identical normal functions with a single function prototype, making the code more maintainable.

9.2.2 Function Templates with Multiple Arguments

Let's take a look at another feature template. This one needs three arguments: two template arguments and one basic type statement.

Example for more than one argument

```
#include <iostream>
using namespace std;
template<class T> T add(T &a,T &b)
{
   T result = a*b;
   return result;
}
int main()
{
  int i =2;
  int j =3;
```

```
  float m = 2.3;
  float n = 1.2;
  cout<<"Multiplication of i and j is :"<<add(i,j);
  cout<<'\n';
  cout<<" Multiplication of m and n is :"<<add(m,n);
  return 0;
}
```

Output –
Multiplication of i and j is :5
Multiplication of m and n is :3.5

9.2.3 Template Function Overloading:

Function template overloading is when the names of the function templates are the same but they are named with different arguments.

The name of the function remains the same if the function template is with the ordinary template, but the number of parameters varies.

The function name stays the same when a function template is overloaded with a non-template function, but the arguments are different.

The following programme shows how to overload a template function with an explicit function:

Example

```
template < class gentype > void swap ( gentype &x, gentype &y )
{
gentype t;
t = x;
x = y;
y = t;
}
void main ()
{
char ch1, ch2;
/* same as in the previous program*\
……
}
```

These types of functions are referred to as "function templates."
When a call to a template function is encountered, the compiler internally recreates a copy of the function from the template with the actual data type passed and executes it.

The user is unaware of the internally created function, which is referred to as a "template templates" A function templates eliminates the time and effort of rewriting functions of the same body but different data types.

Generic functions also aid in the implementation of the c++ programming principle of "one interface, multiple methods."
A comma-separated list can be used to describe several generic data types with the template argument.

9.3 Class Templates

You may sometimes need a class implementation that is the same for all classes with the exception of the data types used.

Normally, you'd have to make a separate class for and data form or separate member variables and functions within a single class. This would unnecessarily bloat the code base and make it difficult to maintain, because any modification to one class or function should be applied to all classes and functions. Class templates, on the other hand, make it possible to reuse the same code for all data types.

Class Templates are similar to Function Templates in that they can be described in the same way. When a class uses the Template definition, it is referred to as a generic class.
What is the best way to declare a class template?

```
template <class T>
class className
{
  ... .. ...
public:
  T var;
  T someOperation(T arg);
  ... .. ...
};
```

T is the template argument in the above declaration, which is a placeholder for the data type used.
A member variable var and a member function memfun() are both of type T within the class body.
What is the best way to make a class template object?
When creating a class template object, you must specify the data type within a < >.

```
className<dataType> classObject;
For example:
className<int> classObject;
className<float> classObject;
className<string> classObject;
```

9.3.1 Class Templates Example

```
#include <iostream>
using namespace std;
template <class T>
class Compute
{
private:
        T a, b;
public:
        Compute (T i, T j)
        {
                a = i;
                b = j;
        }
        void funDisplay()
        {
                cout << “Numbers are: “ << a << “ and “ << b << “.” << endl;
                cout << “Addition is: “ << add () << endl;
                cout << “Subtraction is: “ << sub() << endl;
                cout << “Product is: “ << mul () << endl;
                cout << “Division is: “ << div () << endl;
        }
        T add()
{
        return a + b;
}
        T sub()
{
 return a - b;
}
        T mul()
{
 return a * b;
}
        T div()
{
return a / b;
}
};

int main()
{
        Compute <int> intCompute (2, 1);
        Compute <float> floatCompute (2.4, 1.2);
```

```
        cout << "Int results:" << endl;
        intCompute.funDisplay ();

        cout << endl << "Float results:" << endl;
        floatCompute. funDisplay ();

        return 0;
}
```

Output –
Int results:
Numbers are: 2 and 1.
Addition is: 3
Subtraction is: 1
Product is: 2
Division is: 2

Float results:
Numbers are: 2.4 and 1.2.
Addition is: 3.6
Subtraction is: 1.2
Product is: 2.88
Division is: 2

Explanation :

- A class template named Compute is declared in the preceding programme.
- The class has two private members of type T: a& b, as well as a constructor to initalize the members.
- It also includes public member functions for adding, subtracting, multiplying, and dividing numbers, which return the value of the data type specified by the consumer. A function called funDisplay() is used to view the final output on the computer.
- Two separate Compute objects, intCompute and floatCompute, are generated in the main() function for data types int and float, respectively. The constructor is used to set the values.

Example program for class template

```
#include <iostream>
using namespace std;
template<class T1, class T2>
class Add
{
        T1 i;
        T2 j;
```

```
public:
Add(T1 a,T2 b)
{
  i = a;
  j = b;
}
  void funDisplay()
  {
      cout << "Values of no1 and no 2 are : " << i<<" ,"<<j<<endl;
  }
};
int main()
{
    Add<int,float> d(20,10.5);
    d.funDisplay();
    return 0;
}
```

Output –
Values of no1 and no2 are : 20,10.5

Advantages and Disadvantages of Templates.

Advantages
Use templates in situations that result in duplication of the same code for multiple types.

Disadvantages

- It can be difficult to debug code that is developed using templates. Since the compiler replaces the templates, it becomes difficult for the debugger to locate the code at runtime.
- Compilation time is too much.

9.4 Summary

- Templates help developers create a set of functions or classes to handle a variety of data types.
- If you often write several similar functions that perform the same operation on different data types, consider using a function template instead.
- Similarly, if you find yourself writing several different class requirements that vary only in the form of data acted on, a class template may be useful.
- You'll save time, and the end result will be a more stable and easy-to-maintain application that's much easier to understand.

9.5 List of references

1. The Complete Reference-C++,4th Edition. Herbert Schildt,Tata McGraw- Hill
2. The C++ Programming Language, 4th Edition, Bjarne Stroustrup, AddisonWesly
3. Absolute C++,4th Edition, Walter Savitch,Pearson Education

9.6 Bibliography

1. https://www.tutorialspoint.com
2. https://www.geeksforgeeks.org
3. https://www.javatpoint.com
4. https://guru99.com
5. www.slideshare.net

9.7 Questions

Q.1. What are templates? Give an overview of Class Templates & Function Templates.

Q.2. What is Class Templates and explain with an example

Q.3. What is Function Templates and explain with an example

Q.4. Explain Templates with an example how Multiple Parameters of different type can be used with Templates

Q.5. Explain Overloading of Template Functions with an example

*****

10

WORKING WITH FILES

Unit Structure

10.0 Objectives

10.1 Introduction

10.2 Files Operations

    9.2.1 Template for a Simple Function

    9.2.2 Function Templates with Multiple Arguments

    9.2.3 Template Function Overloading

10.3 Stream classes

10.4 File Pointer and their manipulation

10.5 Summary

10.6 References

10.7 Bibliography

10.8 Questions

10.0 Objectives

The following concepts will be discussed in this chapter :

- To be able to recognise different kinds of files.
- To get a better understanding of the various C++ stream classes
- The hierarchy of stream classes must be comprehended.
- To achieve a deeper understanding of how to use C++ to execute file-related tasks

10.1 Introduction

Files keep data in a storage unit for a long time. The output of a programme can be saved in a file using file handling. While the data is in the file, it can be subjected to a variety of operations.

A stream is a representation of a computer that performs input/output operations. A stream can be interpreted as either a destination or a source of indefinitely long characters. This can be decided by the way they are used. C++ comes with a library that includes methods for dealing with files.

A file is a set of related or identical data stored as a series of bytes on a disc. A file is generated to store data forever. A file is typically used to store vast amounts of data in real-world applications.

In a framework, there are two types of files.

Text documents (ASCII)

Files in binary format

Binary files are made up of a collection of bytes, or eight-bit ordered groupings. These bytes are organised by a developer into a format that stores the information needed for the custom application. Different types of data, such as image, video, and audio data, may be included in the same binary file format.

Only textual data is stored in text files. They do, however, have less chances of being corrupted than binary files. While in a binary file, a small error can render it unreadable.

The main difference between text files and binary files is that text files perform different character translations, such as converting “r+f” to “n,” while binary files do not.
C++ opens files in text mode by default.

10.2 Files Operations

Basic files operation in:
1. Creating or Opening a file
2. Reading a file
3. Writing to a file
4. Closing a file

Name	Function
fopen( )	Open a file
fclose( )	Closes a file
putc( )	Writes a character to a file
fputc( )	Same as putc()
getc( )	Reads a character from a file
fgetc( )	Same as getc()
fgets( )	Reads a string from a file
fputs( )	Writes a string to a file
fseek( )	Seeks to a specified byte in a file
ftell( )	Returns the current file position
fprintf( )	Is to a file whatprintf()is to the console
fscanf( )	Is to a file whatscanf()is to the console
feof( )	Returns true if end-of-file is reached
ferror( )	Returns true if an error has occurred
rewind( )	Resets the file position indicator to the beginning of the file.
remove( )	Erases a file.
fflush( )	Flushes a file.

Resets the file position indicator to the beginning of the file. Erases a file.
Flushes a file.

List.1 Commonly Used File-System Functions

In this section we will see the various steps and operations that can (or must) be performed to use files in C++:

Syntax for file operations

Creating or opening a file

For Text files syntax
ofstream out (“myfile.txt”);
or
ofstream out;
out.open(“myfile.txt”);
Binary Files can be used
ofstream out (“myfile.txt”,ios::binary);
or
ofstream out;
out.open(“myfile.txt”, ios::binary);
For Appending text at the end of the existing file
For Text files syntax
ofstream out(“myfile.txt”,ios::app); or
ofstream out;
out.open(“myfile.txt”, ios::app);

Binary Files can be used like this
ofstream out (“myfile.txt”,ios::app|ios::binary);
or
ofstream out;
out.open(“myfile.txt”, ios::app | ios::binary);

For reading data
For Text files syntax
ifstream in (“myfile.txt”);
or
ifstream in ;
in.open(“myfile.txt”);
For Binary files syntax
ifstream in (“myfile.txt”, ios::binary);
or
ifstream in ;
in.open(“myfile.txt”, ios::binary);

Closing Files
ofstream object “out”
out.close();

Ifstream object “in”
in.close();

Functions that can be used to perform special tasks

Operation	function	Description
Checking the end of the file.	EOF()	Used to check eof during the reading of the file
Check if an operation fails	bad()	Returns true if a reading or writing operation fails
Check if an operation fails	Fail()	Returns true in the same cases as bad (), but also in the case hat a format error happens.
Checking for the opened file.	is_open ();	Checks if the file is opened or not,returns true if the file is opened else false
Several bytes already read	count()	Returns count of the bytes read from the file
Ignoring characters during file read	ignore()	Ignores n bytes from the file (get pointer is positioned after n character)
Checking the next character	peek()	Checks the next available character, will not increase the get pointer to the next character.
Random access (only for binary files).	seekg() seekp() tellg() tellp()	In the case of binary files, random access is performed using these functions. They either give or set the position of getting and put pointers on the particular location

Example: Creating/Opening a File

```
#include<iostream>
#include<conio>
#include <fstream>
using namespace std;
int main()
{
fstream st;                              // Creating object of fstream class
st.open(“E:\samplefile.txt”,ios::out); // Creating new file
if(!st)// Checking whether file exist
{
cout<<“File creation failed”;
}
```

```
else
{
cout<<"New file created";
st.close();   //Closing file
}
getch();
return 0;
}
```

Example:Writing to a File

```
#include <iostream>
 #include<conio>
#include <fstream>
using namespace std;
 int main()
{
fstream st; //Creating object of fstream class

st.open("E:\samplefile.txt",ios::out); // Creating new file

 if(!st) //Checking whether file exist

{
cout<<"File creation failed";
}
else
{
cout<<"New file created";
        st<<"Hello"; //Writing to file
st.close();//Closing file
}
getch();
return 0;
}
```

Example: Reading from a File

```
#include <iostream>
 #include<conio>
#include <fstream>
using namespace std;
 int main()
{
fstream st; //Creating object of fstream class

st.open("E:\samplefile.txt",ios::out); // Creating new file
```

```
if(!st) //Checking whether file exist
{
cout<<"No such file";
}
else
{
char ch;
while (!st.eof())
{
st >>ch;  //Reading from file
cout << ch; //Message Read from file
}
st.close();  //Closing file
}
getch();
return 0;
}
```

Example: Close a File

```
#include <iostream> #include<conio>
#include <fstream>
using namespace std;
 int main()
{
fstream st;          //Creating          object          of          fstream          class
st.open("E:\samplefle.txt",ios::out);        // Step 2: Creating new file
     st.close();                              //Closing file
    getch();
    return 0;
}
```

10.3 Stream classes

A stream is a concept used to denote a continuous flow of data. A stream in C++ is represented by an entity with a certain class. We've only used the cin and cout stream objects so far. Various streams are used to represent various types of data flow. The ofstream class, for example, describes data flow to output disc files.

The ios class is the mother of all stream classes, providing the bulk of the functionality you'll need to interact with C++ streams. The formatting flags, error-status flags, and file operation mode are the three most critical functions. Next, we'll focus at formatting and error-status flags. We'll save the file operations mode for when we address disc files later.

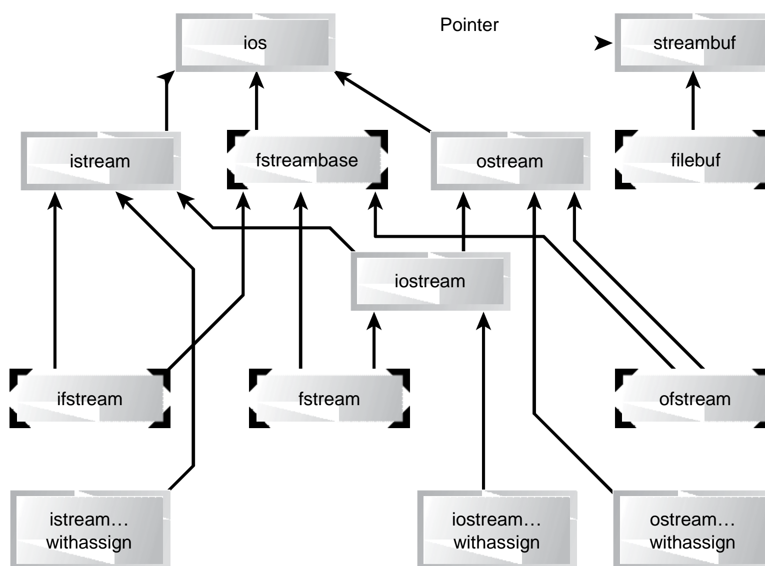
Advantages of Streams

1. Simplicity.
2. To work with classes that you build, you can overload existing operators and functions, such as the insertion (>>) operators.
3. The operators input (>>) and output (<<) are both typesafe. These are more user-friendly than scanf() and printf().

File IO using Stream CLasses

You can do I/O to and from disc files in Modern C++ in a similar way to the standard console I/O streams cin and cout. The global object cin belongs to the class istream (input stream), while the global object cout belongs to the class ostream (output stream). There are two types of file streams: input and output. Ifstream (input file stream) is a descendant of istream, while ofstream (output file stream) is a descendant of ostream. As a consequence, ifstream and ofstream objects have access to all of the member functions and operators that are available for istream and ostream objects.
Working with files usually necessitates the use of the following data communication techniques:
Transfer of information between consoles
Transfer of information between the programme and the disc file.
So far, we've looked at the iostream standard library, which includes the cin and cout methods for reading and writing to standard input and output, respectively. In this chapter, you'll learn how C++ programmes handle files and what functions and syntax are used to do so.
Lists the functions from the istream class that you'll need the most.

The Stream Class Hierarchy



The input and output streams are represented by the istream and ostream classes, which are derived from ios. Get(), getline(), read(), and the overloaded extraction (>>) operators are found in the istream class, while put() and write(), as well as the overloaded insertion () operators, are found in the ostream class.

Multiple inheritance gives the iostream class access to both istream and ostream. Tools that can be opened for both input and output at the same time, such as disc files, can be used with classes derived from it. Istream withassign, ostream withassign, and iostream withassign are three classes that are inherited from istream, ostream, and iostream.

Detail of file stream classes

Class	Contents
filebuf	Its function is to read and write the file buffers. Openprot is a constant that is used in the open() method of file stream classes. Close() and open() are also members.
fstreambase	Operations that are common to file streams are supported. This class is the base for the fstream, ifstream, and ofstream classes. The open() and close() functions are used.
ifstream	Input operations are given. There's an open() function with the default input mode. The functions get(), getline(), read(), seekg(), and tellg() are all inherited from istream.
ofstream	Output operations are given. There's an open() function with the default output mode. The put(), seekp(), tellp(), and write() functions are all inherited from ostream.
fstream	Simultaneous input and output processes are supported.Opens with the default input mode. Iostream inherits all of the functions from the istream and ostream classes.

The ios Class

The ios class is the mother of all stream classes, providing the majority of the features you'll need to work with C++ streams. The formatting flags, error-status flags, and file operation mode are the three most critical features. Next, we'll look at formatting and error-status flags. We'll save the file operations mode for when we address disc files later.

Flag	Meaning
Skipws	Skip (ignore) whitespace on input.
left	Left-adjust output [12.34].
right	Right-adjust output [12.34].
internal	Use padding between sign or base indicator and number
[+ 12.34]. dec	Convert to decimal,
oct	Convert to octal.
hex	Convert to hexadecimal.

The formatting flags can be set in a variety of ways, and different ones can be set in different ways. They must typically be preceded by the name ios and the scope-resolution operator since they are members of the ios class.

The setf() and unsetf() ios member functions can be used to set all of the flags. Consider the following scenario:
cout.setf(ios::left); / output text is left justified cout >> This text is justified on the left;

cout.unsetf(ios::left); / revert to the previous state (right justified)

Stream Error States

- eofbit
  - Set for an input stream after end-of-file encountered
  - cin.eof()returns trueif end-of-file has been encountered on cin
- failbit
  - Set for a stream when a format error occurs
  - cin.fail() - returns true if a stream operation has failed
  - Normally possible to recover from these errors
- badbit
  - Set when an error occurs that results in data loss
  - cin.bad() returns true if stream operation failed
  - normally nonrecoverable
- goodbit
  - Set for a stream if neither eofbit, failbit or badbit are set
  - cin.good() returns true if the bad, fail and eof functions would all return false.
  - I/O operations should only be performed on ‾ good streams
- rdstate
  - Returns the state of the stream
  - Stream can be tested with a switch statement that examines all of the state bits
  - Easier to use eof, bad, fail, and good to determine state

- clear
  - Used to restore a stream‘s state to ¯good
  - cin.clear() clears cin and sets goodbit for the stream
  - cin.clear(ios::failbit) actuallysets the failbit
- Might do this when encountering a problem with a user-defined type
- Other operators
  - operator!
- Returns true if badbit or failbit set
  - operator void*
- Returns false if badbit or failbit set
  - Useful for file processing

The istream Class

The istream class, which is derived from ios, performs input-specific activities, or extraction. It‘s easy to confuse extraction and the related output activity, insertion. istream Functions

Function	Purpose
>>	Formatted extraction for all basic (and overloaded) types.
get(ch);	Extract one character into ch.
get(str)	Extract characters into array str, until \n‘. get(str, MAX) Extract up to MAX characters into array.
get(str, DELIM)	Extract characters into array str until specified delimiter (typically\n‘). Leave delimiting char in stream.
get(str, MAX, DELIM)	Extract characters into array str until MAX characters or the DELIM character. Leave delimiting char in stream.
getline(str, MAX, DELIM)	Extract characters into array str, until MAX characters or the DELIM character. Extract delimiting character.
putback(ch)	Insert last character read back into input stream.
ignore(MAX,	Extract and discard up to MAX characters until (and
DELIM)	including) the specified delimiter (typically\n‘).

The ostream Class

The ostream class handles output or insertion activities.

ostream Functions

Function	Purpose
<<	Formatted insertion for all basic (and overloaded) types.
put(ch)	Insert character ch into stream.
flush()	Flush buffer contents and insert newline.
write(str, SIZE)	Insert SIZE characters from array str into file.
pos = tellp()	Return position of file pointer, in bytes.

Formating file I/O

Writing Data

This program demonstrates basic file operations

```
 #include <iostream>
#include <fstream>
using namespace std;
int main ()
{
ofstream f1;
f1.open ("abc.txt");
f1 << "Writing this to a file.\n";
f1.close();
return 0;
}
```

Reading data

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

int main()
{
char ch ;
int j ;
double d;

string str1 ;

string str2;
ifstream infile("abc.txt"); //create abc.txt
from it infile >> ch >> j >> d >> str1 >> str2;
```

```
cout<<ch<<endl;  //display data
<< j<<endl;
<< d<<endl;
<< str1<<endl;
<< str2<<endl;

cout << “File written\n”; return 0;

}
```

10.4 File Pointer and their manipulation

A file pointer is a pointer to a structure of type FILE, the FILE pointer allows us to read the content of a file when we open the file in read-only mode. It automatically points at the beginning of the file, allowing us to read the file from the beginning. This pointer defines various things about the file, including its name, status, and the current position of the file.

```
FILE *fp;
```

Opening a File

The fopen() function opens a stream for use and links a file with that stream. Then it returns the file pointer associated with that file. Most often, the file is a disk file. The fopen() function has this prototype:

```
FILE *fopen(const char *filename, const char *mode);
```

Where filename is a pointer to a string of characters that make up a valid filename and include a path specification. The string pointed to by mode determines how the file will be opened.

Closing a File

The fclose() function closes a stream that was opened by a call to fopen() function. This function writes any data remaining in the disk buffer to the file and does a formal operating-system-level close on the file. Failure to close a stream invites all kinds of trouble, including lost data, destroyed files, and possible intermittent errors in your program. closing of a file also frees the file control block associated with the stream, making it available for reuse. There is an operating-system limit to the number of open files you may have at any one time, so you may have to close one file before opening another.

The fclose() function has this prototype:

```
int fclose(FILE *fp);
```

Random file access

Random file access enables us to read & write any data in our disk file, in random access we can quickly search for data, modify data & delete data. data fseek( ) and Random-Access I/O

Random-access read and write operations using an I/O system with the help of fseek(), which sets the file position indicator. Its prototype is shown here:
int fseek(FILE *fp, longnumbytes, intorigin);

Here, fp is a file pointer returned by a call to fopen().numbytes is the number of bytes from the origin that will become the new current position, and origin is one of the following macros:

What is a Manipulator?

Manipulators are operators used in C++ for formatting output. The data is manipulated by the programmer s choice of display.

There are numerous manipulators available in C++. Some of the more commonly used manipulators are provided here below:

endl Manipulator:

This manipulator has the same functionality as the \n newline character.
For example:
cout << Exforsys << endl; cout << ¯Training ; produces the output:

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 <iomanip.h> .

```
#include <iostream.h> #include <iomanip.h> void main( )
{
int x1=12345,x2= 23456, x3=7892;
cout << setw(8) << Exforsys << 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 output of the above example is: setw(8) setw(20)
Exforsys 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 cha racter specified in the setfill argument of the manipulator is used for filling the fields. #include <iostream.h>
#include <iomanip.h> void main( )
{
cout << setw(10) << setfill($) << 50 << 33 << endl;
}
The output of the above program is
$$$$$$$$5033
This is because the setw sets 10 width for 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.

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:
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.
#include <iostream.h> #include <iomanip.h> void main( )
{
float x = 0.1;
cout << fixed << setprecision(3) << x << endl; cout << sceintific << x << endl;
}

The above gives ouput as:
0.100
1.000000e-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.
A1234567 7892

peek(ch) Read one character, leave it in stream.
count = gcount() Return number of characters read by a (immediately preceding) call to get(), getline(), or read().
read(str, MAX) For files—extract up to MAX characters into str, until EOF. seekg() Set distance (in bytes) of file pointer from start of file.
seekg(pos, seek_dir) Set distance (in bytes) of file pointer from specified place in file. seek_dir can be ios::beg, ios::cur, ios::end.
pos = tellg(pos) Return position (in bytes) of file pointer from start of file.

10.5 Summary

- In this chapter, we looked at the hierarchy of stream classes and how to manage different types of I/O errors. Then we looked at a few different ways to perform file I/O.
- In C++, files are connected to objects of different classes, most notably ofstream for output, ifstream for input, and fstream for both input and output.
- I/O operations are performed using member functions of these or base groups. For output, operators and functions like, put(), and write() are used, while for data, >>, get(), and read() are used.
- Since the read() and write() functions operate in binary mode, whole objects can be saved to disc regardless of their data type.
- A check for error conditions should be made after each file operation. The file object itself takes on a value of 0 if an error occurred. Also, several member functions can be used to determine specific kinds of errors.
- The extraction operator >> and the insertion operator << can be overloaded so that they work with programmer-defined data types. Memory can be considered a stream, and data sent to it as if it were a file.

10.6 References

1. The Complete Reference-C++,4th Edition. Herbert Schildt,Tata McGraw- Hill
2. The C++ Programming Language, 4th Edition, Bjarne Stroustrup, AddisonWesly
3. Absolute C++,4th Edition, Walter Savitch,Pearson Education

10.7 Bibliography

1. https://www.tutorialspoint.com
2. https://www.geeksforgeeks.org
3. https://www.javatpoint.com
4. https://guru99.com
5. www.slideshare.net

10.8 Questions

Q.1. What are streams? Why we use them?

Q.2. Briefly describe input and output streams?

Q.3. Briefly describe the class hierarchy provided by c++ for streams.

Q.4. What is the difference between a text file and a binary file?

Q.5. Write a program to read a file through c++ program

Q.6. Write a program to write on to a text file through c++ program

Q.7. How many ways can a file to be opened in c++ program?

Q.8. Describe the following manipulators setw(), setprecision(), setfill(),Setiosflags(),resetiosflags().

Q.9. How will you create manipulators?

Q.10. Write the syntax and use of getline () and write () functions.

Q.11. What are the differences between manipulators and ios functions?

*****