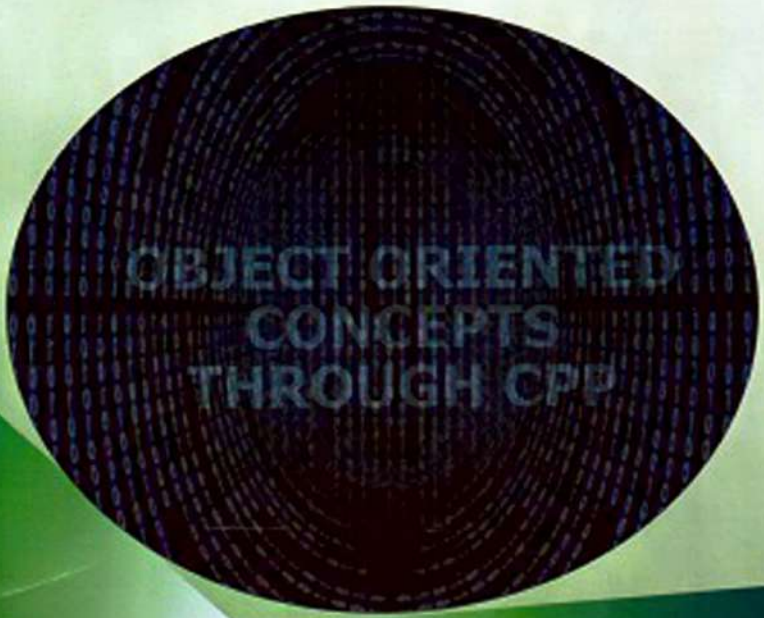
NEW SYLLABUS
CBCS PATTERN

S.Y. B.B.A. (C.A.)
SEMESTER - IV

OBJECT ORIENTED CONCEPTS THROUGH CPP

Dr. Ms. MANISHA BHARAMBE



NIRALI PRAKASHAN
ADVANCEMENT OF KNOWLEDGE

Syllabus ...

1. Introduction to C++ **[Lectures 2]**

- 1.1 Basic Concepts, Features, Advantages and Applications of OOP
- 1.2 Introduction, Applications and Features of C++
- 1.3 Input and Output Operator in C++
- 1.4 Simple C++ Program

2. Beginning with C++ **[Lectures 6]**

- 2.1 Data type and Keywords
- 2.2 Declaration of Variables, Dynamic Initialization of Variables, Reference Variable
- 2.3 Operators:
  - 2.3.1 Scope Resolution Operator
  - 2.3.2 Memory Management Operators
- 2.4 Manipulators
- 2.5 Functions:
  - 2.5.1 Function Prototyping, Call by Reference and Return by Reference
  - 2.5.2 Inline Functions
- 2.6 Default Arguments

3. Classes and Objects **[Lectures 8]**

- 3.1 Structure and Class, Class, Object
- 3.2 Access Specifiers, Defining Data Member
- 3.3 Defining Member Functions Inside and Outside Class Definition
- 3.4 Simple C++ Program using Class
- 3.5 Memory Allocation for Objects
- 3.6 Static Data Members and Static Member Functions
- 3.7 Array of Objects, Objects as a Function Argument
- 3.8 Friend Function and Friend Class
- 3.9 Function Returning Objects

4. Constructors and Destructors **[Lectures 6]**

- 4.1 Constructors
- 4.2 Types of Constructor: Default, Parameterized, Copy
- 4.3 Multiple Constructors in a Class
- 4.4 Constructors with Default Argument
- 4.5 Dynamic Initialization of Constructor
- 4.6 Dynamic Constructor
- 4.7 Destructor

5. Inheritance **[Lectures 6]**

- 5.1 Introduction
- 5.2 Defining Base Class and Derived Class
- 5.3 Types of Inheritance
- 5.4 Virtual Base Class
- 5.5 Abstract Class
- 5.6 Constructors in Derived Class

6. Polymorphism **[Lectures 8]**

- 6.1 Compile Time Polymorphism
  - 6.1.1 Introduction, Rules for Overloading Operators
  - 6.1.2 Function Overloading
  - 6.1.3 Operator Overloading Unary and Binary
  - 6.1.4 Operator Overloading using Friend Function
  - 6.1.5 Overloading Insertion and Extraction Operators
  - 6.1.6 String Manipulation using Operator Overloading
- 6.2 Runtime Polymorphism
  - 6.2.1 This Pointer, Pointers to Objects, Pointer to Derived Classes
  - 6.2.2 Virtual Functions and Pure Virtual Functions

7. Managing console I/O operations **[Lectures 3]**

- 7.1 C++ Streams and C++ Stream Classes
- 7.2 Unformatted I/O Operations
- 7.3 Formatted Console I/O Operations
- 7.4 Output Formatting Using Manipulators
- 7.5 User Defined Manipulators

8. Working with Files **[Lectures 6]**

- 8.1 Stream Classes for File Operations
- 8.2 File Operations - Opening, Closing and Updating
- 8.3 File Updating with Random Access
- 8.4 Error Handling during File Operations
- 8.5 Command Line Arguments

9. Templates [Lectures 3]

- 9.1 Introduction
- 9.2 Class Template and Class Template with Multiple Parameters
- 9.3 Function Template and Function Template with Multiple Parameter
- 9.4 Exception Handling Introduction

■■■

Contents ...

■■■

1... Introduction to C++

Objectives...

- To study basic concepts, features, advantages and applications of OOP.
- To learn applications and features of C++.
- To understand Input and Output operator in C++.
- To learn Simple C++ program.

1.1 INTRODUCTION

- During the late 1970's and early 1980's 'C' had become very popular language. C was a structural programming language. However, for large and complex program, the structural approach failed to show the desired results in terms of bug-free, easy-to-maintain and reusable programs. To deal with this a new way to program was invented "Object Oriented Programming".
- Object Oriented Programming (OOP) is playing an increasingly significant role in the analysis, design and implementation of software systems. OOP languages provide the programmer the ability to create class hierarchies, instantiate objects and send messages between objects to process themselves.
- Object oriented means to organize software as a collection of discrete, real-world objects that incorporate both data structure and behavior.
- Object Oriented Modeling and Design (OOMD) is a new way of thinking about problems using models organized around real-world concepts. The fundamental construct is the object, which combines both data structure and behavior in a single entity.
- Object oriented models are useful for understanding problems, communicating with application experts, modeling enterprises, preparing documentation and designing programs and databases.
- Object oriented development is a conceptual process independent of a programming language until the final stage.
- Object Oriented Programming is a new way of organizing code and data that promises increased control over the complexity of software development process.

- The prime purpose of C++ programming was to add object orientation to the C programming language, which is in itself one of the most powerful programming languages.

1.1.1 Need of OOP

- The use of structured programming made more complex programs, less complex and easier to understand. But even with structured programming, when a program reaches a certain size, its complexity exceeds that which a programmer can manage.
- To deal with this approach, a new way to program, was invented **"Object Oriented Programming"**. This method of programming reduces the program complexity by incorporate rating features like inheritance, encapsulation and polymorphism.
- OOP is a way of programming help the programmer to comprehend and manage large and complex programs.
- Object Oriented Programming models real-world objects with software counterparts. It takes advantage of class relationships where objects of certain class have the same characteristics.
- Object Oriented Programming (OOP) allows us to decompose a problem into a number of entities called objects.
- OOP takes advantages of inheritance and multiple inheritance relationship where newly created classes of objects are derived by absorbing characteristics of existing classes and adding unique characteristics of their own.
- OOP gives us a more natural way to view the programming process, namely by modeling real-world objects. C++ was the first object oriented programming language. It was invented by **"Bjarne Stroustrup"**. This language was initially called as **"C with classes"**.

1.1.2 Object Oriented Vs Procedure Oriented Programming [W-18]

Table 1.1: Difference between Object Oriented Programming and Procedure Oriented Programming

Object Oriented Programming (OOP)	Procedure Oriented Programming (POP)
1. Object oriented programming language is object oriented.	1. Procedural programming languages tend to be action oriented.
2. In OOP, the unit of programming is the class.	2. In procedural programming, unit of programming is the function.
3. OOP programmers concentrate on creating their own user-defined types called classes and components. Each class contains data as well as the set of functions that manipulate that data. The data components of a class are called data members. The function components of a class are called member functions. The instance of a class is called an object.	3. In procedural oriented programming programmers concentrate on writing functions. Groups of actions that perform some common task are formed into functions and functions are grouped to form programs.

contd. ...

4. OOP trace on data.	4. POP trace on procedures.
5. Follow bottom-up approach in program design.	5. Follow top-down approach in program design.
6. Basic building block of OOP is object.	6. Basic building block of POP is function.
7. OOP has access specifiers named Public, Private, and Protected.	7. POP does not have any access specifiers.
8. Examples of OOP are C++, JAVA, VB.NET, C#.NET.	8. Examples of POP are C, VB, FORTRAN, Pascal.
9. In OOP, importance is given to the data rather than procedures or functions because it works as a real world.	9. In POP, importance is not given to data but to functions as well as sequence of actions to be done.
10. OOP provide Data Hiding so it provides more security.	10. POP does not have any proper way for hiding data so it is less secure.
11. In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.	11. In POP, Overloading is not possible.
12. In OOP, objects can move and communicate with each other through member functions.	12. In POP, data can move freely from function to function in the system.

1.2 OBJECT ORIENTED CONCEPTS

1.2.1 Classes

- A class is a group of objects with similar properties (attributes), common behaviour (operations), common relationship to other objects and common semantics.
- Classes are user defined datatypes and behave like the built-in types of programming languages.
- The entire set of data and code of an object can be made user defined data type with the help of class.
- Once, a class has been defined we can create any number of objects belonging to the same class. Person, company, process, window are some examples of classes.
- Fig. 1.1 shows class Person, with two objects i.e. Ram and Shyam.

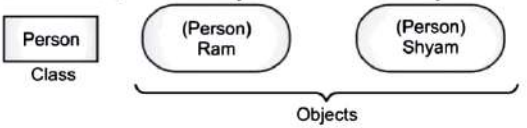


Fig. 1.1: Classes and Objects

1.2.2 Objects

- Objects are discrete, distinguishable entities.
- Objects are basic run-time entities.

- A paragraph in a document, a window on a workstation, the white queen in a chess game is some examples of Objects.
- Objects can be concrete such as a file in a file system or conceptual such as scheduling policy in a multiprocessing operating system.
- Each object has its own inherent identity.

Objects serve two purposes:

1. Objects promote understanding of the real-world and provide a practical basis for computer implementation.
2. Decomposition of a problem into objects depends on judgment and the nature of the problem. Program objects should be chosen such that they match closely with the real-world objects.

- Fig. 1.2 shows different styles of object representation.

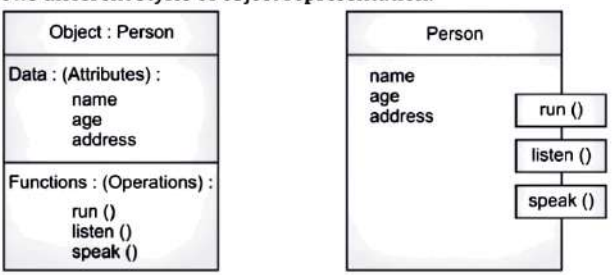


Fig. 1.2: Different styles to represent object

- In a program, objects interact by sending messages to one another.
- Each object contains data and code to manipulate the data.
- Objects can interact without having to know details of each other's data or code. It is sufficient to know the type of message accepted and the type of response returned by the objects.

1.2.3 Data Abstraction [S - 19]

- Abstraction is the selective examination of certain aspects of a problem.
- The goal of abstraction is to isolate those aspects that are important for some purpose and suppress those aspects that are unimportant.
- In other words, we can say that abstraction is the act of representing essential features without including the background details or explanations.
- Data abstraction is the process of defining a data type, often called Abstract Data Type (ADT), together with the principle of data hiding.
- The definition of an ADT involves specifying the internal representation of the ADTs data as well as the functions to be used by others to manipulate the ADT.
- In OOP, classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost and functions to operate these attributes. Classes are also known as ADTs.

1.2.4 Encapsulation [W-18]

- The wrapping up of data and functions into a single unit i.e. class is called as Encapsulation.
- In other words, "the binding of data and function into a single unit (class) is called as encapsulation".
- The data is not accessible to the outside world and only those functions which are wrapped in the class can access it.
- These functions which are wrapped in the class provide the interface between the objects data and the program.
- Encapsulation is a mechanism that keeps the data and code safe from external interference and misuse. This insulation of the data from direct access by the program is called data hiding which is also known as data encapsulation.

1.2.5 Inheritance

- In inheritance process we can create new classes known as derived or child or sub-class from the existing class known as base or parent or super class.
- Inheritance is a process by which object of one class (derived class) can acquire the properties of objects of another class (base class).
- The concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is made possible by deriving a new class from the existing one. The new class will have the combined features of both the classes.
- Fig. 1.3 shows class of graphic geometric figures. Move, Select, Rotate and Display are operations inherited by all subclasses. Scale applies to one and two-dimensional figures. Fill applies only two-dimensional figures.

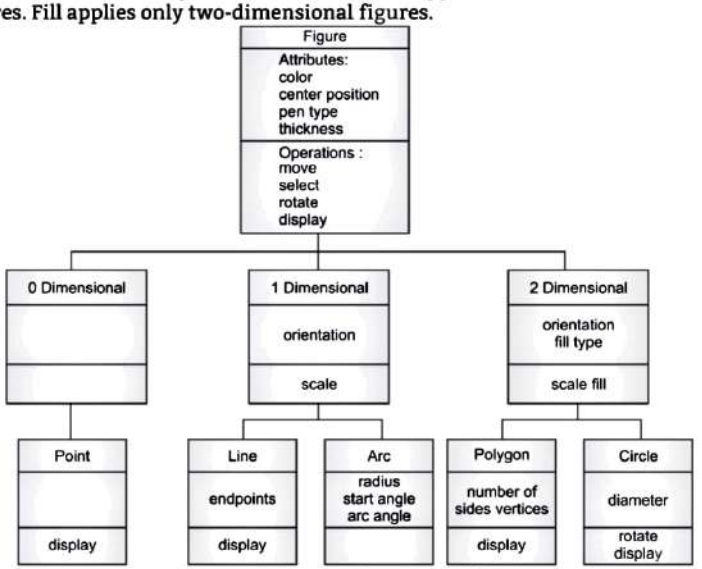


Fig. 1.3: Inheritance for graphic figures

1.2.6 Polymorphism

- In Polymorphism, "poly" means many and "orphism" means form which forms many forms of one things.
- In other words, "Polymorphism means one thing different or many forms" i.e. the ability to take more than one form.
- Polymorphism plays an important role in allowing objects having different internal structures to share the same external interfaces.
- In OOP, polymorphism refers to the fact that a single operation can have different behaviour in different objects. In other words, different objects react differently to the same message.
- For example, consider the operation of addition. For two numbers, the addition should generate the sum. The operation of addition is expressed by a single operator +. You can use the expression x + y to denote the sum of x and y for many different types of x and y integers, floating point numbers and complex numbers and even the concatenation of two strings.
- Similarly, suppose a number of geometrical shapes all respond to the message draw.
- Each object reacts to this message by displaying its shape on a display screen. Obviously, the actual mechanism for displaying the object differs from one shape to another, but all shapes perform this task in response to the same message.

1.2.7 Dynamic Binding

- Binding refers to the linking of a procedure call to the code to be executed in response to the call. This link can be physical or conceptual connection between object instances.
- Link is an instance of an association.
- Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run-time.
- Dynamic binding is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.
- Dynamic binding also known as Late Binding.

1.2.8 Message Passing

- The act of communicating with an object to get something done is called as Message Passing.
- OOP consists of set of objects that communicate with each other.
- The process of programming in an object oriented language, therefore involves the following basic steps:
  1. Creating classes that define objects and their behaviour,
  2. Creating objects from class definitions,
  3. Establishing communication among objects.

- Objects communicate with one another by sending and receiving information.
- A message for an object is a request for execution of a procedure and therefore will invoke a function (procedure) in the receiving object that generates the desired result.
- Message passing involves specifying the name of the object, the name of the function (message) and the information to be sent.
- Example:

```
employee.salary (name);
```

Object Message Information

- Objects have a life cycle. They can be created and destroyed. Communication with an object is feasible as along as it is alive. The way of representing message passing is shown in Fig. 1.4.

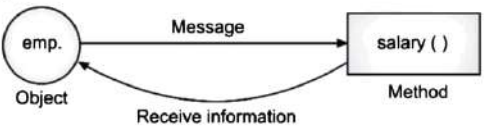


Fig. 1.4: Message Passing

1.3 FEATURES OF OOP [S - 18]

- OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it and protects it from accidental modification from outside functions. OOP allows decomposition, (breaking into small modules) of a problem into number of entities called objects and then builds data and functions around these objects.
- The organization of data and functions in OOP is shown in Fig. 1.5. The data of an object can be accessed only by the functions associated with that object. However, functions of one object can access the functions of other objects.

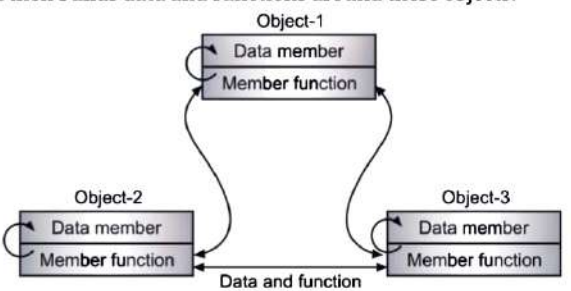


Fig. 1.5: Data and Function Organization in OOP

- The organization of data and function is payroll system is shown in Fig. 1.6.
- In Fig. 1.6, data and function organization is shown with respect to payroll system. We can consider three objects like employee, pay slip and other details.

- All of them have some data members and member functions which act on that data. The member functions of one object can communicate with the member function of another object.

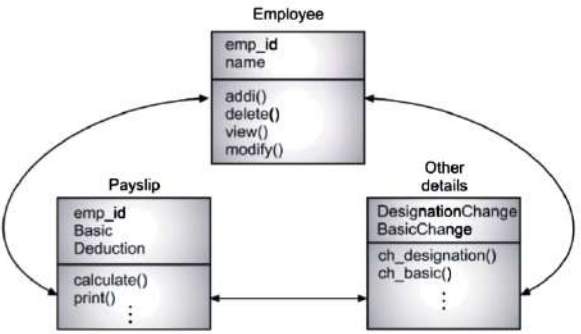


Fig. 1.6: Organization of Data and Member Function in Payroll

- Fig. 1.7 shows the important features of OOPs. It also shows various features offered by C++ programming language. From the Fig. 1.7 we can recognize that the feature persistence is not offered by C++ programming language.

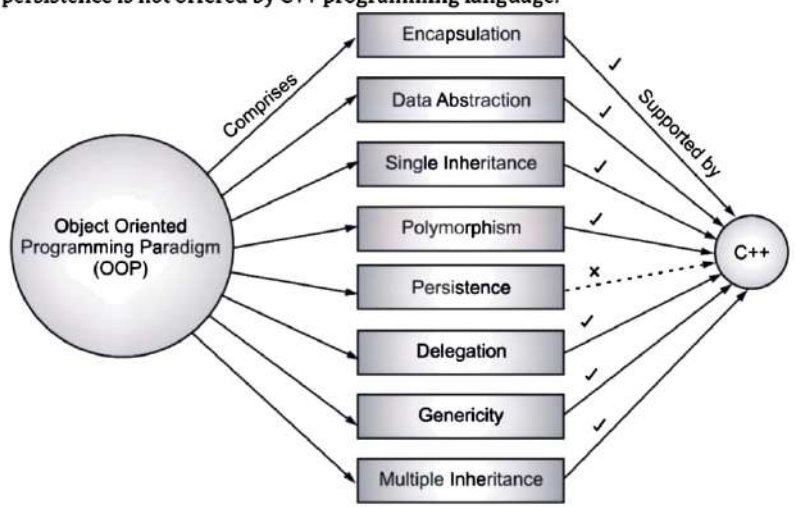


Fig. 1.7: Features of OOP's

1. Data Encapsulation:

- The wrapping up of data and functions into a single unit i.e. class is called as encapsulation. Data encapsulation is also known as data hiding.
- The data is not accessible to the outside world and only those functions which are wrapped in the class can access it. These functions provide the interface between the objects data and the program. They are bound together.

- They are safe from external interference and misuse. This represents encapsulation (or seal). This feature is not available in any conventional procedure-oriented programming language.

  For example: The atmosphere encapsulates the earth, skin encapsulates the internal part of a body.

2. Data Abstraction:

- In object oriented programming, each object will have external interfaces through which it can be used. There is no need to look into its inner details.
- So the user needs to know the external interfaces only, to make use of an object. The internal details of the objects are hidden which makes them abstract. This technique or feature of hiding internal details in an object is known as Data Abstraction.
- Abstraction is the selective examination of certain aspects of a problem. The goal of abstraction is to isolate those aspects that are important for some purpose and suppress those aspects that are unimportant.
- Many different abstractions of the same thing are possible, depending on the purpose for which they are made. In other words, we can say that abstraction as the act of representing essential features without including the background details or explanation. Data abstraction is the process of defining a data type, often called Abstract Data Type (ADT), together with the principle of data hiding.
- The definition of an ADT involves specifying the internal representation of the ADTs data as well as the functions to be used by others to manipulate the ADT.
- In OOP, classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost and functions to operate these attributes. Classes are also known as ADTs.

3. Inheritance:

- The notation of defining a new object in terms of an old one is an integral part of OOP. The term inheritance is used for this concept. Inheritance means one class of objects inherits the data and behaviour from another class.
- Inheritance imposes a hierarchical relationship among classes in which a child class inherits from its parents. The parent class is known as the base class and the child is the derived class. The base class is also known as *Superclass* and the derived class is known as *Subclass*.
- Attributes and operations common to a group of subclasses are attached to the superclass and shared by each subclass. Each subclass is said to inherit the features of its superclass. A child inherits the property of his father. He can acquire new properties or modify the inherited one.
- Same way, a new class can derive its properties from another existing class. The derived class inherits all the properties of the base (old) class. This allows the extension and reuse of existing code. This also enables the creation of hierarchy of classes which simulate the class and subclass concept of real world.

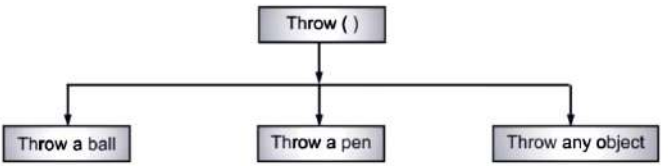
4. Polymorphism:

- It is very useful concept in OOP's. In simple terms it means one name, many duties. It provides common interfaces to carry out similar tasks.

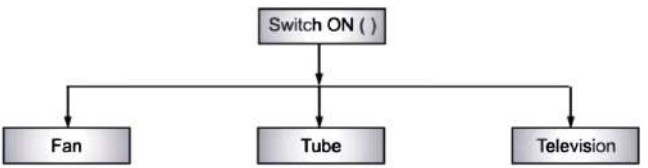
- In other words, we can say that a common interface is created for accessing related objects. In C++, it is achieved by function overloading operator overloading and dynamic binding.

  For example:

  (i)



  (ii)



- Basically, polymorphism means the quality of having more than one form. In OOP, polymorphism refers to the fact that a single operation can have different behavior in different objects.
- In other words, different objects react differently to the same message. For example, consider the operation of addition. For two numbers, the addition should generate the sum.
- The operation of addition is expressed by a single operator +. You can use the expression x + y to denote the sum of x and y for many different types of x and y integers, floating point numbers and complex numbers and even the concatenation of two strings.
- Similarly, suppose a number of geometrical shapes all respond to the message draw. Each object reacts to this message by displaying its shape on a display screen. Obviously, the actual mechanism for displaying the object differs from one shape to another, but all shapes perform this task in response to the same message.

5. Message Passing:

- It is the process of invoking an operation on an object. In response to a message, the corresponding method (function) is executed in the object.
- OOP consists of objects and classes. It consists of set of objects that communicate with each other. It is very necessary to understand properly the classes, objects and also the message passing i.e. communication between objects.
- Objects communicate with one another by sending and receiving information. A message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired result.
- Message passing involves specifying the name of the object, the name of the function (message) and the information to be sent.

6. **Extensibility:**
- It is a feature, which allows the extension of the functionality of the existing software components. In C++, this is achieved through abstract classes and inheritance.

7. **Persistence:**
- The phenomenon where the object (data) outlives the program execution time and exists between executions of a program is known as Persistence.
- In C++, this is not supported. However, the user can build it explicitly using file streams in a program.

8. **Delegation:**
- It is an alternative to class inheritance. It is a way of making object composition as powerful as inheritance. In delegation, two objects are involved in handling a request.
- In C++, this approach takes a view that an object can be a collection of many objects and the relationship is called has-a relationship or containership.

9. **Genericity:**
- It is a technique for defining software components that have more than one interpretation depending on the data type of parameters.
- In C++, genericity is realised through function templates and class templates.

1.4 ADVANTAGES AND APPLICATIONS OF OOP

1.4.1 Advantages of OOP

1. Data abstraction concept consists of focusing on the essential, inherent aspects of an entity and ignoring its accidental properties. The use of abstraction preserves the freedom to make decisions as long as possible.
2. By the use of property inheritance, we can eliminate redundant code and extend the use of existing class.
3. The property of data encapsulation helps to programmer to build secure programs that cannot be invaded by code in other parts of the program.
4. We can have multiple instances of an object to co-exist without any interference.
5. We can map objects in the problem domain to the object in the program.
6. We can modularise the program by dividing it into classes and objects.
7. The data centered design approach enables to capture more details of a model in implementable form.
8. Object oriented programming supports reusability. Reusable software reduces design, coding and testing cost by amortizing effort over several design. Reducing the amount of code also simplifies understanding.
9. In OOP, complex softwares can be easily managed.
10. OOP saves development time and gives higher productivity.
11. In OOP, small projects (programs) can be easily upgraded to larger one.
12. Message passing in OOP provides the interface with external systems.

1.4.2 Applications of OOP

- Real business systems are often much more complex and contain many more objects with complicated attributes and methods. OOP is useful in these types of applications because it can simplify a complex problem.
- The areas for application of OOP includes:
  1. Real time systems, for example, oil exploration plant.
  2. CAM/CAD systems.
  3. Object oriented databases.
  4. Artificial Intelligence (AI) and expert systems, for example, Disease knowledge base.
  5. Hypertext, hypermedia and expertext, for example, web page designing.
  6. Decision support and office automation systems.
  7. Neural networks and Parallel programming.
  8. Simulation and Modeling.

1.5 INTRODUCTION TO C++

- C++ is a hybrid language, in which some entities are objects and some are not.
- C++ is an extension of C language, implemented not only to add object-oriented capabilities but also to readdress some of the weaknesses of C language.
- Many added features are orthogonal to OOP, such as inline expansion of subroutines, overloading of functions and function prototypes.
- C++ is a strongly typed language developed by **Bjarne Stroustrup** at AT and T's Bell Laboratories. It was originally implemented as a preprocessor that translates C++ into standard C.
- C++ contains good facilities for specifying access to attributes and operations of a class. Access may be permitted by methods of any class (public), restricted to methods of subclasses of the class (protected), or restricts to direct methods of the class (private). In addition, instant access can be given to a particular class or function using the friend declaration.
- C++ also supports overloaded operators i.e. several methods that share the same name but whose arguments vary in number of type.
- C++ supports several memory allocation strategies for objects – statically allocated by the compiler, stack based and allocated at run-time from a heap.
- The programmer must avoid mixing objects of different memory types or dangling references that may cause run-time failures.
- Each class can have several constructor and conversion functions, which initialize new objects and convert between types for assignment and argument passing.
- To conclude, C++ is a complex, malleable language characterized by a concern for the early detection of errors, various implementation choices and run-time efficiency at the expense of some design flexibility and simplicity.
- C++ systems generally consist of several parts i.e. a program development environment, the language and the C++ standard library.

1.5.1 Features of C++

- C++ supports all features of structure programming and object oriented programming, they are listed below:
  1. C++ provides overloading of functions and operators.
  2. Exception handing in C++ is done by the try, catch and throw keywords.
  3. C++ provides static methods, friends, constructors and destructors for the class object.
  4. It focuses on function template and class template for handling parameterized data types.
  5. C++ gives the simplest and easiest way to handle encapsulation and data hiding with the help of powerful keywords such as private, class, public and protected and so on.
  6. Inheritance concept in C++, one of the most powerful and important design concept is supported with single inheritance and multiple inheritances of base class and derived class.
  7. In C++, the polymorphism is done through virtual functions, virtual base classes and virtual destructors give the late binding of the compiler.

1.5.2 Applications of C++

- C++ is a flexible language, capable for handling very large programs.
- C++ is suitable for almost any programming task as well as development of databases, editors, communication systems, compiler and any complex real life application systems.
- C++ language is able to map the real world problem properly, effectively and efficiently the C part of C++ language gives the language the ability to get close to the machine level details.
- In C++ language programs are easily expandable and maintainable when a new feature needs to be implemented. It is very easy to add to the existing structure of an objects.
- C++ language allows user to create hierarchy related object, he/she can built special object oriented libraries which can be used later by many programmers.

1.6 INPUT AND OUTPUT OPERATOR IN C++

- C++ support a set of functions for performing input and output operations. The new feature of C++ is called **streams** which is used to handle I/O operations.
- C++ streams are of two types:
  (i) Input stream, and
  (ii) Output stream.
- Stream refers to the flow of data from a particular source to a specified destination.
- There are classes representing each stream **istream** is the class representing the input stream (istream) and **ostream** is the class representing the output stream.
- To achieve the console Input/Output operations, we use the objects of these stream classes.

(i) Input stream:

- The input stream is handling all input operations or reading operations with input devices like keyboard, disk etc. **"cin"** is the predefined object of the istream class. It is used for all console input operations.
- Operator **>>**, (or the right shift operator in C) is called the **extraction** operator, it gets the value from the stream object **cin** on its left and places it in the variable on its right.

Syntax:

```
cin>>variable;
```

- The use of cin is shown in Fig. 1.8.

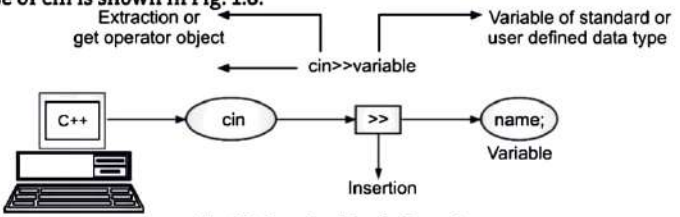


Fig. 1.8: Input with cin Operator

Examples:

1. `int a;`
   `cin>>a;`
2. `float salary;`
   `cin>>salary;`
3. `double area;`
   `cin>>area;`
4. `char name[15];`
   `cin>>name;`

- We can input more than one item using cin object. Such input operations are called cascaded input operations.
- cin will read all the items from left to right.

Syntax for more than one variable as:

```
cin>>var1>>var2>>var3..... >>varn;
```

Examples:

1. `cin>>x>>y;` `//x read first then y`
2. `cin>>name>>address;`
3. `cin>>roll>>name>>marks;`

(ii) Output stream:

- The output stream handles the write operations on output devices like screen, disk etc. **cout** is the predefined object of the **stream** class and is used for all console output operations.
- Operator **<<**, (or the left shift operator in C) is called **insertion** operator. It directs the contents of the variable on its right to the object (cout) on its left.

Syntax:

```
cout<<variable;
```

- The use of cout is shown in Fig. 1.9.

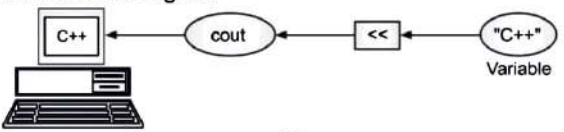


Fig. 1.9: Output with cout Operator

Examples:

1. `int a;`
   `cout<<a;`
2. `float salary;`
   `cout<<salary;`
3. `char name[15];`
   `cout<<name;`

- We can display more than one outputs called cascaded outputs.

Syntax:

```
cout<<var1<<var2<<.....<<varn;
```

Examples:

1. `cout<<"x="<<x;`
2. `cout<<roll<<"  "<<name;`
3. `cout<<roll<<"\t"<<name;`
4. `cout<<name<<endl;`

1.7 A SIMPLE C++ PROGRAM

- A computer program is a sequence of instructions that tell the computer what to do.
- A typical C++ program would contain four sections:
  1. Include files
  2. Class declaration
  3. Member function definition
  4. Main function
- Fig. 1.10 shows structure of a C++ program.

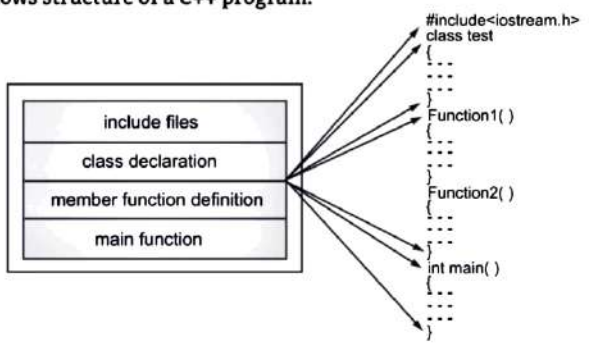


Fig. 1.10: Structure of C++ Program

- C++ program is a collection of functions.

For example:

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     cout << "Hello World!" <<  endl;
6.     return 0;
7. }
```

Output:

```
Hello World!
```

- Line 1 `#include<iostream>` is a special type of statement called a preprocessor directive and it start from #. Preprocessor directives tell the compiler to perform a special task. In this case, we are telling the compiler that we would like to use the iostream library. The iostream library contains code that tells the compiler what cout and endl do. In other words, we need to include the iostream library in order to write to the screen.
- Line 2 `using namespace std;` defines a scope for the identifiers that are used in the program. As you learned in the explanation for line 1, cout and endl live inside the iostream library. However, within iostream, they live inside a special compartment named std (short for standard). This using statement tells the compiler to look inside a compartment named std if it cannot find cout or endl defined anywhere else. In other words, this statement is also necessary so the compiler can find cout and endl, which we use on line 5.
- Line 3 `int main()` declares the main() function which is the point from where all C++ program begin their execution. Every program must have a main() function.
- Lines 4 ({ is the opening curly brace marks the start of a code block) and 7 (} is the closing curly brace, marks the end of function main()) tell the compiler which lines are part of the main function. Everything between the opening curly brace on line 4 and the closing curly brace on line 7 is considered part of the main() function.
- Line 5 `cout <<"Hello World!" << endl;` is our output statement. cout is a special object that represents the console/screen. The << symbol is an operator (much like + is an operator) called the output operator. cout understands that anything sent to it via the << operator should be printed on the screen. endl is a special symbol that moves the cursor to the next line.
- Line 6 `return 0;` is a new type of statement, called a return statement. When an executable program finishes running, it sends a value to the operating system that

indicates whether it was run successfully or not. The return value of main() is used for this purpose. This particular return statement returns the value of 0 to the operating system, which means "everything went okay!".

1. Comments:

- Comments are parts of the source code disregarded by the compiler. They simply do nothing.
- Their purpose is only to allow the programmer to insert notes or descriptions embedded within the source code.
- C++ supports two ways to insert comments:
  (i) // line comment, and
  (ii) /* */ block comment.
- The first of them, known as line comment, discards everything from where the pair of slash signs (//) is found up to the end of that same line.
- The second one, known as block comment, discards everything between the /* characters and the first appearance of the */ characters, with the possibility of including more than one line.

Program 1.1: Program using comments.

```
#include <iostream>
using namespace std;
int main()
{
   cout << "Hello World!" << endl;         //Prints Hello World!
   cout << "WelCome to C++ !" << endl;     //Prints Welcome to C++!
   return 0;
}
```

Output:

```
Hello World!
Welcome to C++ !
```

- If you include comments within the source code of your programs without using the comment characters combinations //, /* */, the compiler will take them as if they were C++ expressions, most likely causing one or several error messages when you compile it.

2. Header Files:

- Each standard library has a corresponding header file containing the function prototypes for all the functions in that library and definitions of various data types and constants needed by those functions.
- Some common C++ header files that may be included in standard C++ program, are given below:

Table 1.2: Standard Library Header Files

Standard library header files	Explanation
<assert.h>	Contains macros and information for adding diagnostics that aid program debugging.
<ctype.h>	Contains function prototypes for functions that test characters for certain properties and that can be used to convert lowercase letters to uppercase letters.
<float.h>	Contains the floating point size limits of one program.
<limits.h>	Contains integral size limits of the system.
<math.h>	Contains function prototype for math library function.
<stdio.h>	Contains function prototype for the standard Input/Output library functions and information used by them.
<stdlib.h>	Contains function prototype for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions.
<string.h>	Contains function prototype for C style string processing function.
<time.h>	Contains function prototype for manipulating time and date.
<iostream.h>	Contains function prototype for standard input and output functions.
<iomanip.h>	Contains function prototype for the stream manipulators that enable formatting of streams of data.
<fstream.h>	Contains function prototype for functions that perform input from files on disk and output to files on disk.

- The programmer can create custom header files. Programmer defined files should end in .h, which can be included by using the #include preprocessor directive.
- For example, the header file `square.h` can be included in our program by the directive, `#include "square.h"` at the top of the program.

1.8 C VS C++

- C++, as the name suggests is a superset of C. As a matter of fact, C++ can run most of C code while C cannot run C++ code.
- Below are the major differences between C and C++.
  1. **C follows the procedural programming paradigm while C++ is a multi-paradigm language (procedural as well as object oriented):** In case of C, importance is given to the steps or procedure of the program while C++ focuses on the data rather than the process. Also, it is easier to implement/edit the code in case of C++ for the same reason.
  2. **In case of C, the data is not secured while the data is secured (hidden) in C++:** This difference is due to specific OOP features like Data Hiding which are not present in C.

3. **C is a low-level language while C++ is a middle-level language:** C is regarded as a low-level language (difficult interpretation and less user friendly) while C++ has features of both low-level (concentration on what's going on in the machine hardware) and high-level languages (concentration on the program itself) and hence is regarded as a middle-level language.
4. **C uses the top-down approach while C++ uses the bottom-up approach:** In case of C, the program is formulated step by step, each step is processed into detail while in C++, the base elements are first formulated which then are linked together to give rise to larger systems.
5. **C is function-driven language C++ is object-driven language:** Functions are the building blocks of a C program while objects are building blocks of a C++ program.
6. **C++ supports function overloading while C does not:** Overloading means two functions having the same name in the same program. This can be done only in C++ with the help of Polymorphism.
7. **We can use functions inside structures in C++ but not in C:** In case of C++, functions can be used inside a structure while structures cannot contain functions in C.
8. **The namespace feature in C++ is absent in case of C:** C++ uses namespace which avoid name collisions.
9. **The standard input and output functions differ in the two languages:** C uses scanf and printf while C++ uses cin (>>) and cout (<<) as their respective input and output functions.
10. **C++ allows the use of reference variables while C does not:** Reference variables allow two variable names to point to the same memory location. We cannot use these variables in C programming.
11. **C++ supports exception handling while C does not:** C does not support it "formally" but it can always be implemented by other methods. Though you don't have the framework to throw and catch exceptions as in C++.
12. **C is a structure oriented language** where as **C++ is an object oriented language** where the coding is done by using the user defined objects.
13. C files are stored with an extension **'.c'** whereas C++ files are stored with **'.cpp'** extension.

Summary

- OOP stands for Object Oriented Programming. OOP is an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand.
- Examples of OOP are C++, Java, Eiffel, Smalltalk, VB.Net, C#.Net etc.
- Basic object-oriented concepts are objects, classes, encapsulation, inheritance, and polymorphism.
- Object is the basic unit of object-oriented programming. In OOP, a problem is considered as a collection of a number of entities called objects. Objects are instances of classes.

- In OOP a class gives the blueprint for a set of similar objects. A class is a user defined type consisting of data members and member functions.
- Data abstraction increases the power of programming language by creating user defined data types. Data abstraction also represents the needed information in the program without presenting the details.
- In OOP insulation of data from direct access by the program is called data hiding.
- Inheritance is the process of forming a new class from an existing class or base class. The base class is also known as parent class or super class. The new class that is formed is called derived class. Derived class is also known as a child class or sub class.
- Data encapsulation combines data and functions into a single unit called class. When using data encapsulation, data is not accessed directly; it is only accessible through the functions present inside the class.
- Data encapsulation enables the important concept of data hiding possible. Inheritance helps in reducing the overall code size of the program.
- In OOP polymorphism means one name, multiple forms. It allows us to have more than one function with the same name in a program.
- Persistence feature of OOP allows the object (data) outlives the program execution time and survives several executions of a program.
- Extensibility feature of OOP allows extension of the functionality of the existing software components.
- In OOP, dynamic binding means that the code associated with a given procedure is not known until the time of the call at run-time.
- Binding refers to the linking of a procedure call to the code to be executed in response to the call. When memory is allocated at run-time it is known as dynamic binding.
- In OOP, message passing involves specifying the name of the object, the name of the function (message) and the information to be sent.
- Object oriented approach has many benefits like Ease in software design, Ease in software maintenance and Reusability of software.
- Application of OOP has gained importance in almost all areas of computing including real-time business systems.

Check Your Understanding

1. The word "oriented" means showing .....
   (a) ambiguity (b) functionality
   (c) methodology (d) none
2. OOP decompose the problem into smaller unit known as .....
   (a) object (b) class
   (c) function (d) none
3. Object-oriented programming language gives emphasis on .....
   (a) flowchart (b) algorithm
   (c) data (d) none

4. Object-oriented programming language follows the ..... approach.
   (a) top-down (b) bottom-up
   (c) top-bottom (d) none
5. The function that operate on the data of an object are tied together in the data structure, this concept is known as .....
   (a) Polymorphism (b) inheritance
   (c) encapsulation (d) none
6. ..... allows creation of new classes from old one.
   (a) Polymorphism (b) inheritance
   (c) class (d) none
7. ..... are basic run-time entities.
   (a) data (b) classes
   (c) objects (d) none
8. The object with the attributes and methods are grouped together to form a .....
   (a) data (b) class
   (c) object (d) none
9. An object is an ..... of a class.
   (a) abstraction (b) operation
   (c) instance (d) none
10. The code redundancy can be eliminated through .....
    (a) polymorphism (b) inheritance
    (c) class (d) none
11. C++ was developed by .....
    (a) Stroustrup (b) Denis Richie
    (c) Pascal (d) none
12. The statement which starts with the symbol // is treated as ..... comment.
    (a) one (single) line (b) multiline
    (c) miniline (d) none

Answers

1. (b)	2. (a)	3. (c)	4. (b)	5. (c)	6. (b)	7. (c)	8. (b)	9. (c)	10. (b)
				11. (a)	12. (a)				

Practice Questions

Q.I Answer the following questions in short:

1. What is meant by OOP?
2. List out different concepts in OOP?
3. What is Class?
4. Define the term Polymorphism in detail.
5. What is C++?

Q.II Answer the following questions:

1. Explain the term Data Abstraction in detail.
2. What are the benefits of OOP?
3. Write short note on: Message Passing.
4. What are the Applications of OOP?
5. Enlist various Applications of C++.
6. Explain the following terms:
   (a) Object
   (b) Class.
7. Write a C++ program of print "Welcome to BCA-IV".
8. Compare C and C++.
9. List various features of C++.
10. State advantages and disadvantages of C++.

Q.III Define the term:

1. Encapsulation
2. Data abstraction
3. Polymorphism
4. Inheritance
5. Object
6. Classes
7. Dynamic binding.

Previous Exam Questions

April 2018

1. List any four features of OOPs. **[2 M]**

Ans. Refer to Section 1.3.

October 2018

1. What is encapsulation? **[2 M]**

Ans. Refer to Section 1.2.4.

2. Differentiate between object oriented programming and procedure oriented programming. **[4 M]**

Ans. Refer to Section 1.1.2.

April 2019

1. What is data abstraction? **[2 M]**

Ans. Refer to Section 1.2.3.

■■■

2... Beginning with C++

Objectives...

- To understand Basic Concepts of C++.
- To learn Operators, Namespace, Manipulators, Variables in C++.
- To learn Data types and keywords.
- To study about functions.

2.1 INTRODUCTION

- C++ is a general purpose programming language which has been derived from C programming language.
- C++ could be considered a superset of C language with extensions and improvements with object oriented features included in it. C++ runs on a variety of platforms, such as Windows, Mac OS, UNIX, LINUX etc.
- C++ is regarded as a middle-level language, as it comprises a combination of both high-level and low-level language features.
- C++ is popular languages because of following reasons:

  1. C++ is ideally suited for development of reusable software.

  2. C++ is highly flexible language with versatility.

2.1.1 Tokens [S - 19]

- The smallest individual units in a program are known as Tokens.
- A token is a group of characters that logically belong together.
- Token is a sequence of characters from the character set of C++, which is identified by the compiler.
- C++ tokens are keywords, literals, identifiers, operators, other symbols (separators) are shown in Fig. 2.1.
- A C++ program or application is written using these tokens, white spaces and the syntax of the language.

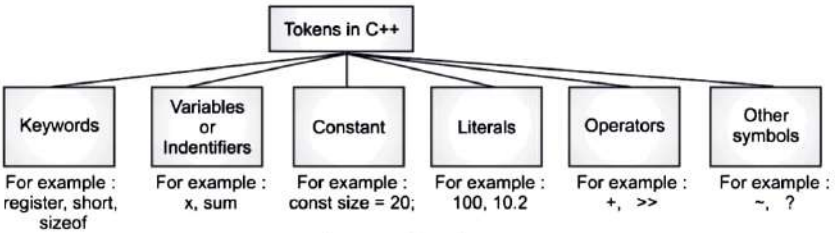


Fig. 2.1: Tokens in C++

2.1.2 Identifiers [W-18]

- An Identifier is any name of variables, functions, classes etc. given by the programmers.
- Identifiers are the fundamental requirement of any language.
- The identifier is a sequence of characters taken from C++ character set.
- The rules for the formation of an identifier are:
  1. An identifier can consist of alphabets, digits and/or underscores.
  2. An identifier must not start with a digit. It start with an alphabet or underscore(_).
  3. C++ is case sensitive that is uppercase and lowercase letters are considered different from each other.
  4. Identifier should not be a reserved word.
  5. Identifier contains maximum 32 characters.
- Some valid identifiers are: `result, x, _n1, acc_no, basicpay`

2.1.3 Constants [W-18]

- Constant refer to fixed values that the program cannot alter or change.
- A constant is an explicit number or character (such as 1, 0.5, or 'c') that does not change. As with variables, every constant has a type.
- A number which does not change its value during execution of a program is known as a constant.
- A constant in C++ can be of any of the basic data types, const qualifier can be used to declare constant as shown below:

```
const float pi = 3.1415;
```

- The above declaration means that pi is a constant of float types having a value 3.1415.
- Examples of valid constant declarations are:

```
const int rate = 50;
const float pi = 3.1415;
const char ch = 'A';
```

- The following types of constants are available in C++:

1. Integer Constants:

- Integer constants are whole number without any fractional part. C++ allows three types of integer constants.
  - **(i) Decimal Integer Constants:** It consists of sequence of digits and should not begin with 0 (zero).
    For example, 124, –179, +108.
  - **(ii) Octal Integer Constants:** It consists of sequence of digits starting with 0 (zero).
    For example, 014, 012.
  - **(iii) Hexadecimal Integer Constant:** It consists of sequence of digits preceded by ox or OX.

2. Character Constants:

- A character constant in C++ must contain one or more characters and must be enclosed in single quotation marks.
- For example 'A', '9', etc. C++ allows nongraphic characters which cannot be typed directly from keyboard, e.g., backspace, tab, carriage return etc. These characters can be represented by using an escape sequence.
- An escape sequence represents a single character. The following table gives a listing of common escape sequences.

Table 2.1: Escape Sequence

Escape Sequence	Nongraphic Character
\a	Bell (beep)
\n	Newline
\r	Carriage Return
\t	Horizontal tab
\0	Null Character
\\	Blackslash
\?	Quotation mark (?)
\v	Vertical tab
\b	backspace
\'	Single quote (')
\"	Double quote (")

2. Floating Constants:

- Floating Constants are also called real constants.
- They are numbers having fractional parts.
- They may be written in fractional form or exponent form.
- A real constant in fractional form consists of signed or unsigned digits including a decimal point between digits.
  For example 3.0, -17.0, -0.627 etc.

2.1.4 Literals

- Literals (often referred to as constants) are data items that never change their value during the execution of the program.

String Literals:

- A sequence of character enclosed within double quotes is called a String Literal.
- String literal is by default (automatically) added with a special character '\0' which denotes the end of the string.
- Therefore, the size of the string is increased by one character.
- For example, "COMPUTER" will re represent as "COMPUTER\0" in the memory and its size is 9 characters.

2.2 KEYWORDS

- The keyword implements specific C++ language features.
- Keywords are explicitly reserved identifiers and cannot be used as name for the program variables.
- It is mandatory that all the keywords should be in lowercase letters.
- The keywords are reserved words, predefined by the language and user cannot change.
- Keywords are used for specific purposes in C++ and compiler can interprets these words.
- Table 2.2 shows the list of keywords which are common in C and C++ language and Table 2.3 shows the keywords specific to C++.

Table 2.2: Keywords common to C and C++

auto	default	float	return	union
break	do	for	shortsigned	unsigned
case	double	goto	sizeof	void
char	else	ifint	staticstruct	volatile
const	enum	long	switch	while
continue	extern	register	typedef	

Table 2.3: Keywords specific to C++

asm	inline	template	using
bitand	new	this	virtual
bitor	namespace	throw	wchar_t
catch	operator	true	
class	private	try	
delete	protected	typeid	
friend	public	typename	

Table 2.4: New Keywords which are supporting new features in C++

bool	false	true
const_cast	mutable	typeid
dynamic_cast	namespace	typename
explicit	reinterpret_cast	using
export	static_cast	wchar_t

2.3 DATA TYPES

2.3.1 Basic Data Types

- In programming, we store the variables in our computer's memory, but the computer has to know what kind of data we want to store in them, since it is not going to occupy the same amount of memory to store a simple number than to store a single letter or a large number, and they are not going to be interpreted the same way.
- The memory in our computers is organized in bytes.
- A byte is the minimum amount of memory that we can manage in C++. A byte can store a relatively small amount of data: one single character or a small integer, (generally an integer between 0 and 255).
- In addition, the computer can manipulate more complex data types that come from grouping several bytes, such as long numbers or non-integer numbers.
- Data types in C++ are classified into different categories is shown in Fig. 2.2.

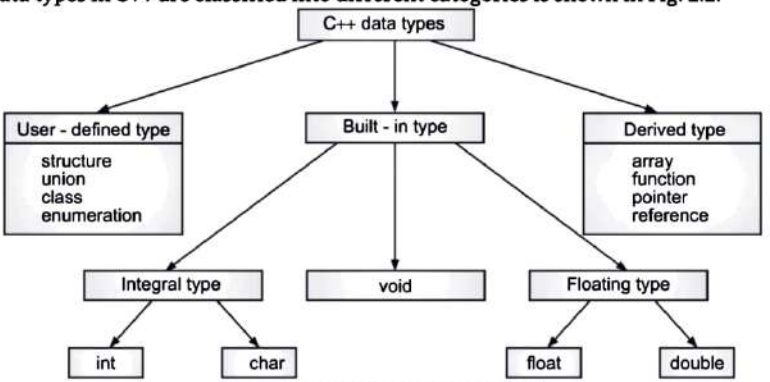


Fig. 2.2: Hierarchy of C++ data types

- Following table list down seven basic C++ data types:

Table 2.5: Basic C++ Data Types

Type	Keyword
Boolean	bool
Character	char
Integer	int
Floating point	float
Double floating point	double
Valueless	void
Wide character	wchar_t

- The following table shows the variable type, how much memory it takes to store the value memory, and what is maximum and minimum value which can be stored in such type of variables.

Table 2.6: Data Types, size and its range

Data Type	Typical Bit Width	Typical Range
char	1 byte	–127 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	–127 to 127
int	4 bytes	–2147483648 to 2147483647
unsigned int	4 bytes	0 to 4294967295
signed int	4 bytes	–2147483648 to 2147483647
short int	2 bytes	–32768 to 32767
unsigned short int	Range	0 to 65,535
signed short int	Range	–32768 to 32767
long int	4 bytes	–2,147,483,647 to 2,147,483,647
signed long int	4 bytes	s3ame as long int
unsigned long int	4 bytes	0 to 4,294,967,295
float	4 bytes	+/– 3.4e +/– 38 (~7 digits)
double	8 bytes	+/– 1.7e +/– 308 (~15 digits)
long double	8 bytes	+/– 1.7e +/– 308 (~15 digits)
wchar_t	2 or 4 bytes	1 wide character

2.3.2 User Defined Data Types

1. Enumerations:

- An enumeration is a user defined type consisting of a set of named constant called enumerators.
- Enumerations are a way of defining constants.
- **Syntax**:

```
enum tag {member1, member2 ……., membern};
```

- Here, enum is the keyword and tag is the name of enumeration and member1, member2 ..... are identifiers which represents integer constant values.
- The member should be unique. Once, the enumeration is defined then variable or object of that type is defined.

Syntax:

```
enum tag {member1, member2 ……} variables;
                OR
enum tag {member1, member2 ……} object_name;
```

- Here, all members take integer values starting with zero i.e. member 1 is assigned 0, member 2 is assigned 1 ..... and so on.

For example,

```
enum colors {black, green, red, yellow, blue};
enum colors background;
```

Here,

```
Black = 0
green = 1
red = 2
   .
   .
   .
```

- Enumeration variables are particularly useful as flags to indicate various options are to identify various conditions.
- It increases clarity of the program. If we want to have blue colors for the background, it is easier to understand with,

  ```
  background = blue;
  ```

  than with,

  ```
  background = 4;
  ```
- It makes program more readable.
- Enumerated types are valuable when an object can assume a known and reasonably limited set of values.

2. Structure:

- 'C' structure is a collection of data items of different data types. Structure is a collection of variables under a single name.
- Variables can be of any type such as, int, float, char etc. The main difference between structure and array is that arrays are collections of the same data type and structure is a collection of variables under a single name.

Syntax:

```
struct structure_Name
{
   data_type member 1;
   data_type member 2;
   ..........
};
```

- The structure is declared by using the keyword struct followed by structure name, also called a tag.
- Then the structure members (variables) are defined with their type and variable names inside the open and close braces { }.
- Finally, the closed braces end with a semicolon denoted as; Structure in C++ is define as,

```
struct student
{
   int roll_no;
   char name[20];
};
```

- For example,

```
struct student s;
printf ("%s",  s.name);
```

3. Union:

- A union is a user defined data or class type that, at any given time, contains only one object from its list of members, (although that object can be an array or a class type).

Syntax:

```
union tag
{
   member1;
   member2;
   '
   '
   member n;
} objectname;
```

Example:

```
union example
{
   int a;
   char b;
   float c;
} e;
```

4. Class:

- Class is user define data type with data elements and functions.
- A class in C++ is an encapsulation of data members and functions that manipulate the data.
- The class can also have some other important members which are architecturally important.
- The data members can be of any legal data type, a class type, a struct type etc., they can also be declared as pointers and accessible normally as like other data members.
- **Access Level:** The classes in C++ have three important access levels. They are Private, Public and Protected. The explanations are as follows:

  (i) Private: The members are accessible only by the member functions or friend functions.

  (ii) Protected: These members are accessible by the member functions of the class and the classes which are derived from this class.

  (iii) Public: Accessible by any external member.

- **For example:**

```
class Example_class
{
   private:
         int x; //Data member
         int y; // Data member
   public:
         Example_Class()//Constructor for the C++
         {
            x = 0;
            y = 0;
         };
         ~Example_Class() //destructor for the C++
         { }
         int Add()
         {
            return x+y;
         }
};
```

2.3.3 Derived Data Types in C++

1. **Arrays:**

- In C, array size is exact same length of the string constant.
  For example: `char string[5] = 'abcde';  //valid in C.`
- But in C++, the array size is one larger than the number of characters present in the string. Here, "\0" is treated as a one character.
  For example:
  ```
  char string[5] = "abcde";   //Invalid in C++
  char string[5] = "abcd";    //Valid in C++
  ```

2. **Functions:**

- A function is a discrete block of statements that perform certain task.
- Once, a function has been written to play a particular role it can be called upon repeatedly throughout the program.
- Functions are either created by the user or supplied by the system. We can also write our own function according to the requirements.
- A function must be declared before its use, such a declaration is known as a function prototype it ends with semicolon.
  Syntax: `return_type function_name (parameters);`
  For example: `int max (int x, int y);`

3. Pointers:

- A pointer is a variable that represents the location, (rather than the value) of a data item such as a variable or an array element, (i.e. pointers refer *l*-value).

(i) const pointers:

- C++ also has constant pointer declaration and pointer to constant declaration.

For example:

```
int *const m = "xyz";  //constant pointer
int const *m = &l;     //pointer to a constant
```

- In constant pointer we cannot change address of 'm' and in pointer to a constant, the contents of 'l' cannot change.

(ii) void pointer:

- void pointers used to point the other data type of variable in C++. If we define,

```
int *m;
float *l;
m = &l;  //results in compilation error
```

- For such type compatibility void pointers are used.
- **For example,**

```
void *v;
int *i;
float *f;
v = &i;
v = &f;
```

Here, void pointer v will hold pointer of any type.

(iii)void type:

- void is used for two purposes:
  1. If the function is not having any arguments, function_name(void);
  2. To specify the return type of function when function is not returning any value.

```
void function_name (void);
```

2.4 VARIABLES

- A variable is a named location in memory that is used to hold a value that may be modified by the program.
- Variable is the data name that refers to the stored value.
- A variable name is an identifier or a symbolic name assigned to memory location where data is stored.
- A variable provides us with named storage that our programs can manipulate.
- Each variable in C++ has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.
- The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C++ is case-sensitive.

2.4.1 Declaration of Variables

- All variables must be declared before use, although certain declarations can be made implicitly by content.
- A declaration of variable specifies a type, and contains a list of one or more variables of that type as follows:

  Syntax: `type variable_list;`
- Here, type must be a valid C++ data type including char, w_char, int, float, double, bool or any user defined object etc., and variable_list may consist of one or more identifier names separated by commas.
- Some valid declarations are shown here:

```
int i, j, k;
char c, ch;
float f, salary;
double d;
```

- A variable declaration with an initializer is always a definition. This means that storage is allocated for the variable and could be declared as follows:

```
int i = 100;
```

Program 2.1: Program for declaration of variables.

```
#include<iostream>
using namespace std;
int main()
{
   // declaring variables
   int a, b;
   int result;
   // initialization of variable
   a = 5;
   b = 2;
   a = a + 1;
   result = a - b;
   // print out the result
   cout << result;
   // terminate the program
   return 0;
}
```

Output:

```
4
```

Scope of Variables:

- A scope is a region of the program and broadly speaking there are three places where variables can be declared:
  (i) Inside a function or a block which is called local variables.
  (ii) In the definition of function parameters which is called formal parameters.
  (iii) Outside of all functions which is called global variables.
- All the variables that we intend to use in a program must have been declared with its type specifier in an earlier point in the code, like we did in the previous code at the beginning of the body of the function main when we declared that a, b, and result were of type int.
- A variable can be either of global or local scope.
- Variables that are declared inside a function or block are local variables. They can be used only by statements that are inside that function or block of code.
- Local variables are not known to functions outside their own.
- A global variable is a variable declared in the main body of the source code, outside all functions, while a local variable is one declared within the body of a function or a block.

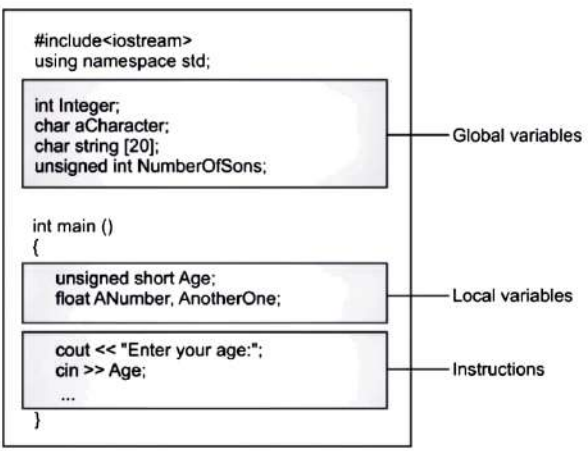


Fig. 2.3: Scope of Variable

- Global variables can be referred from anywhere in the code, even inside functions, whenever it is after its declaration.
- The scope of local variables is limited to the block enclosed in braces ({ }) where they are declared.
- For example, if they are declared at the beginning of the body of a function, (like in function main) their scope is between its declaration point and the end of that function.

- Global variables are defined outside of all the functions, usually on top of the program. The global variables will hold their value throughout the lifetime of your program.
- A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration.

Program 2.2: Program using global and local variables.

```
#include <iostream.h>
//using namespace std;
// Global variable declaration:
int g;
int main()
{
   // Local variable declaration:
   int a, b;
   // actual initialization
   a = 10;
   b = 20;
   g = a + b;
   cout << g;
   return 0;
}
```

Output:

```
30
```

2.4.2 Dynamic Initialization of Variables

- When declaring a regular local variable, its value is by default undetermined. But you may want a variable to store a concrete value at the same moment that it is declared. In order to do that, you can initialize the variable.
- There are two ways to do this in C++:
  (i) The first one, known as C-like initialization, is done by appending an equal sign followed by the value to which the variable will be initialized: `type identifier = initial_value;`

  For example, if we want to declare an int variable called a initialized with a value of 0 at the moment in which it is declared, we could write: `int a = 0;`

  (ii) The other way to initialize variables, known as constructor initialization, is done by enclosing the initial value between parentheses [( )]: `type identifier (initial_value);`

  For example: `int a(0);`
- Both ways of initializing variables are valid and equivalent in C++.

- Some examples are:

```
int d = 3, f = 5;     // initializing d and f.
byte z = 22;          // initializes z.
double pi = 3.14159; // declares an approximation of pi.
char x = 'x';         // the variable x has the value 'x'.
```

Program 2.3: Program for initialization of variables.

```
//initialization of variables
#include<iostream.h>
//using namespace std;
int main()
{
   int a=5;              // initial value = 5
   int b(2);             // initial value = 2
   int result;           // initial value undetermined
   a = a + 3;
   result = a - b;
   cout << result;
   return 0;
}
```

Output:

```
6
```

2.4.3 Reference Variables [S - 18]

- A reference variable is an alias or an alternative name for an object to which it is referring.
- It is a variable provides an alias (alternative name) for a previously defined variable.
- A reference variable is created as,

```
data_type &reference_name = variable_name;
```

- C++ references allows you to create a second name for the variable that you can use to read or modify the original data stored in that variable.
- While this may not sound appealing at first, what this means is that when you declare a reference and assign it a variable, it will allows you to treat the reference exactly as though it were the original variable for the purpose of accessing and modifying the value of the original variable even if the second name (the reference) is located within a different scope.
- This means, for instance, that if you make your function arguments references, and you will effectively have a way to change the original data passed into the function.
- This is quite different from how C++ normally works, where you have arguments to a function copied into new variables.
- Reference variables also allows you to dramatically reduce the amount of copying that takes place behind the scenes, both with functions and in other areas of C++, like catch clauses.

- For example:

```
int a = 20;
int &b = a;  // & is a reference operator
2132   ← Address
[20]   ← Value
a      ← Variable name
b      ← reference to a
```

- Manipulating a reference to an object allows manipulation of the object itself. This is because no separate memory is allocated for a reference.
- A reference to a variable is created using the address of operator (&). When the & operator is used in the declaration, it becomes the reference operator.
- A reference variable is declared as follows:

```
datatype &refvar = variable;
```

where, refvar is name of reference variable.

- In above example, a is an integer variable whose address (*l*-value) is 2132 which is intialize to 20. In the second statement, 'b' is a reference to 'a'. If we manipulate the value of b (say 10), then the vlaue of the memory location 2132 gets changed or in other words the value of 'a' changes.

Program 2.4: Program for reference variable.

```
#include<iostream.h>
//using namespace std;
int main()
{
   int p = 50;
   int *a = &p;
   int &b = *a;
   cout<<"*a="<<*a<<"b="<<b<<endl;
   int x = 10;
   int &y = x;
   x = x + 10;
   cout<<"x="<<x<<"y="<<y<<endl;
   return 0;
}
```

Output:

```
*a=50 b=50
x=20 y=20
```

2.5 OPERATORS IN C++

- Operators are special symbols used for specific purposes.

1. Arithmetical Operators:

Arithmetical operators +, -, *, /, and % are used to performs an arithmetic (numeric) operation. You can use the operators +, -, *, and / with both integral and floating-point data types. Modulus or remainder % operator is used only with the integral data type. Operators that have two operands are called binary operators.

Table 2.7: Arithmetic Operators

Operators	Meaning
+	Addition
–	Subtraction
*	Multiplication
/	Division
%	Modulus

2. Relational Operators:

The relational operators are used to test the relation between two values. All relational operators are binary operators and therefore require two operands. A relational expression returns zero when the relation is false and a non-zero when it is true. The following table shows the relational operators.

Table 2.8: Relational Operators

Relational Operators	Meaning
<	Less than
<=	Less than or equal to
==	Equal to
>	Greater than
>=	Greater than or equal to
! =	Not equal to

3. Logical Operators:

The logical operators are used to combine one or more relational expression. The logical operators are:

Table 2.9: Logical Operators

Operators	Meaning
||	OR
&&	AND
!	NOT

4. **Unary Operators:**

C++ provides two unary operators for which only one variable is required.

For example, a = –50;

a = +50;

Here plus sign (+) and minus sign (-) are unary because they are not used between two variables.

5. **Assignment Operator:**

The assignment operator '=' is used for assigning a variable to a value. This operator takes the expression on its right-hand-side and places it into the variable on its left-hand-side.

For example: m = 5;

The operator takes the expression on the right, 5, and stores it in the variable on the left, m.

x = y = z = 32;

This code stores the value 32 in each of the three variables x, y, and z.

In addition to standard assignment operator shown above, C++ also supports compound assignment operators.

Table 2.10: Assignment Operators

Operators	Example	Equivalent to	Description
+=	A += 2	A = A + 2	Addition and Assignment
-=	A -= 2	A = A - 2	Subtraction and Assignment
%=	A %= 2	A = A % 2	Modulus and Assignment
/=	A/ = 2	A = A / 2	Division and Assignment
*=	A *= 2	A = A * 2	Multiplication and Assignment

6. **Increment and Decrement Operators:**

C++ provides two special operators '++' and '--' for incrementing and decrementing the value of a variable by 1.

The increment/decrement operator can be used with any type of variable but it cannot be used with any constant.

Increment and decrement operators each have two forms, pre and post.

The syntax of the increment operator is:

Pre-increment: `++variable`

Post-increment: `variable++`

The syntax of the decrement operator is:

Pre-decrement: `-- variable`

Post-decrement: `variable --`

In Prefix form first variable is first incremented/decremented, then evaluated.

In Postfix form first variable is first evaluated, then incremented/decremented.

```
int x,y;
int i=10,j=10;
x = ++i;      //add one to i, store the result back in x
y= j++;       //store the value of j to y then add one to j
cout<<x;      //11
cout<<y;      //10
```

7. **Conditional Operator:**

The conditional operator?: is called ternary operator as it requires three operands.

Syntax:

```
Conditional_ expression? expression1: expression2;
```

If the value of conditional expression is true then the expression1 is evaluated, otherwise expression2 is evaluated.

```
int a = 5, b = 6;
big = (a > b)? a: b;
```

The condition evaluates to false, therefore biggets the value from b and it becomes 6.

8. **Comma Operator:**

The comma operator gives left to right evaluation of expressions.

When the set of expressions has to be evaluated for a value, only the rightmost expression is considered.

```
int a=1, b=2, c=3, i;  // comma acts as separator, not as an operator
i = (a, b);            // stores b into i
```

Would first assign the value of a to i, and then assign value of b to variable i. So, at the end, variable i would contain the value 2.

9. **New C++ Operators:** C++ contains some new additional operators they are listed below:

Table 2.11: New C++ Operators

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

2.5.1 Scope Resolution Operator [W-18]

- The scope resolution operator is denoted by a pair of colons (::).
- It is use to access global variable even if the local variable has the same name as global.
- When the same function name is used for many classes, then this complexity is solved by using scope resolution operator to indicate the class with which the function is associated.
- This operator can be used to uncover a hidden variable.
- **Syntax:**

```
::variable_name
```

- Suppose in a C program, there are two variables with the same name a. One is global, (outside a function) and other is local, (inside a function).
- We attempt to access the variable a in the function, we always access the local variable (priority to local).
- C++ allows the flexibility of accessing both the variables using scope resolution operator (::).

Program 2.5: Simple scope resolution operator.

```
#include<iostream.h>
int a=10;
int main()
{
   int a=15;
   cout<<"\n Local a="<<a<<"Global a="<<::a;
   ::a=20;
   cout<<"\n Local a="<<a<<"Global a="<<::a;
   return 0;
}
```

Output:

```
Local a=15 Global a=10
Local a=15 Global a=20
```

- In the following example, the declaration of the variable X hides the class type X, but you can still use the static class member count by qualifying it with the class type X and the scope resolution operator.

```
#include<iostream.h>
// using namespace std;
class X
{
public:
static int count;
};
```

```
int X::count = 10;              // define static data member
int main ()
{
   int X = 0;                   // hides class type X
   cout<<X::count<<endl;        // use static member of class X
   return 0;
}
```

- There are two uses of the scope resolution operator in C++.
  1. The first use being that a scope resolution operator is used to unhide the global variable that might have got hidden by the local variables. Hence, in order to access the hidden global variable one needs to prefix the variable name with the scope resolution operator (::).

     For example:

```
int i = 10;
int main()
{
  int i = 20;
  cout << i;    // this prints the value 20
  cout <<::i;   // in order to use the global i one needs to prefix it
                   with the scope resolution operator.
}
```

  2. The second use of the operator is used to access the members declared in class scope. Whenever, a scope resolution operator is used the name of the member that follows the operator is looked up in the scope of the class with the name that appears before the operator.

Program 2.6: Use of scope resolution operator for class.

```
#include<iostream.h>
class T
{
   int a,b;
   public:   void getdata();
             void putdata();
};
void T::getdata()
{
   cin>>a>>b;
}
void T::putdata()
{
   cout<<"a="<<a<<endl;
   cout<<"b="<<b<<endl;
}
```

```
int main()
{
   T a;
   a.getdata();
   a.putdata();
   return 0;
}
```

Output:

```
10 20
a=10
b=20
```

2.5.2 Memory Management Operator [W-18]

- For dynamic memory allocation, C++ provides new operator. In C, we have malloc( ) and calloc( ) functions for dynamic allocation.
- For dynamic deallocation, C++ provides delete( ) operator, which is same as free( ) function in C.

1. **new operator:** This operator is used for allocation of memory.

   Syntax:

   ```
   ptr_var = new datatype;
   ```

   For example,

   ```
   char *P;
   P = new char[5];    // reserves 5 bytes of contiguous memory of
                          characters
   int *x;
   x =  new int;    // allocate 2 bytes for integer
   ```

2. **Delete operator:** The delete operator deallocates or frees the memory allocated by new.

   Syntax:

   ```
   delete ptr_var;
   ```

   The `ptr_var` is the pointer to a data object which is created by new.

   For example,
   ```
   delete P;
   delete x;
   ```

   - If the array is allocated by new, then to free it we use the following syntax,
     ```
     delete [size] ptr_var;
     ```
   - If the size is not specified then entire array get deleted.
     For example, `delete [ ] a;`

Program 2.7: Program for new and delete operators.

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

```
int main()
{
    char *x;
    x = new char[10];
    strcpy(x, "COMPUTER");
    cout<<"\n the string is:"<<x<<endl;
    delete x;
    return 0;
}
```

Output:

```
the string is: COMPUTER
```

Note:

- For malloc( ) we use **sizeof** operator, for new "sizeof" is not required, it automatically compute the size.
- The operators new and delete are overloaded.
- Type casting is not required since new operator automatically returns the correct pointer type.

2.6 MANIPULATORS

- Manipulators are the functions used in input/output statement.
- All the manipulators are defined in header field **iomanip.h.**
- We can use chain of manipulator that is one or more manipulators in statement.

Table 2.12: Manipulators

Manipulators	Equivalent I/O function
setw()	width()
setprecision()	precision()
setfill()	fill()
setioflags()	setf()
resetioflags()	unsetf()

1. Setting of field width:

- setw() is used to define the width of field necessary for the output for a variable.
- For example: `setw(5);` This function will reserve 5 digits for a number.

```
setw(5);
cout << 123;
```

Output:

		1	2	3

2. Setting precision for float nos:

- We can control number of digits to be displayed after decimal point for float numbers by using setprecision() function.

```
setprecision(2);
```

- This function will display only two digits after a decimal point.
- For example:
```
setprecision(2);
cout<<3.14159;
```
Output: 3.14

3. Filling of unused positions:

- Filling of unused positions of the field can be done by using setfill().
```
setfill('*');
```
- Here, unused positions will be filled by using * sign.
```
setw(5);
setfill ('$');
cout << 123;
```
Output:

$	$	1	2	3

4. setbase():

- It is used to show the base of a number, for example:
```
setbase(10);
```
- It shows that all numbers are decimal numbers.

5. Flags:

- C++ defines some format flags for standard input and output, which can be manipulated with the `flags()`, `setf()` and `unsetf()` functions.
- The flags() function either return the format flags for the current stream or sets the flags for the current stream to be f.

Syntax:
```
fmtflags flags();
fmtflags flags(fmtflags f);
```

6. setf() and unsetf(): [S - 19]

- setf() function can also be used with only one argument. Arguments will be as follows:

Table 2.13: Arguments

Arguments	Use
ios:: showbase	It helps to use base indicator an output
ios:: uppercase	It helps to use uppercase letters.
ios:: skipws	It skips (blank) white spaces from input
ios:: showpoint	It helps to display trailing zeros.
ios:: showpos	It helps to display + sign for number.

- The function `unsetf( )` is used to clear the given flags associated with the current system.

Syntax: `void unsetf(fmtflags flags);`

7. endl:

- This manipulator used in an output statement, causes a linefeed to be inserted. It has the same effect as using the new line character "\n".

For example:

```
cout <<"a=" << a << endl
     <<"b=" << b << endl
     <<"c=" << c << endl;
```

If we assume value of variable as 2215, 181 and 17 the output will:

a = 2 2 1 5

b = 1 8 1

c = 1 7

2.7 FUNCTIONS

- Functions are the basic building blocks of C++ and the place where all program activity occurs. A function is a group of statements that together perform a task.
- A function is a subprogram that acts on the program data and often returns values.
- A program written with numerous functions is easier to maintain, update and debug than one very long or large program.
- Functions are used to implements large programs. In structured programming functions are used to break the larger programs into smaller ones; which makes debugging of program easier.
- Each function has its own name. When that name is encountered in a program, the execution of the program branches to the body of that function.

Features of Functions:

1. The structure of a function definition is look like the structure of main(), with its own list of variables declaration and program statements.
2. A function can have a list of zero or more parameters inside its brackets, each of which has a separate type.
3. A function may have more than one "return" statement in which case the function definition will end execution as soon as the first "return" is reached.
4. Function declarations are like variable declaration, (functions specify which return type the function will return).
5. A function has to be declared in a function declaration at the top of the program and before it can be called by main().

Advantages of Functions:

1. To reduce the size of a program by calling them at different places in the program.
2. Functions can be compiled separately.
3. Using functions, programs are easier to design, debug and maintain.
4. Reuse of the function is possible in multiple programs.
5. The functions provide interfaces to communicate with object.

How a Function Works?

- A C++ program does not execute the statement in a function until the function is called or invoked.
- When the C++ program's function is called or invoked, the control passes to the function and returns back to the calling part after the execution of function is over or exit.
- The calling program can send information to the functions in the form of argument. An argument of function stores data needed by the function to perform its task.
- Following program shows working of functions:

Program 2.8: Program for working of function.

```
#include<iostream>
using namespace std;
int add(int x, int y)  //declaration of function
{
   int z;
   z = x + y;
   return(z);
}
int main()
{
   int a;
   a = add(10, 20);  //calling a function
   cout<<"The result is" << a;
   return 0;
}
```

Output:

```
The result is 30
```

- In above program, we declare the variable a of type in main() then a call to a function named add().
- We will able to see the similarity between the structure of the call to the function and the declaration of the function itself in the code lines below.

```
int add(int x, int y)
          ↑      ↑
a = add(10,    20);
```

- Within the main() we called the add(), passing 10, 20 values that corresponds to the int x and int y parameters declared for the add().
- add() declares a new variable int z and by means of the expression z = x + y; it assigns to z the result of x plus y, because the passing parameters for x and y are 10, 20 respectively the result in 30.
- The function has elements like: 1. A function declaration, 2. A function definition, 3. A function call, and 4. Returning from function. Let us see in detail.

1. A Function Declaration:

- A function declaration tells the compiler about a function's name, return type and parameters. This is also referred as Function Prototyping.
- A function is declared with a prototype, which consists of return type, a function name and argument list.
- **Syntax for function declaration:**

```
return_type function_name(argument1, argument2, ...)
```

- A function can be declared either in the main function or in a class.
- For example,

```
int add (int a, int b);    //function add with 2 arguments
void func(void);           //function prototype with no arguments
float display();           //function prototype with no arguments
```

Program 2.9: Program illustrates the function declaration.

```
#include<iostream.h>
using namespace std;
int main()
{
   int a, b,c;
   //function declaration
   cout<<"Enter two numbers \n";
   cin>>a>>b;
   int add (int, int) ;
   cout<<"addition is"<<c;
}
//function definition
int add(int x, int y)          //function header
{
   int sum;
   sum = x + y;
   return sum;
}
```

Output:

```
Enter two numbers
  5       6
addition is 11
```

2. Function Definition:

- The function definition tells the compiler what task the function will be performed.
- Function definition consists of function prototype and actual body of a function. It can be written anywhere in C++ program.
- The function definition consists of two parts:
  (i) Function header or Prototype, and
  (ii) Function body.

- The function header is similar to declaration. The function header is a replica of function declaration. The basic difference between these two is that the declaration in the calling function will end with semicolon whereas, in the called function it is not.
- The function body consists of local variables (if needed) and simple or compound statements.
- **Syntax:**

```
return_type function_name(argument list)
{
    local_variables declaration;
    statements;
}
```

- Here, argument list consists of any number of arguments with any data type or no arguments. The return_type is the type of return value from the function. If we are using return statement, then return_type should be specified otherwise void is used as return_type.
- For example,

```
int add (int a, int b)  //function header
{
    int total          //local variable
    total = a + b;
    return total;            //function body
}
```

Program 2.10: Program for function definition.

```
#include<iostream>
using namespace std;
int mult(int a, int b)
{
    int prod;
    prod = a * b;
    return(prod);
}
int main()
{
    int x = 3, y = 4, z;
    z = mult(5, 4);
    cout<<"The first product is="<<z<<"\n";
    cout<<"The second product is="<<mult(x,y)<<"\n";
    cout<<"The third product is="<<mult(5,4);
    return 0;
}
```

Output:

```
The first product is=20
The second product is=12
The third product is=20
```

3. A Function Call:

- Function can be called directly by simply writing the function name with arguments or indirectly in the expression if it returns a value.
- We have to pass the actual parameters (arguments) to the functions. In the function definition, the arguments are referenced. Therefore, calling a function is also known as Function Reference.
- The arguments which we pass should match with the formal parameters which are declared into the function. The data type and the order should also be matched.
- For example,

```
(i) add (x, y);              //function add called
(ii) display( );             //function display called
(iii) larger = max(a,b);     //function max called
(iv) total = add(10, 20)     //function add called
```

4. Returning from Function:

- To return from function we use return statement.
- *return* statement pushes a value onto the return stack and control is return back to the calling function.
- If return type is void, the function does not return any value.

Syntax:

```
return;  OR
return value;
```

Program 2.11: Program for return statement.

```
/* function void */
   #include<iostream>
   using namespace std;
   void func(void)
   {
    cout<<"This is void function";
   }
   int main()
   {
    func();
   }
```

Output:

```
This is void function
```

2.8 FUNCTION PROTOTYPING [W-18]

- One of the most important features of C++ is the function prototype.
- A function prototype is a declaration that delivers the return type and the parameters of a function.
- A function is declared with a prototype.

- The function prototype tells the compiler the name of the function, the type of data returned by the function, the number of parameters the function expects to receive, the types of the parameters and the order in which these parameters are expected.
- The compiler uses function prototype to validate function calls.
- The function prototype is also called the function declaration.
- Functions are declared similar to variables, but they enclose their arguments in parenthesis [( )].
- The function prototype describes the function interface to the compiler by giving details like the numbers and type of arguments and the type of return values.
- **Syntax:** `return_type function_name (list of parameters);`
- **For examples:**
  1. `int add(x, y);` //Declaration of add with x, y arguments.
  2. `int max( );` //Declaration of max with no argument.

2.9 INTERACTION BETWEEN FUNCTIONS

- The function [main() and other] interacts with each other through parameters. We pass the parameter to the function for the communication of calling function and the called function.
- There are two types of parameters actual parameters (used in the function call) and formal parameters (used in function definition).

2.9.1 Call by Reference [S - 18]

- When a function is called the address of the actual parameters are copied on to the formal parameters, though they may be referred by different variable names.
- This content of the variables that are altered within the function block are return to the calling function program. As the formal and the actual arguments are referencing the same memory location or address. This is known as call by reference, when we use this concept is functions.
  - Called function makes the alias of the actual variable which is passed.
  - Actual variables reflect the changes made on the alias of actual variables.
  - Number of values can be modified together.
  - For call by reference concept C++ uses a reference operator (&).
  - & operator allows true call by reference functions, eliminating the need to dereference arguments passed to the function.
  - The & operator is placed before the variable name of argument in the parameter list of the function. Then the function can be called without passing the address and without dereferencing within the function.

Program 2.12: Program to swap the values by call by reference.

```
#include<iostream.h>
void swapr (int &p, int &q)
{
    int dummy;
    dummy = p;
    p = q;
    q = dummy;
}
```

```
int main()
{
    int x = 225, y = 305;
    cout<<x<<"\t"<<y<<endl;
    swapr(x,y);
    cout<<"After swap"<<endl;
    cout<<x<<"\t"<<y;
}
```

Output:

```
225     305
After swap
305     225
```

- Let us see how this program gets executed in the memory. This is shown by Fig. 2.4.
- In the Fig. 2.4, p is alias of x and q is alias of y. Therefore, we do not need 4 bytes allocated for p and q, [each will have 2 bytes].
- Here, it first copies the value of x i.e. p to dummy. Then the value of q or y is assigned to P and then the value of dummy is assigned to y or q. In this way the values are swapped without wasting more amount of memory. This is because the formal parameters are of reference type.
- The compiler always treats them similar to pointers but compiler does not allow the modification of the pointer value. The changes made to the formal parameters p and q are reflected on the actual parameters x and y as shown in Fig. 2.4.
- The following points can be noted about reference parameters:

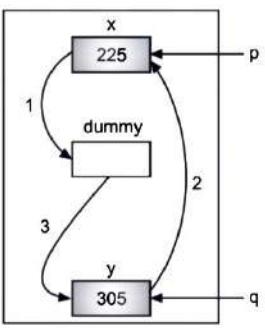


Fig. 2.4: Call by Reference

1. Here, parameters passed to a function by its reference and not by its value. Reference means address of variable is passed to function.
2. A reference cannot be null.
3. It should always refer to a object or a variable.
4. Once, a reference points towards one object or variable, then it cannot be changed. Even it should not point towards different object.
5. The reference does not require any explicit mechanism to dereference the memory. Whenever the program execution terminates, all the memory locations gets free. References are also lost.
6. It is good for performance because it eliminates the overhead of copy large amount of data.

2.9.2 Call by Value

- In call by value, we called function makes the copy of actual variable passed.
- The actual variables do not reflect the changes made on the copies of actual variables. In call by value only one value can be returned.

Program 2.13: Program for call by value.

```
#include<iostream.h>
void swap (int &p, int &q)
{
    int dummy;
    dummy = p;
    p = q;
    q = dummy;
}
int main()
{
    int x = 225, y = 305;
    cout<<x<<"\t"<<y<<endl;
    swap(x,y);
    cout<<"After swap"<<endl;
    cout<<x<<"\t"<<y;
}
```

Output:

```
225      305
After swap
305      225
```

Explanation of Program 4.6:

- Here, we have used three variables of type integer p, q and dummy. Hence, we required 3 memory location each of 2 bytes. Therefore, 3 * 2 = 6 bytes are allocated for swapping the values of two variables as shown in Fig. 2.5.
- Fig. 2.5 specifies the steps with numbers:
  1. value x is passed to p.
  2. value y is passed to q.
  3. value p is passed to dummy.
  4. value q is assigned to p.
  5. value of dummy assigned to q.
- The above program has not return any values. The return value is actually the result of computation in the called functions.

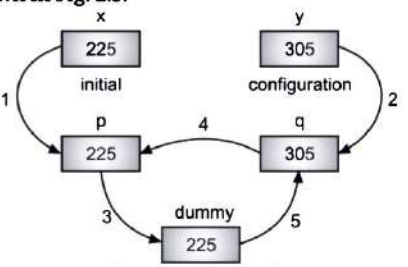


Fig. 2.5: Call by Value

- The returned value is stored in a data type in the functions. We can also return the expressions. If return statement is not present into the function, it means that the return type is **void**.
- We know that there is not restriction in passing any number of values to a function. But for return value there is a restriction. So we can pass whole array to a function but it can return single value only. The program 2.14 explain this concept well.

Program 2.14: Program to find largest no in an array.

```
#include<iostream.h>
int main()
{
   int a[] = {5, -2, 82, 15, 24, 9};
   int s = 5, max;
   int maxi (int a[], int s);    //prototype;
   max = maxi(a,s);
   cout<<max;
}
int maxi (int p[], int s)
{
   int i, maxp = 0;
   for (i=0;i<s;i++)
   {
         if (p[i]>maxp)
         {maxp = p[i];}
   }
   return maxp;
}
```

Output:

```
82
```

2.9.3 Return by Reference [S - 19]

- The function can also return the reference of variable. This reference variable is actually an alias for the referred variable.
- The method of returning reference is used generally in operator overloading. This method will form a cascade of member function calls.

  For example: `cout << k << a;`
- This statement is a set of cascaded calls which returns reference to the object cout.

Program 2.15: Program for function which returns value by reference.

```
#include<iostream.h>
int &Large (int &m, int &n);   //function prototype
int main()
{
   int a,b,m,n;
```

```
    cout<<"Enter values for two integers";
    cin>>a>>b;
    //the function call is given
    //the reference which is returned is assigned to - 1
    Large (m, n) = - 1;
    cout<<"First number"<<m;
    cout<<"second number"<<n;
}
int &Large (int &p, int &q)
{
    if (p>q)
    {return p;}  //function returns the value through reference
    else
    {return q;}
}
```

Output:

```
Enter values for two integers     10     20
First number     10
Second number     - 1
If we executive the program one more time then output will be:
Enter values for two integers       8      3
First number     - 1
Second number     3
```

- When the program gets executed, it returns the reference to the variable which holds the larger value. And this assigns the value - 1 to it.

2.10 INLINE FUNCTIONS [S - 18, 19; W - 18]

- C++ inline function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.
- Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.
- The advantage of using function is to save the memory space. Consider a situation where a small function is called number of times calling a function requires certain overheads such as:
  - Passing to the function,
  - Passing the values or push arguments into the stack,
  - Save the registers,
  - Return back to calling function.

- In such a situation (if small program called number of times), execution time requires more. For the above problem, C programming language provides macros. But macros are called at execution time, so usual error checking does not occur during compilation.
- For the above situation, C++ compiler put the code directly inside the function body of calling program. Every time when function is called, at each place the actual code from the function would be inserted, instead of a jump to the function. Such functions are called inline functions.
- A function can be defined as an inline functions by writing the keyword inline to the function header as shown below:

```
inline function_name (list of arguments)
{
   //function body
}
```

- When we do not specify keyword inline then only one copy of function code (object code) is called depending upon the functions calls. This is illustrated in Fig. 2.6.

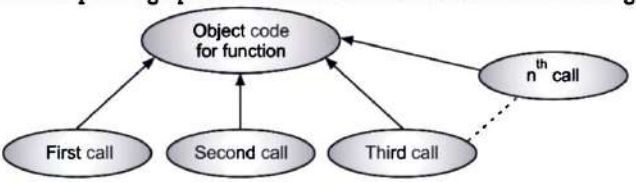


Fig. 2.6: Calls Given to a Function

- But as we said above, lot of time is needed for all these calls to execute. If we make this function inline then the calling statement is replaced with the function code. So it minimizes the time spent on jumping, pushing arguments and so on. This is shown in Fig. 2.7.

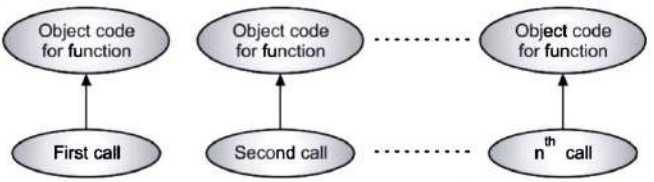


Fig. 2.7: Calls given to Inline Function

Program 2.16: Program illustrates the use of inline function to compute square of a number. [S - 18]

```
#include<iostream.h>
inline int sq (int x)
{
   int p;
   p = x * x;
   return p;
}
```

```
int main()
{
   int n;
   cout<<"Enter number";
   cin>>n;
   cout<<"square is"<<sq(n)<<endl;
}
```

Output:

```
Enter number 12
square is 144
```

2.11 DEFAULT ARGUMENTS [S - 18]

- C++ allows to define an argument whose value will be automatically used by the compiler, if it is not provided during the function call, this argument is called as default arguments.
- When declaring a function, we can specify a default value for each parameter in C++. In C++, the concept of default argument is used. We know that when a function call is given the number of arguments of function definition and function call should be same; otherwise, error will be displayed. But in C++ we can specify a default value for each parameter when function is declared.
- The parameters which does not have any default arguments are placed first whereas those with default values are placed later. Therefore, we can call the same function with fewer arguments than the defined one in prototype. The default values are specified when the function is declared.
- Compiler takes care of all the default arguments and values through the prototype. In a short way, this means that when a function uses default arguments, the actual function call has the option to specify new values or use the default values.
- The default arguments are generally used in the situation where some arguments are having the same value.

Program 2.17: Program shows the use of default argument.

```
#include<iostream.h>
int mult(int a, int b=2, int c=3, int d=4);  //prototype
int main()
{
   cout<<mult(1,5)<<'\n';
   cout<<mult(3,1,4)<<'\n';
   cout<<mult(6)<<'\n';
   cout<<mult(5,2,1,3)<<'\n';
}
```

```
int mult(int a, int b, int c, int d)
{
    int product;
    product = a * b * c * d;
    return product;
}
```

Output:

```
60
48
144
30
```

- In the first call, we have specified 2 arguments, hence default value of c and d are used. In second call default value d is used. In 3rd call, default value of b, c, d are used.
- The default arguments are checked for the type at the time of declaration whereas they are evaluated at the time of call. The default values are checked from right to left. Therefore, if we have declared the prototype in following manner, compiler tells it whether it is valid or invalid.
- **For example:**
  1. `add (int a, int b, int c = 30, int d = 40); valid`
  2. `add (int a = 5, int b, int c = 30, int d = 40); invalid`
  3. `add (int a, int b, int c = 20, int d); invalid`
  4. `add (int a, int b, int c, int d = 40); valid`
- It is very important to role the two main points about default arguments:
  1. If we first declare a function and then define it, make sure to specify the default argument values in the declaration and not in definition.
  2. Don't specify the default values in the function definition if they are already specified in function declaration.

Program 2.18: Program which consist a function to find out the rate of interest with default arguments. [S - 18]

```
#include <iostream>
using namespace std;
float si(int p, int n, int r=5)
{
    return (p*n*r)/100;
}
int main()
{
    int p, n, r;
    cout<<"Enter principal amount: ";
    cin>>p;
```

```
    cout<<"Enter duration (in years): ";
    cin>>n;
    cout<<"Enter rate of interest: ";
    cin>>r;
    cout<<"Simple interest = "<<si(p, n, r);
    cout<<"Simple interest = "<<si(p, n);
    return 0;
}
```

Output:

```
Enter principal amount: 1000
Enter duration (in years): 2
Enter rate of interest: 10
Simple interest = 200Simple interest = 100
```

Program 2.19: Write a program to calculate area and circumference of a circle using inline function. [W-18]

```
#include<iostream.h>
#include<conio.h>
const pi = 3.14159;

inline float circum(float x)   //finds circumference of circle
{
    return(2*pi*x);
}
inline float area(float x)
//finds area of circle
{
    return(pi*x*x);
}
void main()
{
    float r;
    clrscr();
    cout << "\n Enter the radius of the circle: ";
    cin >> r;
    cout << "\n Circumference: " << circum(r);
    cout << "\n Area: " << area(r);
    getch();
}
```

Output:

```
Enter the radius of the circle: 6
Circumference: 37.68
Area: 113.04
```

Summary

- The smallest individual units in a program are known as tokens.
- An Identifier is any name of variables, functions, classes etc. given by the programmers.
- Constant refer to fixed values that the program cannot alter or change.
- A variable is a named location in memory that is used to hold a value that may be modified by the program.
- A function is a block of code that performs some specific operation or task. The function can invoked, or called, from any number of places in the program. The values that are passed to the function are the arguments, whose types must be compatible with the parameter types in the function definition.
- Functions are of two types in C++ namely, Library functions (User can use library function by invoking function directly; they do not need to write it themselves) and User-defined Functions (defined/developed by user for their own needs).
- Every C++ program has at least one function, which is main(), and all the most trivial programs can define additional functions.
- A function declaration (also called function prototype) consists of a return type, a function name, and a list of arguments as number and type of arguments.
- A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.
- If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.
- The call by value method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
- By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function.
- The call by reference method of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.
- An inline function is a function whose code is copied in place of each function call. In other words, each call to inline function is replaced by its code. Inline functions can be declared by prefixing the keyword inline to the return type in the function prototype.
- Default arguments are used in calls where trailing arguments are missing.

Check Your Understanding

1. Which of the following is legal declaration of a reference?
   (a) int &a = 10;
   (b) int &a = m;
   (c) int &a = sqr(m);
   (d) int *a = &15;

2. Which of the following statements is illegal?
   (a) int *p = new int (15); (b) int *p = new int;
   (c) int *p = new int(10); (d) delete p( );
3. Which of the following is not keyword?
   (a) bool (b) abstract
   (c) protected (d) mutable
4. Which of the following keywords is used to control access to a class member?
   (a) default (b) break
   (c) protected (d) goto
5. Which of the following statements is true in C++?
   (a) Classes cannot have data as public members.
   (b) Structures cannot have functions as members.
   (c) Class members are public by default.
   (d) None of these
6. The ..... operator extracts the value of a variable from cin object.
   (a) extraction (b) insertion
   (c) instance (d) none
7. The words which has predefined meaning and cannot be changed by the users are known as .....
   (a) constants (b) identifiers
   (c) keywords (d) none
8. The maximum number of characters used in identifiers are .....
   (a) 356 (b) 31
   (c) 8 (d) none
9. The dynamic memory allocation can be done through ..... operator.
   (a) new (b) delete
   (c) pointer (d) none
10. Following which keyword is used to declare a constant?
    (a) const keyword (b) #define preprocessor
    (b) both (a) & (b) (d) none of these
11. What is?: called?
    (a) ternary operators (b) binary operator
    (c) arithmetic operator (d) none of these
12. Which keyword is used to access to variable in namespace?
    (a) static (b) using
    (c) const (d) dynamic

Answers

1. (b)	2. (d)	3. (b)	4. (c)	5. (d)	6. (b)	7. (c)	8. (b)	9. (a)	10. (c)
				11. (a)	12. (b)				

Practice Questions

Q.I Answer the following questions in short:

1. What is Token?
2. Enlist various keywords of C++.
3. Define operator. What are the types of Operators in C++?
4. Enlist various user and derived data types.
5. What is function? Enlist its features.
6. How a function works?
7. When should we use an inline function?
8. Does an inline function increase the code size and improve performance?

Q.II Answer the following questions:

1. Explain basic data types in C++.
2. What are manipulators used in C++? Explain one in detail.
3. Explain the term Reference Variable in detail.
4. What is Operator? What are its Types?
5. Explain Memory Management Operators in brief.
6. Explain Scope Resolution Operator with example.
7. Write a program to print year is leap or not.
8. In C++, a variable can be declared anywhere in the scope. What is the significance of this feature?
9. What do you mean by dynamic initialization of a variable? Give an example.
10. What are the benefits of pass by reference method of parameter passing?
11. What are default arguments? Write a program to compute tax. A tax-compute function takes two arguments: amount and tax percentage. Default tax percentage is 15% of income.
12. Write an inline function for finding minimum of two members.

Q.III Define the term:

1. Identifier
2. Constant
3. Token
4. Reference variable
5. Variable
6. Function
7. Inline function
8. Default argument

Previous Exam Questions

April 2018

1. What is Reference Variable? [2 M]

Ans. Refer to Point 2.4.3.

2. What is INLINE function? [2 M]

Ans. Refer to Point 2.10.

3. Explain default argument with the help of suitable example and also write in which situation default arguments are useful. [4 M]

Ans. Refer to Point 2.11.

4. What is need of call by reference? Explain with example. [4 M]

Ans. Refer to Point 2.9.1.

5. Write a C++ program to calculate square and cube of an integer number by using inline function. [4 M]

Ans. Refer to Program 2.16.

6. Write a program which consist a function to find out the rate of interest with default arguments. [4 M]

Ans. Refer to Program 2.18.

October 2018

1. Define the following terms: [2 M]

(i) Identifier

Ans. Refer to Point 2.1.3.

(ii) Constant

Ans. Refer to Point 2.1.4.

2. What is function prototype? [2 M]

Ans. Refer to Point 2.8.

3. Write a note on memory management operators in C++. [4 M]

Ans. Refer to Point 2.5.2.

4. What is inline function? Write difference between macro and inline function. [4 M]

Ans. Refer to Point 2.10.

5. Write a program to calculate area and circumference of a circle using inline function. [4 M]

Ans. Refer to Program 2.19.

6. Explain scope resolution operator with an example. [4 M]

Ans. Refer to Point 2.5.1.

7. Trace the output of the following program and explain it. Assume there is no syntax errors. **[4 M]**

```
void position (int & C1, int C2 = 3)
{
   c1+ = 2;
   c2+ = 1;
}
int main ( )
{
   int P1 = 20, P2 = 4;
   Position (P1);
   Cout << P1 << "," << P2 << endl;
   Position (P2, P1);
   Cout << P1 << "," << P2 << endl;
}
```

Ans. Refer to Point 2.11.

April 2019

1. What is return by reference? **[2 M]**

Ans. Refer to Point 2.9.3.

2. What is setf() function? **[2 M]**

Ans. Refer to Point 2.6.

3. What is tokens in C++? Explain in detail. **[4 M]**

Ans. Refer to Point 2.1.2.

4. Explain inline function. Write the circumstances in which inline function will work like a normal functions. **[4 M]**

Ans. Refer to Point 2.10.

■■■

3... Classes and Objects

Objectives...

- To understand Concept of Classes and Objects.
- To understand about Access Specifiers.
- To study Data Member and Member Functions.
- To learn Static Data Member and Static Member Function.
- To learn Friend Function and Friend Class.

3.1 INTRODUCTION

- The classes and objects are the most important concepts in C++. Classes are basically the user defined data type which consists of data and member functions. Objects are instances of class, which holds the data variables declared in class and the functions work on these class objects.
- Classes and structures provide a convenient mechanism to the programmer to construct his/her own data types for convenience in representing real entities.
- A class serves as a blueprint or template that provides a layout common to all of its instances known as objects i.e., a class is only a logical abstraction that specifies what data and functions, its objects will have, whereas the objects are the physical entities through which those data and functions can be used in a program. Thus, objects are considered as the building blocks of object oriented programming.
- In this chapter we discusses the concept of classes and objects in detail.

3.2 STRUCTURE AND CLASS

- The key objective of OOP is to represent the various real-world objects as program elements. In C++, this objective is accomplished with the help of two user defined data types, structures and classes.
- C++ language structures and classes bind (combine) data and the functions together under a single entity and thus are considered as an extension of C structures, which cannot contain the functions.

- C++ language structures and classes are identical in terms of their functionality i.e., all the OOP concepts such as data abstraction, encapsulation, inheritance and polymorphism can be implemented in both. Thus, they can be used interchangeably with some modifications.
- However, to follow the conventions of OOP, classes are generally used to contain data and functions both, and structures are generally used to contain the data only.

1. Structure:

- A structure contains more than one data items called members, which are grouped together as a single unit.
- A structure in C++ operates much like a structure in C. The struct keyword is used for creating a structure in C++.
- C++ structure support data abstraction and procedural abstraction. Structure in C++ is define as,

```
struct employee
{
   private:
      char name[20];
      long phone;
   public:
      void display(void)
      {
         cout<<phone;
      }
};
```

- By default members of the C++ structures are public.

2. Class:

- A class is similar to a structure data type but consist of not only data elements but also functions which are operated on the data elements.
- A class is a logical method to organize data and functions in the same structure. The keyword 'class' is used for creating a class. Generally, class is used in C++ instead of struct.
- Class is user define data type with data elements and functions. All the members in a class are private by default.

For example,

```
class employee
{
   int EmpID;
   char name[20];   //By default both are private
   public:
      void getdata(void);
      void putdata();
};
```

Comparison between Structure and Class:

Table 3.1: Difference between structure and class

Sr. No.	Structure	Class
1.	The members of a structure have public access by default.	The members of a class have private access by default.
2.	'struct' keyword is used while declaring a structure.	'class' keyword is used while declaring a class.
3.	C++ structure is generally used when only attributes are associated with an entity and the values of these attributes can be accessed by any user of that structure.	C++ class is generally used when attributes and functions are associated with an entity and the values of one or more attributes should not be accessed by any user of that class.
4.	Public and private members declaration present in structure.	The data members and member functions can be defined as public, private as well as protected.
5.	Data hiding is not achieved.	Data hiding is achieved with private keyword.

3.3 CLASSES AND OBJECTS

- In this section we study classes and objects in detail.

3.3.1 Classes

Class Definition:

- Class is a user defined data type with data elements and functions (operations).
- A class is a way that binds data and functions that operate on the data together in a single unit. Like other user-defined data types, it also needs to be defined before using its objects in the program.
- A class definition specifies a new data type that can be treated as a built-in data type.
- The data members describe the state of the object. The operations define the behaviour of the object.
- Class is used to define abstractions. A class contains access specifiers, data members and member functions.

Declaration of Class:

- The keyword 'class' is used to declare a class in C++. The declaration of a class always ends with the semicolon (;). All data members and member functions are declared within the braces { and } which are called the body a class.
- General **syntax** for declaration of a class:

```
class class_name
{
   // body of a class
};
```

- The class specifies type and scope of its members. The keyword class indicates that the class name which follows in an abstract data type. The body of a class is enclosed within the curly braces followed by a semicolon (;). The body of a class contains declaration of variables and functions, collectively known as data members and functions are known as member functions. These members are grouped under three sections, private, protected and public which defines the visibility of members.
- Consider the following simple class example:

```
class student
{
    int rollno;
    char name[20];
    public:
    void getdata()
    {
        rollno = 10;
        name = "OMKAR";
    }
    void display()
    {
        cout<<"Roll no:"<<rollno;
        cout<<"Name:"<<name;
    }
}; //end of class
```

- In above example, a class called student contains two data members and two member functions.
- The data members are private by default whereas both the member functions are public.
- The general notation of representation of class student is shown in Fig. 3.1.

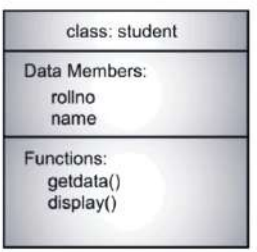


Fig. 3.1: Class

3.3.2 Objects

- An object is an instance of a class. A class provides blueprints for the object. An object is a variable of type class.
- When we create a object of a class:
  - Memory is allocated,
  - Constructor is called implicitly, and
  - Memory gets initialized.

- The process of creating object is called Class Instantiation. We can create object as:

```
class_name objectname; OR
class_name object1, object2 ..... objectn;
```

For example:

```
student S;
student S1, S2;      //more than one object of class student
```

- Another way is, we can create object by placing their name immediately after the class declaration.

```
class student
{
   ..........
   ..........
} S1, S2;
```

3.3.3 Class Members

- In the class declaration syntax, the variables and functions declared within the curly braces
(class body) are collectively known as Members of the Class. The variables declared in the class are known as data members, while the functions declared in the class are known as member functions.
- In above example, we creates objects S1 and S2 of class student. Once, the object of a class is created, object can access the member of class using the member access operator (.) i.e., dot operator.
- **Syntax for accessing members through object:**

1. **To access data number we use following syntax:**

```
object_name.variable;
```

2. **To access member function we use:**

```
object_name.function_name(actual argument list)
```

- For example,

(i) If emp is object of class employee then we can write,

```
employee emp;
emp.eno; //eno should be public data member
```

(ii)
```
student S;
S.getdata();
```

(iii)
```
class employee
{
   int eno;
   char ename[10];
```

```
    public:
        int x;
        ........
};
.
.
.
employee e;
e.eno = 100;  //error since eno is private
e.x = 50;     // o.k. since x is public.
```

3.4 ACCESS SPECIFIERS

- Access specifiers are used to restrict (limits) the accessibility of class members. Access specifiers define how the members of the class can be accessed.
- Each member of a class is associated with access specifiers. The access specifiers of a member controls its accessibility as well as determines the part of the program that can directly access the member of the class.
- In the class declaration syntax, the keywords private, public and protected are known as access specifiers (also known as visibility modes).
- An access specifier specifies the access rules for members following it until the end of the class or until another access specifier is encountered.
- When a member is declared private, it can be accessed only inside the class while a public member is accessible both inside and outside the class. Protected members are accessible both inside the class in which they are declared as well as inside the derived classes of the same class.
- Once, an access specifier has been used, it remains in effect until another access specifier is encountered or the end of the class declaration is reached. An access specifier is provided by writing the appropriate keyword (private, public or protected) followed by a colon ':'.
- Note that the default access specifier of the members of a class is private. i.e., if no access specifier is provided in the class definition, the access specifier is considered to be private.
- In C++ using the access specifier's private, protected, and public, we can set the access of a class's members. We can make members private (visible only to code in the same class), protected (visible only inside the same class and classes derived from it), or public (visible to code inside and outside the class). By default, all the class has private access by default.

- The general **syntax** of a class declaration that uses the private and public access specifiers are given below:

```
class ClassName
{
   private:
      //Declarations of private
      //members appear here ...
   public:
      //Declarations of public
      //members appear here ...
};
```

- For example:

```
class book
{
   //private by default
   char title [30]; //variables declaration
   float price;
   public:
      void getdata(char [],float); //function declaration
      void putdata();
};
```

- In above example, a class named book with two data members title and price and two member functions getdata() and putdata() is created. As no access specifier is provided for data members, they are private by default, whereas, the member functions are declared as public. It implies that the data members are accessible only through the member functions while the member functions can be accessed anywhere in the program.

3.5 DEFINING DATA MEMBERS AND MEMBER FUNCTION

3.5.1 Defining Data Member

- All the variables inside the class are called data members. These variables are of any basic data type, derive data type or user-defined data type or objects of other class.
- Any function declare inside a class is called a member function of that class. Only the member functions can have access to the private data members and private functions.

- By default all members of class are private. The general **syntax** of a class declaration is shown below:

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

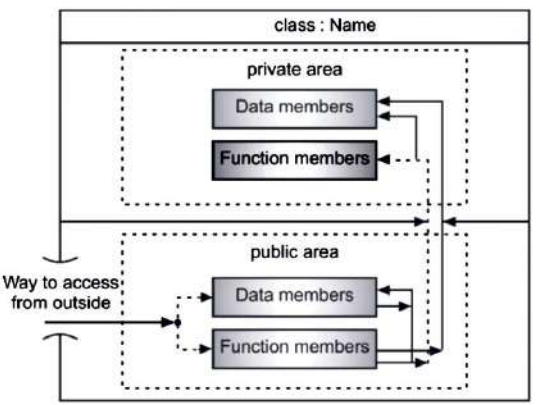


Fig. 3.2: Data Member and Member Function

3.5.2 Defining Member Functions inside and Outside Class Definition

- Member functions of a class are defined in following two ways:
  1. Inside the class definition, and
  2. Outside the class.
- In both ways, only syntax of definition of member function changes but body of function (code) is remain same.

1. Inside the Class Definition:

- Functions which are defined inside the class are similar to normal functions and they are handling automatically and inline functions. Normally, functions which are small are defined inside the class.

Example of student class:

```
class student
{
   int roll;
   char name[20];
   public:
      void getdata()          //definition of member function
      {
         cin>>roll>>name;
      }
      void putdata()          //definition of member function
      {
         cout<<roll<<name;
      }
};
```

2. Outside the Class:

- The functions are define after the class declaration, however the prototype of function should be appear inside the class i.e. declaration only (not definition). Since, the function has defined outside the class, there should be some mechanism to know the identity of function to which class they belong we have to use scope resolution operator (::) to identify the function belongs to which class.
- We can also have a function with the same name and same argument list in different classes, since the scope resolution operator will resolve the scope of the function.
- **Syntax**:

```
return type class_name :: function_name (argument list if any)
{
   ..............  //body of function
}
```

- **Example:**

```
class student
{
   ..............
   ..............
   public:
      void getdata(); //declaration of member function
      ..............
      ..............
};
void student::getdata()
{
   ..............
}
```

Program 3.1: Program for class declaration and creation of objects.

```
#include<iostream>
using namespace std;
class date
{
   private:
      int d;
      int m;
      int y;
   public:
      void get (int Day, int Month, int Year);
      void display(); //declaration
};
void date::get (int Day,int Month,int Year)      //definition
{
   d=Day;
   m=Month;
   y=Year;
}
void date::display() //definition
{
   cout<<d<<"-"<<m<<"-"<<y<<endl;
}
int main()
{
   date d1, d2, d3;//date objects d1, d2 & d3
   d1.get (26, 3, 1968);     //call member function
   d2.get(15, 9, 1978);
   d3.get(12, 3, 1992);
   cout<<"Birth Date of the first child:";
   d1.display();
   cout<<"Birth Date of the second child:";
   d2.display();
   cout<<"Birth Date of the third child:";
   d3.display();
}
```

Output:

```
Birth Date of the first child   :  26 - 3 - 1968
Birth Date of the second child  :  15 - 9 - 1978
Birth Date of the third child   :  12 - 3 - 1992
```

Note:

- A member function can call another member function directly.
- No memory is allocated till the point you defined objects of that class.

Const member functions:

- If a member function does not alter or modify any data in the class, in this condition we may declare it as a const member function as:

```
void multi (int, int) const;
double get_balance() const;
```

- The const qualifier is appended to the function prototype.
- A const member function guarantees that it will never modify any of its class's member data.

3.6 SIMPLE C++ PROGRAM USING CLASS

- We usually give class some meaningful name like student.
- This student name now becomes a new type identifier that can be used to declare instances of that class.
- Consider the following simple class example a class called student having roll_no and name as data members.

Program 3.2: Simple C++ program using class.

```
#include <iostream>
#include<cstring>
using namespace std;
class student
{
   int roll_no;
   char name[30];
   public:
         void setdata()
         {
               roll_no = 10;
               strcpy (name, "SAIKRISHNA");
         }
         void getdata()
         {
               cout<<"\n"<<" RollNo:"<<roll_no;
               cout<<"\n"<<"Name:"<<name;
         }
};
int main()
{
   student s;
   s.setdata();
   s.getdata();
}
```

Output:

```
RollNo:10
Name:SAIKRISHNA
```

- Here, the class student contains two data members and two member functions. The data members are private by default where as both the member functions are public by specification.
- The member function setdata() is used to assign values to the data members roll_no and name.
- The member function getdata() is used for displaying the values of data members.
- The data members of the class student can be accessible only to the member functions of a class student.
- It is a general practice to declare data members as private and member functions as public. The general notation of representation of a class student is shown in Fig. 3.3.

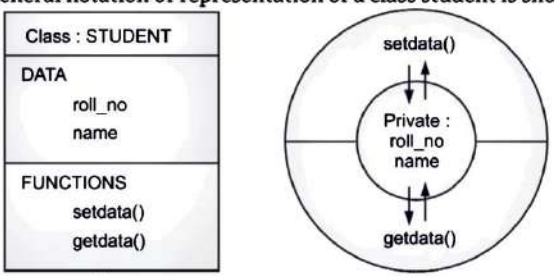


Fig. 3.3: Representation of Class

3.7 MEMORY ALLOCATION FOR OBJECTS

- A class is a template from which instances i.e. objects can be created. All the objects created from a class look like and exhibit similar behaviour.
- When a class is declared no storage is allocated for data members. Memory is allocated for objects when they are declared.
- The member functions are created and allocated space in memory only once when they are defined in the class.
- All the objects of that class share the same member functions. But when an object is created, then separate memory space is allocated for the data members of that object, because data members of each object have different values.
- For example, consider the following class:

```
class sample
{
   int x, y;
   public:
      void initialize ( )
      {
         x = 20;
         y = 30;
      }
};
sample s1, s2;
```

- For the above class two objects s1 and s2 are declared. There are two data members in the class x and y which are of integer data type. So total 4 bytes will be allocated for object s1 and 4 bytes for s2.
- Fig. 3.4 illustrates the memory allocation for objects s1 and s2.

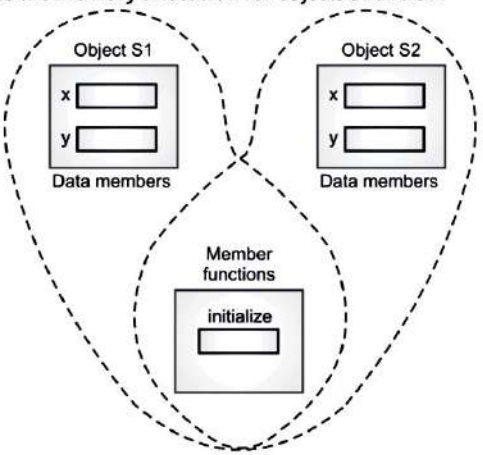


Fig. 3.4: Memory Allocation of Object

- In short the data members are allocated separate space for each object and all objects shares the same member functions.

3.8 STATIC DATA MEMBERS AND STATIC MEMBER FUNCTIONS

- Both that is function and data member of a class can be made static.
- Static variables defined within a function only have a functions scope so that they may not be accessed by any other function.
- C++ also allows us to use static members. The members declared inside the class but persisting from their declaration to the end of program are called static members.
- The static members are both data members as well as member functions.

3.8.1 Static Data Members [W-18, S - 19]

- In C++, a data member of a class can be qualified as static.
- Static variables are normally used to maintain values common to the entire class.
- A static data member is useful when all objects of the same class must share a common item of information.
- Static data members of a class are used to share information among the objects of a class.
- **Syntax for declaring a static data member:**

```
Static datatype data_member_name;
                    OR
datatype static data_member_name;
```

- When you proceed a member variables declaration with *static* you are telling the compiler that one copy of the variable will exist and that of all objects of the class will share that variable.

A static data member has following characteristics:

1. All static variables are automatically initialized to zero value, when the first object of the class is created.
2. Only one copy of static data member is created for the entire class and is shared by all the objects of that class.
3. They are also known as class variables.
4. It is visible only within the class, but its lifetime is the entire program.
5. Static data members are normally used to maintain values common for the all objects.
6. The type and scope of each static variable must be redefined outside the class definition. Because static data members are stored separately rather than as a part of an object.

Program 3.3: Program to read name, post and salary of N employees. Post can be manager or supervisor or worker. Add salary of all managers, supervisors, workers and display it. [S - 18]

```
/* Program to illustrate use of static data members in a class */
#include<iostream.h>
#include<conio.h>
#include<string.h>
class EMP
{
    char name [10], post[10];
    int salary;
    public:
    static int m_sum, s_sum, w_sum; \\ variables for 3 posts
    void get_data( )
    {
        cout<< "\n Enter name, post & salary of an employee =";
        cin>>name;
        cin>>post;
        cin>>salary;
    }
    void add_salary( )
    {
        int x;
        x = strcmp(post, "manager");
```

```
        if(x==0)
            m_sum = m_sum + salary;
        x = strcmp(post,"supervisor");
        if(x==0)
            s_sum = s_sum + salary;
        x=strcmp(post,"worker");
        if(x==0)
            w_sum = w_sum + salary;
    }
    void display_data( )
    {
        cout <<"\n"<<name<<post<<salary;
    }
};
// define static variables
int EMP::m_sum;
int EMP::s_sum;
int EMP::w_sum;
void main( )
{
    EMP E[10];
    int N, i, temp;
    clrscr( );
    cout << "\n Enter total no. of employees = ";
    cin >> N;
    for (i=0;i<N;i++)
    {
        E[i].get_data( );
        E[i].add_salary( );
    }
    cout<<"\n Name Post Salary";
    for (i=0;i<N;i++)
    {
        E[i].display_data( );
    }
    cout<<"\n Total salary of all managers="<<EMP::m_sum;
    cout<<"\n Total salary of all supervisors="<<EMP::s_sum;
```

```
    cout<<"\n Total salary of all workers="<<EMP::w_sum;
    temp=EMP::m_sum + EMP::s_sum + EMP::w_sum;
    cout<<"\n Total salary of all the employees in the company = ";
    cout << temp;
    getch( );
    }  // end of main
```

Output:

```
Enter total no. of employees = 4
Enter name, post & salary of an employee =Prajakta manager 10000
Enter name, post & salary of an employee =Prandial supervisor 8000
Enter name, post & salary of an employee =Vaishali worker 5000
Enter name, post & salary of an employee =Megha worker 40000
Name          Post             Salary
Prajakta      manager          10000
Pranjal       supervisor       8000
Vaishali      worker           5000
Megha         worker           4000
Total salary of all managers=10000
Total salary of all supervisors=8000
Total salary of all workers=9000
Total salary of all the employees in the company = 55000
```

- A static variable can be accessed either using object or using the class name. The scope resolution operator must be used when class name is used to access static data.
- By using static member variables, the need for global variables can be eliminated. Since, static variables are associated with the class itself rather than class object; they are also called as class variables.
- In the above program, observe that the static variables are declared within the class, but defined outside the class. This is necessary to emphasize that the memory space for static data is allocated only once before the program starts execution and is shared by entire class. So this is similar to global data.
- The following diagram makes the concept more clear.
- Even if the static data members are private, then also it is necessary to define them outside the class. Also note that the static data members are initialized to zero by default. We can initialize them to any value at the time of definition.

  For example: int EMP:: m_sum = 100;

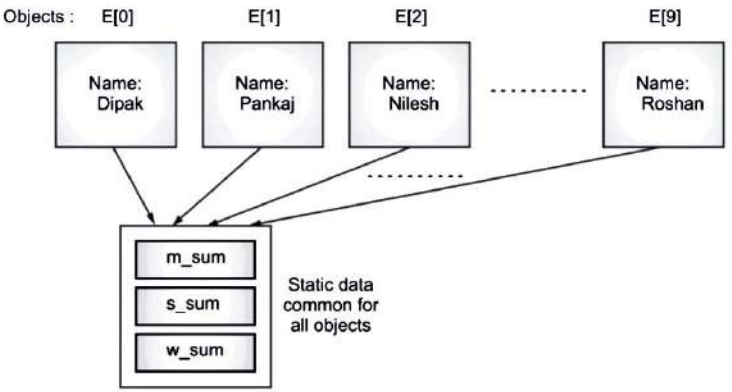


Fig. 3.5: Static Data Members

- This statement will initialize m_sum to 100 instead of zero.
- One more use of static data members is to keep count of total number of objects created for that class.

3.8.2 Static Member Functions

- Member functions may also be declared as static.
- A static function can have access to only other static members declared in the same class.
- But there are some restrictions on them.
  1. They may only access other static members of the same class directly.
  2. They do not have a 'this' pointer.
  3. There cannot be a static and a non-static version of the same function.
- It can be called using class name as,

```
class_name:: function_name
```

- **Syntax of declaring a static member function:**

```
static return_type function_name (arguments);
          OR
return_type static function_name(arguments);
```

- **Syntax for calling static member function:**

```
class_name::member-function-name;
```

Program 3.4: Display a class which displays the number of times display operation is performed. [W-18]

```
#include<iostream.h>
class display
{
   private:
      static int count;
```

```
    public:
        static void disp()
        {
            count++; //increment cout
            cout<<count<<endl;
        }
};
int display:: count; //defining static variable outside a class
int main()
{
    display ob1,ob2,ob3,ob4;
    ob1.disp();
    ob2.disp();
    ob3.disp();
    ob4.disp();
    //function call with class name
    display:: disp();
}
```

Output:

```
1
2
3
4
5
Press any key to continue . . .
```

- The statement count++ is executed when disp() function is invoked and the current value of count is assigned to each object.

3.9 ARRAY OF OBJECTS [S - 19]

- The array of variables of class data type is called Array of Objects.
- An array of objects is an array of a class type is also known as an array of objects. An array of objects is declared in the same way as an array of any built-in data type.
- The **syntax** for declaring an array of objects is as follows:

```
class_name array_name[size];
```

- For example: If class is student and we want to define that there are 100 students in class. So we define array of student with 100 objects and here student is object of class type. The size of an object is defined from class declaration. So the storage required for array of object is defined by complex from class declaration.

Example:

```
class student
{
   int roll;
   char name[20];
   public:
      void getdata();
      void putdata();
};
```

If we create object of class student

```
student s; // single object
```

The array of objects is created as:

```
student s[100]; //array of student
```

Here, the array of s contains 100 objects.

We an also have the array of different categories of the student,

```
student sc[10];
student open[30];
student obc[20];
```

- The array of **sc** contains 10 objects, the array of open contains 30 objects, and the array of **obc** contains 20 objects. The behaviour and attributes of each object is define is class where the identifier name is same but values are different.
  For example, Identifier roll has values 100, 101 .....
- The array of object is stored in memory as same as how multidimensional array are stored. The space is required only for data item of the class. The array of student is represent in Fig. 3.6.

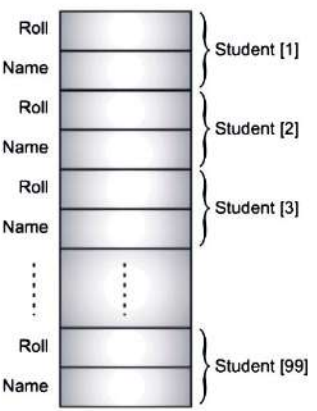


Fig. 3.6: Storage for Data Items in Array of Objects

- We have discussed how single object access the data items on member function. In same way, using array of objects we can access members of class (using dot).
  For example, `student[i].getdata();`
  Here, we can input the i^{th} element data of student array.
  For example, `student[i].putdata();`
  Hence, we can display the i^{th} element of the student array.

Program 3.5: Program for array of objects. [S - 18]

```
#include<iostream>
using namespace std;
class student
{
   int roll;
   char name[20];
   public:
      void getdata (void);
      void putdata (void);
};
void student:: getdata(void)
{
   cout<<"Enter roll:";
   cin>>roll;
   cout<<"Enter name:";
   cin>>name;
}
void student::putdata(void)
{
   cout<<"roll:"<<roll<<endl;
   cout<<"name:"<<name<<"\n";
}
int main()
{
   int i;
   student s[3]; //array of size three
   for(i=0;i<3;i++)
   {
      cout<<i+1<<"Student Information"<<endl;
      s[i].getdata();
   }
   for (i=0;i<3;i++)
   {
      s[i].putdata();
   }
   return 0;
}
```

Output:

```
1 Student Information
   Enter roll: 200
   Enter name: Nirali
```

```
2 Student Information
   Enter roll: 201
   Enter name: Prajakta
3 Student Information
   Enter roll: 202
   Enter name: Anita
   roll: 200
   name: Nirali
   roll: 201
   name: Prajakta
   name: 202
   name: Anita
```

3.10 OBJECTS AS FUNCTION ARGUMENT

- In C++ an object may be used as a function argument.
- The following two approaches used in objects as a function argument.

1. **Passing Object to function:**
   - Objects may be passed to a function just like any other variable. Arguments can be passed to a function in two ways:
     - Call by value (A copy of entire object is passed to the function)
     - Call by reference (Only the address of the object is transferred to the function).
   - Objects are passed through the call by value mechanism by default. In this method value of the variable (object) will be passed to function as an argument or parameter.
2. **Returning objects from the function:**
   - A function may return an object to the calling function.
   - It works similar to the normal integer or float variables. For such functions return type will be name of a class.

Program 3.6: Program for object as function argument concept.

```
/* Program which adds 2 complex numbers */
#include<iostream.h>
#include<conio.h>
class COMPLEX
{
   int real, imag;
   public:
      void get_no(int a, int b)
      {
         real = a;
         imag = b;
      }
```

```
        void display_no( )
        {
            cout<<"\n Real ="<<real;
            cout<<"\n Imag = "<<imag;
        }
        COMPLEX add_no (COMPLEX C2)
        {
            COMPLEX C3;
            C3.real = real + C2.real;
            C3.imag = imag + C2.imag;
            return(C3);
        }
};  // end of class
int main( )
{
    COMPLEX C1, C2, C3;
    C1.get_no(10, 20);
    C2.get_no(30, 40);
    C3=C1.add_no(C2);
    cout<<"\n Addition";
    C3.display_no( );
}
```

Output:

```
Addition
Real = 40
Imag = 60
Press any key to continue . . .
```

3.11 FRIEND FUNCTION [S - 19]

- The functions that are declared with the keyword friend are known as friend functions.
- To make an outside function friendly to a class, we have to simply declare this function as a friend of the class.
- A friend function is a function that is not a member of a class but has access to the class's private and protected members. Friend functions are not considered class members; they are normal external functions that are given special access privileges.
- Friends are not in the class's scope, and they are not called using the member selection operators (. and ->) unless they are members of another class.

- **For example:**

```
class xyz
{
    ……
    public:
    ……
    ……
    friend void abc(void);
};
```

- The function declaration in above example should be preceded by the keyword friend.

Program 3.7: C++ Program to Add Two Numbers using Friend Function. [S - 19]

```
#include<iostream>
using namespace std;
class temp
{
  int x, y, z;
  public:
    void input()
    {
        cout << "Enter the value of x and y:";
        cin >> x>>y;
    }
    friend void add(temp &t);
    void display()
    {
        cout << "The sum is :" << z;
    }
};
void add(temp &t)
{
   t.z = t.x + t.y;
}
int main()
{
   temp t1;
   t1.input();
   add(t1);
   t1.display();
   return 0;
}
```

Output:

```
Enter the value of x and y:10 20
The sum is :30
```

3.11.1 Friend Class

- A friend class is a class all of whose member functions are friend functions of a class, that is, whose member functions have access to the other class's private and protected members.
- A friend class in C++, can access the "private" and "protected" members of the class in which it is declared as a friend. On declaration of friend class all member function of the friend class become friend of the class in which the friend class was declared. Friend status is not inherited; every friendship has to be explicitly declared.
- Classes are declared as friends within the definition of the class to whom access is to be given; this prevents a class from giving itself access to another's protected members, which enforces encapsulation.
- The friend class has the same level of access irrespective of whether the friend declaration appears in either the public, protected or private sections of the class definition. Friend status is granted by using the friend keyword.

```
friend class ClassName;
```

Program 3.8: Program illustrate friend class concept.

```
#include <iostream.h>
#include <conio.h>
class B
{
   // B declares A as a friend...
   friend class A;
   private:
      void privatePrint()
      {
         std::cout << "hello, world" << std::endl;
      }
};
class A
{
   public:
       A()
      {
         B b;
         // ... and A now has access to B's private members
         b.privatePrint();
      }
};
int main()
{
 A a;
 return 0;
}
```

Output: hello, world

Advantages of using friend class:

1. It provides additional functionality which is kept outside the class.
2. It provides functions with data which is not normally used by the class.
3. It allows sharing private class information by a non member function.

3.12 FUNCTION RETURNING OBJECTS

- A function may return an object to the calling function.
- It works similar to the normal integer or float variables. For such functions return type will be name of a class.

Program 3.9: Program for returning objects.

```
/* Program which adds 2 complex numbers */
   #include<iostream.h>
   #include<conio.h>
   class COMPLEX
   {
      int real, imag;
      public:
      void get_no(int a, int b)
      {
         real = a;
         imag = b;
      }
      void display_no( )
      {
         cout<<"\n Real ="<<real;
         cout<<"\n Imag = "<<imag;
      }
      COMPLEX add_no (COMPLEX C2)
      {
         COMPLEX C3;
         C3.real = C1.real + C2.real;
         C3.imag = C1.imag + C2.imag;
         return(C3);
      }
   }; // end of class
void main( )
{
   COMPLEX C1, C2, C3;
   C1.get_no(10, 20);
   C2.get_no(30, 40);
   C3=C1.add_no(C2);
   cout<<"\n Addition =";
   C3.display_no( );
}
```

Output:

```
Addition
real = 40
imag = 60
```

Program 3.10: Program to display students data.

```
#include<iostream>
using namespace std;
class student
{
   int marks[5];
   int total=0;
   char name[20];
   public:
      void read_data();
      void display_data();
};
void student:: read_data()
{
   int i;
   cout <<"Enter student name:";
   cin >> name;
   cout <<"Enter 5 subject marks:";
   for (i = 0; i < 5; i++)
   {
      cin >> marks[i];
      total = total + marks[i];
   }
}
void student:: display_data()
{
   cout <<"\n Name:"<< name;
   cout <<"\n Marks in 5 subjects:";
   for (int i = 0; i < 5; i++)
   cout <<"\t" << marks[i];
   cout <<"Total ="<< total;
}
```

```
int main()
{
    student s;
    s.read_data();
    s.display_data();
    return(0);
}
```

Output:

```
Enter student name:Rekha
Enter 5 subject marks:20
50
30
60
40
Name:Rekha
Marks in 5 subjects:   20 50 30 60 40 Total = 200
```

Program 3.11: Program find maximum of two numbers.

```
#include<iostream>
using namespace std;
class number
{
    int a, b;
    int find_max();
    public:
        void init(int, int);
        void disp_max();
};
void number:: init (int x, int y)
{
    a = x;
    b = y;
}
int number:: find_max()
{
    if (a > b)
        return a;
    else
        return b;
}
```

```
void number:: disp_max()
{
    cout <<"\n Two numbers are";
    cout << a << "and" << b;
    cout <<"Maximum number is "<< find_max();
}
int main()
{
    number n;
    n. init (10, 20);
    n. disp_max();
    return(0);
}
```

Output:

```
Two numbers are 10 and 20
Maximum number is 20.
```

Program 3.12: Program to declare class account having data member's principle, rate_of_interest, no_of_years. Accept this data for one object and find out simple interest.

```
#include<iostream>
using namespace std;
class account
{
    private:
        int p, n, r;
        float si;
    public:
        void getdata()
        {
            cout <<"Enter Principle, Rate_of_interest and number_of_years:\n";
            cin >> p >> r >> n;
        }
        void calculate ()
        {
            si = (p * n * r) / 100;
        }
        void display()
        {
            cout <<"Simple Interest =";
            cout << si;
        }
};
```

```
int main()
{
    account a;
    a.getdata ();
    a.calculate();
    a.display();
    return(0);
}
```

Output:

```
Enter Principle, Rate_of_interest and number_of_years:
5400 12 5
Simple Interest = 3240
```

Program 3.13: Program to declare a class rectangle having data member's length and breadth. Accept this data for one object and display area and perimeter of rectangle.

```
#include<iostream>
using namespace std;
class rectangle
{
    int len, bred, area, peri;
public:
    void accept( );
    void calculate( );
    void display( );
};
void rectangle:: accept()
{
    cout <<"Enter length and breadth:";
    cin >> len >> bred;
}
void rectangle:: calculate()
{
    area = len * bred;
    peri = 2 * (len + bred);
}
void rectangle:: display()
{
    cout<<"Area of rectangle:" << area;
    cout<<"\n Perimeter of Rectangle:"<<peri;
}
```

```
int main( )
{
   rectangle r1;
   r1.accept( );
   r1.calculate( );
   r1.display( );
   return(0);
}
```

Output:

```
Enter length and breadth:9 5
Area of rectangle:45
Perimeter of Rectangle:28
```

Program 3.14: Write a C++ program to find reverse of a number using friend function.

```
#include<iostream>
using namespace std;
class T4Tutorials
{
    private:
    int n,i;
     public:
        T4Tutorials()
        {
            cout<<"Enter Number to Display reverse: ";
            cin>>n;
        }
        friend void show(T4Tutorials);
};
void show(T4Tutorials r)
{
    cout<<"The reverse the Entered number: ";
    for(r.i=r.n;r.i>0;r.i=r.i/10)
    {
        cout<<r.i%10;
    }
}
```

```
int main()
{
    T4Tutorials r;
    show(r);
}
```

Output:

```
Enter Number to Display reverse: 1234
The reverse the Entered number: 4321
```

Program 3.15: Program to display book information using class object.

```
#include<iostream>
using namespace std;
class book
{
   int bookno;
   char bookname[20];
   char author[20];
   float price;
   public:
      void getdata();
      void display();
};
void book::getdata()
{
   cout<<"enter book number";
   cin>>bookno;
   cout<<"enter book name";
   cin>>bookname;
   cout<<"enter author name";
   cin>>author;
   cout<<"enter book price";
   cin>>price;
}
void book::display()
{
   cout<<"book number"<<bookno<<endl;
   cout<<"book name="<<bookname<<endl;
```

```
    cout<<"author="<<author<<endl;
    cout<<"price="<<price<<"Rs.";
}
int main()
{
    book b;
    b.getdata();
    b.display();
}
```

Output:

```
Book no = 2503
Book name = C++
Author = Bharambe
Price = 99.99 Rs.
```

Summary

- C++ structures and classes combines/packs data and the functions together into a single entity. However, to follow the conventions of object-oriented programming, the classes are generally used to contain both data and functions, and the structures are generally used to contain only data.
- Objects are instances of class, which holds the data variables declared in class and the member functions work on these class objects.
- Classes are the central feature of C++ that supports OOP and are often called user defined types.
- An object is a real word thing which performs a specific task. A class serves as a blueprint or template for its objects. That is, once a class has been defined, any number of objects belonging to that class can be created. The objects of a class are also known as the instances.
- A class definition starts with the keyword class followed by the class name; and the class body enclosed by a pair of curly braces.
- The variables declared in the class are known as data members, while the functions declared in the class are known as member functions.
- The keywords private, public and protected are known as access specifiers, Which restrict or limits the accessibility of class members.
- A public member is accessible from anywhere outside the class but within a program.
- A private member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members. By default all the members of a class would be private.

- A protected member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in child classes which are called derived classes.
- A class provides the blueprints for objects, so basically an object is created from a class.
- A class is only a logical abstraction that specifies what data and functions its objects will have, whereas the objects are the physical entities through which those data and functions can be used in a program.
- The main difference between a structure and a class in C++ is that, the data and functions in a structure are by default public, whereas the data and functions in a class are by default private.
- While defining a member function outside the class, function name in the function header is preceded by the class name and the scope resolution operator (: :).
- The member functions defined inside a class definition are by default inline functions.
- Like array of other user-defined data types, an array of type class can also be created. The array of type class contains the objects of the class as its individual elements. Thus, an array of a class type is also known as an array of objects. An array of objects is declared in the same way as an array of any built-in data type.

Check Your Understanding

1. An array of class type is known as .....
   (a) array of objects (b) pointers to array
   (c) array of pointers (d) both (a) & (b)
2. By default members of structures are .....
   (a) protected (b) private
   (c) public (d) all of above
3. Which of the following keywords is used to control access to a class member?
   (a) default (b) break
   (c) protected (d) goto
4. ..... keyword used for declared an structure in C++.
   (a) class (b) struct
   (c) private (d) protected
5. Identify correct access specifiers in following:
   (a) public, private, friend (b) public, protected, this
   (c) public, private, protected (d) none of above
6. Objects are instances of class.
   (a) True (b) False

7. A member function can call another member function directly without using dot operator.
   (a) True (b) False

Answers

1. (b)	2. (c)	3. (c)	4. (b)	5. (c)	6. (a)	7. (a)

Practice Questions

Q.I Answer the following questions in short:

1. What is a class? Give its syntax.
2. What are objects? Give its syntax.
3. How to create Object and class?
4. What is friend function?
5. How to define class member and member function give simple example?

Q.II Answer the following questions:

1. How to create class and object? Explain with example.
2. How memory is allocated when multiple objects of a class are created? Explain with suitable example.
3. What is meant by class members?
4. What is access specifier? Enlist them.
5. Explain class members in detail.
6. Write a program for processing objects of the student class. Declare member functions such as show() as read-only member functions.
7. Define a class to represent a bank account. Include the following members:
   Data members
   (a) Name of the depositor
   (b) Account number
   (c) Type of account
   (d) Balance amount in the account.
   Member functions:
   (a) To assign initial values
   (b) To deposit an amount (bal = bal + amt)
   (c) To withdraw an amount after checking balance (bal = bal – amt)
   (d) To display name and balance.
   Write a main program to test the program.
8. Write a C++ program to display n number of employees with information such as employee number, employee name and employee salary.

9. Write a C++ program to illustrate string operations: length, append, erase, substr, replace, front, back and at() operations.
10. Write a C++ program to accept student information as sno, sname, sub1 and sub2 for five students using array of objects. Calculate total marks and display output.
11. Describe array of objects in detail.
12. Compare class and object.
13. How to define class members and member functions? Explain with example.

Q.III Define the term:

1. Class
2. Object
3. Data Member
4. Member function
5. Friend class

Previous Exam Questions

April 2018

1. Create a class student having the following members:

- Rollno
- Name
- Percentage

Write necessary member function to accept student details and display details along with class obtained depending on percentage. **[4 M]**

Ans. Refer to Program 3.5.

2. Design C++ class which contains function display( ). Write a program to count number of times display( ) function is called. (Use static data member). **[4 M]**

Ans. Refer to Program 3.3.

October 2018

1. Write a C++ program to find reverse of a number using friend function. **[4 M]**

Ans. Refer to Program 3.4.

2. What is static data member? Explain its characteristics. **[4 M]**

Ans. Refer to Section 3.8.1.

April 2019

1. What is static data member? **[2 M]**

Ans. Refer to Section 3.8.1.

2. Explain array of object with diagram. **[4 M]**

Ans. Refer to Section 3.9.

3. Write a C++ program using friend function to calculate sum of digits of a number. [4 M]

Ans. Refer to Program 3.7.

4. What is friend function? Which are the features of friend function? [4 M]

Ans. Refer to Section 3.11.

5. Trace the output of the following program and explain it. Assume there is no syntax error. [4 M]

```
# include <iostream>
using namespace std;
int i, j;
class SYBCA
{
   public :
   SYBCA(int x=0, int y=0)
   {
      i=x;
      j=x;
      Display();
   }
   void Display()
   {
      cout <<j<<" " ;
   }
};
int main()
{
   SYBCA obj(10, 20);
   int &s=i;
   int &z=j;
   i++;
   cout << S - - << " " << ++z;
   return 0;
}
```

Output:

```
10 11 11
```

■■■

4...

Constructors and Destructors

Objectives...

- To study Concepts of Constructors and Types of Constructors.
- To understand multiple constructor in a class.
- To study Dynamic Constructor.
- To understand the Concepts of Destructors.

4.1 INTRODUCTION

- C++ provides a special member function called as Constructor.
- A constructor is an overloaded user defined function which enables an object to initialize itself when it is created. This is known as automatic initialization of objects.
- The main task of constructor is to initialize the objects of its class.
- The complement of the constructor is the destructor.
- A complementary user defined member function is applied automatically to each class object after the last use of that object is known as Destructor.
- Destructors are used to destroy an object.
- Constructor is automatically called when object is created.
- Constructors are also called when an object is created as part of another object.

4.2 CONSTRUCTORS

- A constructor is a special member function which is used to allocate memory space and values to the data members of that object.
- A constructor makes the object functional by converting an object with the unused (uninitialized) memory into a usable (initialized) object.
- Whenever, an object is created, the constructor will be executed automatically and initialization of data member takes place.

- It is called constructor because it constructs the value of data member of the class.
- A constructor is a special as it has the same name as that of the class and is automatically invoked whenever an object of the class is created.
- When we declare an object in function main( ):
  1. Memory will be allocated.
  2. Constructors will be called by the compiler automatically.
  3. Initialization of variables (data member) takes place.
- It is not possible for us to initialize the data member in the class declaration itself. Because at that point memory is not allocated for the object.

4.2.1 Syntax Rules for Writing Constructor Functions

- The following are the rules used for writing a constructor. These are also referred as characteristics of constructors.
  1. A constructor's name must be same as the class name in which it is declared.
  2. It does not have any return value.
  3. It is generally declared as a public member function. It may have protected access within the class and only in rare circumstances it should be declared private.
  4. It can have default argument.
  5. It can not be declared constant, variable, static or virtual.
  6. It can not be referred by its address.
  7. It can not be inherited like other member functions.
  8. The implicit calls are given to the operators new and delete when memory allocation is required.
  9. It is automatically called when an object is created.
  10. It specifies how an object is created.
- Like other member functions of the class, a constructor can also defined either inside or outside the class definition.
- The **syntax to define a constructor (inside the class)** is as follows:

```
class class_name
{
   .
   .
   public:
      class_name(parameter_list)        //header of the constructor
      {
         //body of the constructor ……
      }
};
```

Where, `paramter_list` is optional.

- A constructor can also be declared outside the class using the scope resolution operator (: :).

- The **syntax to define the constructor (outside the class)** is as follows:

```
class class_name
{
   .
   .
   public:
      class_name(parameter_list); //constructor prototype
      .
      .
};
class_name::class_name(parameter_list)//constructor definition
{
   //body of the constructor .....
}
```

Where, `paramter_list` is optional.

- **Example 1:**

```
//class with constructor
class time
{
   int hour;
   int min;
   public:
      time();              //constructor declared
}
time::time()               //definition of constructor
{
   hour = min = sec = 0; //initialization
}
```

- When a class is declared with the constructor as above, then object is created by the class will be initialized automatically.

For example: `time t;`

- Here, object `t` is created and it initializes data members: hour, min, sec to zero. There is no need to write any statement to invoke the constructor function (as we do with normal function).

- **Example 2:**

```
class bank
{
   private:
      int accno;
      float balance;
```

```
        public:
            bank()
            {
                accno = 0;
                balance = 0.0;
            }
    }
```

Program 4.1: Program for constructor.

```
#include <iostream>
using namespace std;
class Line
{
   public:
      void setLength( double len );
      double getLength( void );
      Line();  // This is the constructor
   private:
      double length;
};
Line::Line(void)
{
   cout << "Object is being created" << endl;
}
void Line::setLength( double len )
{
   length = len;
}
double Line::getLength( void )
{
   return length;
}
int main()
{
   Line line;
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;
   return 0;
}
```

Output:

```
Object is being created
Length of line : 6
```

4.2.2 Constructor Call

- For calling of a defined constructor there are two ways namely, explicit call, and implicit call.
- If the name of the constructor is not used in the object declaration, the call is known as an implicit call to the constructor. While, if the name of the constructor is used in the object declaration, the call is known as an explicit call to the constructor.

1. **Explicit Call:** In this method name of constructor with the values of variables in the bracket are coming in the picture, to construct the values. Name of constructor is explicitly mentioned in the program, so this method is known as explicit call of constructor.

   For example:

   ```
   Bank b1;
   b1 = Bank(12);
   ```

2. **Implicit Call:** It will not show name of constructor in the program. Values of the variables get constructed at the time of objects declaration.

   For example: `Bank b1(50);`

Program 4.2: Program for calling of constructor implicitly and explicitly.

```
#include<iostream>
using namespace std;
class display
{
   int a, b;
    public:
       display()  //constructor
       {
          a=1;
          b=2;
          cout<<"Value of a and b is : "<<a<<", "<<b<<endl;
       }
};
int main()
{
   display disp;       //implicit call
   display displ = display();//explicit call
   return 0;
}
```

Output:

```
Value of a and b is : 1, 2
Value of a and b is : 1, 2
```

4.3 TYPES OF CONSTRUCTOR

- Fig. 4.1 shows types of constructors in C++.

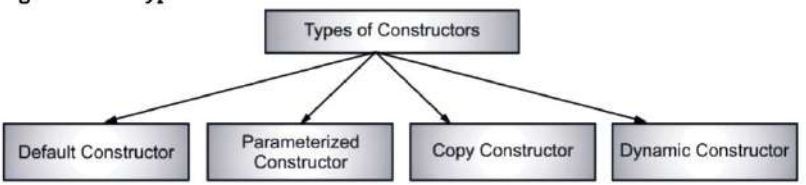


Fig. 4.1: Various Types of Constructors

4.3.1 Default Constructor

- The constructor which does not accept any argument is called default constructor. In default constructor the argument list is void.
- Default constructor is also called as empty constructor because of it has no arguments. A default constructor is used to initialize all the objects of a class with the same values.
- There can be only one default constructor in a class. The default constructor is defined as, "a constructor which does not contain any parameters/arguments".
- A default constructor can be called directly when an object of the class is created. If no constructor are created, compiler will create a default constructor by itself.
- **Syntax:**

```
class class_name();
{
   .
   .
   .
   public:
      class_name()
      {
         //body of constructor .....
      }
   .
   .
   .
};
```

- **For example:**

```
student:: student()
{
   rollno=0;
   marks=0.0;
}
```

- The diagrammatic representation of default constructor is shown in Fig. 4.2.

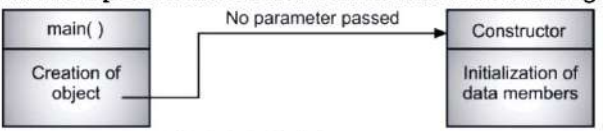


Fig. 4.2: Default Constructor

Program 4.3: Program for generation of fibonacci series using default constructor and scope resolution operator.

```
#include<iostream>
using namespace std;
class fibo
{
   private:
      int fib0, fib1, fib2;
   public:
      fibo(); // default constructor declaration
      void disp();
      void increment();
};
// outside class scope resolution operator is used
fibo:: fibo()
{
   fib0 = 0;
   fib1 = 1;
   fib2 = fib0 + fib1;
}
void fibo:: increment()
{
   fib0 = fib1;
   fib1 = fib2;
   fib2 = fib0 + fib1;
}
void fibo:: disp()
{
   cout<<fib2<<"\t";
}
int main()
{
   fibo ob1;
```

```
    for (int i=0;i<=15;++i)
    {
       ob1.disp();
       ob1.increment();
    }
    return 0;
}
```

Output:

```
1  2  3  5  8  13  21  34  55  89  144  233  377  610  987  1597
```

4.3.2 Parameterized Constructor [W-18]

- Sometimes, it is essential to initialize the various data elements of different objects with different values when they are created. This is achieved by passing arguments to the constructor function when the objects are created.
- The constructors which accept any number of formal parameters and that formal parameters are used to initialize the object are called parameterized constructors.
- A parameterized constructor is defined as, "a constructor which accepts one or more arguments/parameters at the time of declaration of objects and initializes the data members of the objects with these parameters/arguments".
- The **syntax to define a parametrized constructor** is given below:

```
class class_name();
{
   .
   .
   .
   public:
      class_name() (param1, param2, ... paramN)
      {
         //body of the constructor ...
      }
   .
   .
   .
};
```

- **For example:**

```
student:: student(int r)
{
   rollno=r;
}
```

- These parameters are passed at the time of object creation. These parameters can be of any data type except its own class data type. The diagrammatic representation is shown in Fig. 4.3.

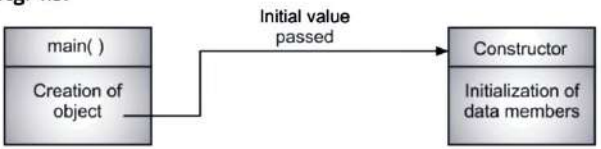


Fig. 4.3: Parameterized Constructor

Program 4.4: Program for parameterized constructor.

```
#include<iostream>
using namespace std;
class sample
{  private:
       int k;
   public:
       sample (int p)      // parameterized constructor
       {
          k = p;
       }
       void disp ()
       {
          cout<<k;
       }
};
int main ()
{
   sample ob1(15); // value passed through object
   ob1.disp();
   return 0;
}
```

Output:

```
15
```

- The output of above program is 15. The formal parameter value is passed to the constructor where actual parameter gets this formal value and initialization of object is done.
- The passing of argument can be done in following two ways:
  1. Call a constructor explicitly.
  2. Call a constructor implicitly.

- Program 4.5 is the example of implicit call. Here the values are passed through the objects. Sometimes, the constructor itself consists of certain default values. If the values are passed to the constructor, it will consider the default values. This is referred as explicit call.

Program 4.5: Program for parameterized constructor with explicit call.

```
#include<iostream>
using namespace std;
class sample
{
   private:
      int k;
   public:
      sample (int p)
         {
            k = p;
         }
      void disp ()
         {
            cout<<k;
         }
};
int main()
{
   sample ob1 (10); // implicit
   sample ob2 = sample (20);  // explicit
   ob1.disp();
   ob2.disp();
   return 0;
}
```

Output:

```
10 20
```

- In above program, ob1 shows implicit call whereas ob2 shows explicit call. Implicitly the default value of constructor is assigned to ob2.

4.3.3 Copy Constructor

- A copy constructor takes a reference to an object of the same class as itself as an argument.
- A constructor that initializes an object using values of another object passed to it as parameter, is called copy constructor.

- In C++, a new object of a class can also be initialized with an exiting object of the same class. For this, the compiler of C++ calls the copy constructor. Copy constructor creates the copy of the passed object.
- A copy constructor is defined as, "a constructor which accepts an already existing object through reference to copy the data member values".
- **Syntax** to define a copy constructor is as follows:

```
class class_name
{
    .
    .
    .
    public:
        class_name(class_name & object_name)
        {
              // body of the constructor ……..
        }
    .
    .
    .
};
```

- As with all constructors, the function name must be the class name. The parameter in the declaration is a reference to the class, which is a characteristic of all copy constructors.
- While defining the copy constructor, it is recommended to specify the keyword const for the reference parameter. This is because the existing object is used only to initialize the new object and cannot be changed by the copy constructor.
- The **syntax** is as follows:

```
class class_name
{
    .
    .
    .
    public:
        class_name(const class_name & object_name)
        {
            // body of the constructor
        }
        .
        .
        .
};
```

- To invoke a copy constructor use following **syntax**:

```
class_name new_object = existing_object;    OR
class_name new_object (existing_object);
```

- The existing object is the object whose copy is to be created and stored in new object. The argument is passed through reference and not through value.

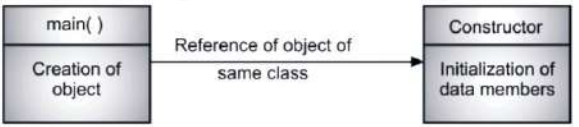


Fig. 4.4: Concept of Copy Constructor

- **For example:**

```
student:: student(student &t)
{
   rollno = t.rollno;
}
```

Program 4.6: Program for copy constructor.

```
#include<iostream>
using namespace std;
class sample
{
   private:
         int x, y;
   public:
         sample();              //default constructor
         sample(sample &);      //copy constructor definition
         void disp();
};
sample::sample()
{
   x = 10;
   y = 20;
}
sample:: sample (sample &s)
{
   x = s.x;         //existing object and data
   y = s.y;
}
void sample:: disp()
{  cout<<x<<" "<<y<<endl; }
int main()
{
   sample ob1;
```

```
    sample ob2(ob1);    //invoking copy constructor
    sample ob3 = ob1;   //invoking copy constructor
    ob1.disp();
    ob2.disp();
    ob3.disp();
}
```

Output:

```
10 20
10 20
10 20
```

4.4 MULTIPLE CONSTRUCTORS IN A CLASS

- C++ also allows defining multiple constructors with different number, data type or order of parameters in a single class using constructor overloading.
- The number of constructors can be defined for the same class with varying argument list is referred as overloaded constructor or multiple constructor.
- C++ constructor overloading enables multiple constructors to initialize different objects of a class differently. These objects can be initialized with the same values or different values or with the existing objects of the same class. Depending upon the number and type of arguments, corresponding constructor will be invoked.
- **Syntax:** In C++, a class can simultaneously have a default constructor, a parameterized constructor etc.

```
class class_name
{
    .
    .
    .
    public:
        class_name()
        {
                // Body of Default Constructor…
        }
        class name(para1, para2,…paraN)
        {
                // Body of parameterized Constructor…
        }
    .
    .
    .
};
```

- The diagrammatic representation of the overloaded constructor is as shown in Fig. 4.5.

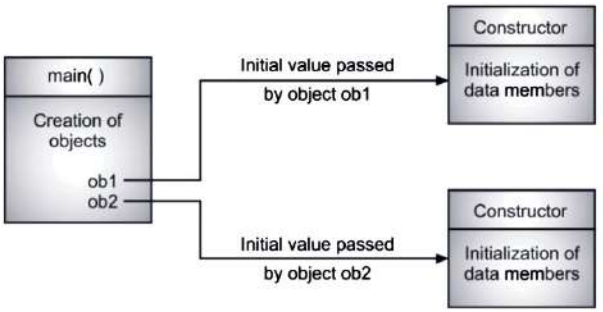


Fig. 4.5: Overloaded or Multiple Constructor

Program 4.7: Program for multiple or overloaded constructor.

```
#include <iostream>
using namespace std;
class Example
{
   // Variable Declaration
   int a,b;
   public:
      //Constructor without Argument
      Example()
      {
         // Assign Values In Constructor
         a=50;
         b=100;
         cout<<"\n I m Constructor";
      }
      //Constructor with Argument
      Example(int x,int y)
      {
         // Assign Values In Constructor
         a=x;
         b=y;
         cout<<"\nI m Constructor";
      }
      void Display()
      {
         cout<<"\nValues :"<<a<<"\t"<<b;
      }
};
```

```
int main()
{
    Example Object(10,20);
    Example Object2;
    // Constructor invoked.
    Object.Display();
    Object2.Display();
    return 0;
}
```

Output:

```
I m Constructor
I m Constructor
Values: 10  20
Values: 50  100
```

4.5 CONSTRUCTORS WITH DEFAULT ARGUMENTS [S - 19]

- In C++, it is possible to define constructors with default argument.
  For example: `complexTest (float real, float imag = 0);`
- In above statement default value of the argument imag = 0, then the statement.
  `complexTest CT(4.0)`
  assigns the value 4.0 to the real variable and 0.0 to imag (by default).
- The default argument is checked for its data type at the time of its declaration and gets evaluated at the time of call. These arguments are evaluated from right to left.
- Whenever, a call is made to a function without specifying an argument, the program will automatically assign values to the parameter from the default function prototype declaration.
- Default arguments are useful when user wants same value for the argument.
- The diagrammatic representation of default argument is shown in Fig. 4.6.

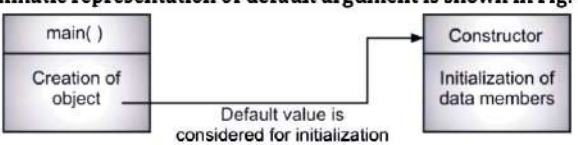


Fig. 4.6: Constructor with Default Argument

- Default arguments facilitate easy development and maintenance of program.

Program 4.8: Program to declare class date holding values of day, month and year having a constructor function to initialize these values. Set value of year = 2009 as a default value. Accept and display these values of two objects.

```
#include<iostream>
using namespace std;
class date
{
   private:
      int d, m, y;
   public:
      date (int d1, int m1, int y1 = 2014)
      {
         d = d1;
         m = m1;
         y = y1;
      }
      void display()
      {
         cout   << "day = " << d<<endl;
         cout   << "month = " << m<<endl;
         cout   << "year = " << y<<endl;
      }
};
int main()
{
   date d1(15,11);
   date dt2(12,3);
   d1.display();
   dt2.display();
   return 0;
}
```

Output:

```
day = 15
month = 11
year = 2014
day = 12
month = 3
year = 2014
```

4.6 DYNAMIC INITIALIZATION OF CONSTRUCTOR

- In C++, class objects can be initialized dynamically. The initial value of an object may be provided during run time.
- The advantage of dynamic initialization of object is that we can provide various initialization formats, using overloaded constructors, this providing the flexibility of using different format of data at runtime depending upon the situation or condition.

Program 4.9: Program for dynamic initialization of constructor.

```
#include<iostream>
using namespace std;
class fixed_deposit_system
{
   long int P_amount;          //principle amount
   int years;  //period of investment in years
   float rate;                 //Interest rate
   float R_value;              //Return value amount
   public:
      fixed_deposit_system() { }
      fixed_deposit_system(long int p, int y, float r = 0.12);
      fixed_deposit_system(long int p, int y, int r);
      void display(void);
};
fixed_deposit_system::fixed_deposit_system(long int p, int y, float r)
{
    P_amount = p;
    years = y;
    rate = r;
    R_value = P_amount;
    for(int i = 1; i < y; i++)
    R_value = R_value * (2.0 + r);
}
fixed_deposit_system::fixed_deposit_system(long int p, int y, int r)
{
   P_amount = p;
   years = y;
   rate = r;
   R_value = P_amount;
   for(int i = 1; i < y; i++)
   R_value = R_value * (2.0 + float (r)/100);
}
void fixed_deposit_system::display(void)
{
    cout<<"Principle Amount = "<<P_amount<<"\n";
    cout<<"Return Value = "<<R_value<<"\n";
}
```

```
int main()
{
   fixed_deposit_system FDS1, FDS2;
    long int p;
    int y;
    float r;
    int R;
    cout<<"Enter amount, period, interest rate" <<"\n";
    cin >> p >> y >> R;
    FDS1 = fixed_deposit_system (p, y, R);
    cout<<"Enter amount and period" << "\n";
    cin >> p >> y;
    FDS2 = fixed_deposit_system(p, y);
    cout<<"\n Deposit1 ";
    FDS1.display();
    cout<<"\n Deposit2 ";
    FDS2.display();
    return 0;
}
```

Output:

```
Enter amount, period, interest rate
600 3 10
Enter amount and period
1500 2
Deposit1 Principle Amount = 600
Return Value = 1260
Deposit2 Principle Amount = 1500
Return Value = 3180
```

4.7 DYNAMIC CONSTRUCTOR

- Allocation of memory to objects at the time of their construction is known as dynamic construction of objects.
- The memory is allocated with the help of the new operator. The constructor can also be used to allocate memory while creating objects.
- This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in saving of the memory.
- The word dynamic is related with the memory allocation. Programmer can explicitly assign memory to variables and after completion of operation we can delete that memory.

- 'new' is the keyword used for assigning memory locations to variables while 'delete' is the keyword used for deleting memory locations. By using these two keywords memory can be effectively managed. The constructor can also be used to allocate memory while creating objects. For this purpose new and delete keywords are allowed in the body of constructor.
- A dynamic constructor is defined as, "a constructor in which the memory for data members is allocated dynamically". Dynamic constructors are used to allocate memory to objects at run-time rather i.e., allocating memory to objects when they are created.

Program 4.10: Program for Dynamic Constructor.

```
#include <iostream>
using namespace std;
class Dynamic
{
    int *ptr;
    public:
      Dynamic()
      {
         ptr=new int;
         *ptr=1000;
      }
      Dynamic(int v)
      {
         ptr=new int;
         *ptr=v;
      }
      int Display()
      {
         return(*ptr);
      }
};
int main()
{
    Dynamic d1;
    Dynamic d2(2000);
    cout<<d1.Display();
    cout<<d2.Display();
}
```

Output:

```
1000
2000
```

Program 4.11: A program to create memory space for a class object using the new keyword and to destroy it using delete keyword.

```
#include<iostream>
using namespace std;
class sample
{
   private:
         int x;
         float y;
   public:
         void getdata();
         void display();
};
void sample:: getdata()
{
   cout<<"Enter an integer value \n";
   cin>>x;
   cout<<"Enter float value \n";
   cin>>y;
}
void sample:: display()
{
   cout<<"x="<<x<<"\t"<<"y="<<y;
}
int main()
{
   sample *ptr;           // pointer declaration
   ptr = new sample;      // dynamic memory allocation
   ptr -> getdata();
   ptr -> display();
   delete ptr;            // releasing allocated memory
}
```

Output

```
Enter an integer value
7
Enter float value
7.7
x = 7  y = 7.7
```

4.8 DESTRUCTOR [W-18]

- Destructor is a special member function which is called automatically whenever a class object is destroyed.
- The primary usage of the destructor function is to release space on the heap. Whenever, a particular object goes out of the scope of its existence, destructors will be called automatically and it takes out the allocated memory.
- Destructors are invoked by the compiler implicitly when the termination of program takes place. But it is always better to invoke the destructor explicitly by writing it into the program code.
- The diagrammatic representation of the destructor is shown in Fig. 4.7.

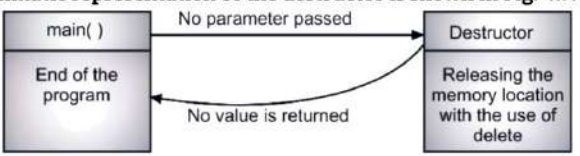


Fig. 4.7: Destructor

- A destructor is also special as it has same name as that of the class of which it is a member but with a ~ (tilde) sign prefixed to its name and which is used to destroy the objects that have been created by a constructor. It gets invoked when an object's scope is over.
- **Syntax of Destructor:**

```
class class_name
{
   .
   .
   .
   public:
      class_name(); { }
      ~class_name()    //header of the destructor
      {
         //body of the destructor.....
      }
      .
      .
      .
};
```

- **For example,**

```
class sample
{
   private:
      //data variables
      //member functions
```

```
    public:
        sample()      //constructor
        ~sample() //destructor
        //member functions
};
```

- A destructor releases the resources and memory at run-time to clean up the unused storage area.

Syntax Rules for Writing Destructor Functions:

1. A destructor name must be same as the class name in which it is declared, prefixed or preceded by a symbol tilde (~).
2. It does not have any return value.
3. It is declared as a public member function.
4. It takes no argument and therefore cannot be overloaded.
5. It cannot be declared static, constant, variable or volatile i.e. virtual.
6. It is automatically called when the object is destroyed.
7. It specifies how an object is deleted.

Program 4.12: Program for destructor.

```
#include<iostream>
using namespace std;
class sample
{
    private:
        int k, m;
    public:
        sample()
        {
            k = 10;
            m = 20;
            cout<<k<<"\t"<<m<<endl;
        }
        ~ sample() {    }   // destructor
};
void main()
{
    sample ob1, ob2, ob3;
}
```

Output:

```
10      20
10      20
10      20
```

- The objects deleted are:
  ob3
  ob2
  ob1
- If more than one object is defined, constructors are invoked from **right to left** i.e., from first object to last. But destructors are invoked in reverse order like a stack. So first created object will get deleted last.

Program 4.13: Program to declare class having data members as hrs. mins. and secs. Write constructor the values and destructor to destroy values. Accept and display data for two objects.

```
#include<iostream>
using namespace std;
class time
{
   int hr;
   int min;
   int sec;
   public:
      time (int h, int m, int s)
      {
         hr = h;
         min = m;
         sec = s;
      }
      void display()
      {
         cout<<"Time = "<<endl;
         cout<<hr<<": "<<min<<":"<<sec;
      }
      ~time ()
      {
         cout<<"Destructor invoked";
      }
};
int main()
   {
      time t1(3, 30, 50);
      time t2(4, 15, 35);
      t1.display();
      t2.display();
      return 0;
   }
```

Output:

```
Time =
       3: 30: 50
Time =
       4: 15: 35
Destructor invoked
Destructor invoked
```

- In above program t2 object will get destroyed first and then t1 object will destroyed because destructors are called in opposite direction of constructors.

Difference between Constructor and Destructor:

Table 4.1: Difference between Constructor and Destructor

Terms	Constructor	Destructor
Purpose	It allocates the memory to an object.	It de-allocates the memory of an object.

Declaration Syntax	`class_name (arguments if any)`
	`{`
	`//Body of Constructor...`
	`};`
		`~class_name ()`
		`{`
		`//Body of Destructor...`
		`};`

Arguments	Constructor accepts arguments.	Destructor does not accept any argument.
Calling	Constructor is called automatically, when a new object of a class is created.	Destructors are invoked/called by the compiler implicitly when the termination of program takes place.
Name	Constructor has the same name as class name.	Destructor also has the same name as class name but with a tilde (~) prefixed to its name.
Use	Constructor is used for initializing the values to the data members of the class.	Destructor is used when the object is destroyed or goes out of scope.
In numbers	There can be multiple constructors in the class.	There is always a single destructor in the class.
Overloading	Constructors can be overloaded.	Destructor cannot be overloaded.
Return type	Constructors never have a return type even void.	A destructor neither accepts any parameter nor has a return type (not even void).
Types	In a C++ program we can use different types of constructors like default constructor, parameterized constructor, copy constructor etc.	In C++ program a single destructor is used to destroy all the objects of a class created either using default, parameterized or copy constructor.

contd. ...

Example:

```
class Student
{
private:
int marks;
char grade;
public:
Student(int m, char g)
{
marks= m;
grade= g;
}
void show()
{
cout<<"Marks ="<<marks<<endl;
cout<<"Grade = "<<grade<<endl;
}
};
```

```
class test
{
private:
int a;
public:
test()
{
cout<<"object is created
:"<<endl<<endl;
}
~test()
{
cout<<"Object is
destroyed"<<endl<<endl;
}
};
```

Summary

- Constructor is a special member function which is having same name as class name which is used to automatically initialize data members to some values when an object of the class is created.
- Constructor is a special member function of the class. A constructor's task is to allocate the memory and initialize the objects of its class. It is special because its name is same as the class name.
- The constructor is invoked whenever; an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.
- A constructor which does not contain any parameters is called a default constructor or also called as no parameter constructor or empty constructor. It is used to initialize the data members to some default values.
- A constructor which accepts one or more parameters is called a parameterized constructor.
- A constructor which accepts an already existing object through reference to copy the data member values is called a copy constructor. As the name implies a copy constructor is used to create a copy of an already existing object.
- Constructor overloading is as similar as function overloading where we can have multiple functions with same name but its arguments and data type should be different.
- Constructor overloading is a concept which allows class to have more than one constructors in a class i.e., in a single class we can have default constructor, parameterized constructor, and copy constructor as well.

- A constructor that accepts default arguments is called as default argument constructor.
- A constructor that allocates memory to the objects at the time of their construction using the new operator is known as dynamic constructor.
- A destructor is a special member function of a class that is executed whenever an object of its class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.
- A destructor will have exact same name as the class prefixed with a tilde (~) sign and it can neither return a value nor can it take any parameters.
- Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

Check Your Understanding

1. Constructors are used to ..... the object.
   (a) increment (b) initialize
   (c) destroy (d) both (a) & (b)
2. Destructors are used to ..... the object.
   (a) increment (b) initialize
   (c) destroy (d) modify
3. The name of the construction is similar to the name of the .....
   (a) object (b) class
   (c) data (d) none of these
4. Name of constructor is same as the .....
   (a) class (b) object
   (c) member function (d) both (a) & (b)
5. The memory is allocated to the data members of class, when ..... of the class is declared.
   (a) constructor (b) destructor
   (c) both (a) & (b) (d) object
6. Constructors, like other member functions, can be declared anywhere in the class.
   (a) True (b) False
7. Constructors do not return any value.
   (a) True (b) False
8. A constructor that accepts no parameters is known as the default constructor.
   (a) True (b) False
9. A class should have at least one constructor.
   (a) True (b) False
10. Destructor takes arguments.
    (a) True (b) False

Answers

1. (b)	2. (c)	3. (b)	4. (a)	5. (d)	6. (a)	7. (a)	8. (a)	9. (b)	10. (b)

Practice Questions

Q.I Answer the Following Questions in short:

1. What are constructor and destructor?
2. How many times will the constructor of class student are invoked for the following statement. student S, * P;
3. State the difference between constructor and destructor.
4. What is destructor? How many destructors can be defined in a single class?
5. Give any four characteristics of constructor.
6. What are the ways of constructor calling in main program? Give their syntax.
7. What is mean by copy constructors?

Q.II Answer the following Questions:

1. Write short note on: Default constructor.
2. Explain multiple constructor with example.
3. What is static data member? Where to declare them?
4. What is meant by default argument? Illustrate concept of constructor with default argument with suitable example.
5. Define parameterized constructor with its syntax and example.
6. Declare a class simple interest having data members as principle amount rate of interest, number of years. The constructor will have default value of rate of interest as 11.5%. Accept this data for two objects. Calculate and display simple interest for each object.
7. Write a program to declare class time having data members as hrs, min. sec. Write a constructor to accept data and use display function for two objects.
8. What are the rules for writing destruction function and when they are invoked?
9. Write a program to find sum of nos. between 1 to n using constructor where value of n will be passed to the constructor.
10. Write a program in C++ which prints the factorial of a given number using a constructor and destructor member function.
11. Write a program in C++ which prints the factorial of a given number using a copy constructor and a destructor member function.
12. Write a program in C++ that determines whether a given number is prime or not and print that number using default constructor and destructor member function.
13. State any four differences between constructor and destructor.

Q.III Define the term:

1. Copy constructor
2. Constructor
3. Parameterized constructor
4. Dynamic constructor
5. Overloaded constructor.

Previous Exam Questions

October 2018

1. Define destructor. **[2 M]**

Ans. Refer to Section 4.8.

2. Explain parameterized constructor with a suitable example. **[4 M]**

Ans. Refer to Section 4.3.

April 2019

1. Write a program for constructor with default arguments. **[4 M]**

Ans. Refer to Section 4.5.

■■■

5...

Inheritance

Objectives...

- To study how to define Base class and Derived class.
- To learn about different Types of Inheritance.
- To understand Virtual Base Class.
- To study Abstract class.
- To learn Constructors in derived class.

5.1 INTRODUCTION [W - 18, S - 19]

- Inheritance is a form of software reusability in which new classes are created from existing classes by absorbing their attributes and behavior and overriding these with capabilities the new classes requires.
- Inheritance is the process by which one object can acquire the properties of another object. Inheritance is the process of creating new classes from an existing class.
- The existing class is known as base class and the newly created class is called as a derived class. The derived class inherits all capabilities of the base class. A derived class is more specific than its base class and represents a smaller group of objects.
- For example, a Red Delicious apple is part of the classification apple, which in turn is part of the fruit class. Hence, inheritance supports the concept of classification; with the help of classification an object need only define those qualities that make it unique within its class.
- The inheritance makes it possible for one object to be a specific instance of a more general case. Example of inheritance hierarchy of shape is shown in Fig. 5.1.

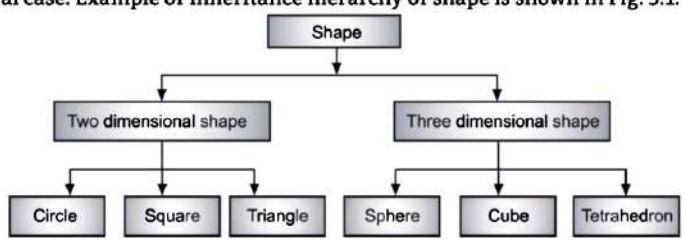


Fig. 5.1: 'Shape' Hierarchy

- The main advantages of inheritance are:
  1. It helps to implement software reusability. Once the hierarchy is done, the specific data to the particular class can be added as per the specification of the derived classes in hierarchy. Reusing existing code saves time and money.
  2. It increases the reliability of the code.
  3. Inheritance can also help in the original conceptualization of a programming problem and in the overall design of the program.
  4. It is useful to avoid redundancy, leading to smaller models that are easier to understand.
  5. It adds some enhancements to the base class.
  6. We can create new class based on base class without modifying base class.

5.2 DEFINING BASE CLASS AND DERIVED CLASS

- In C++, a class that is inherited is referred to as a base class. The class that does the inheriting is called the derived class.
- A derived class can be defined by specifying its relationship with the base class. The pictorial representation of single inheritance is shown in Fig. 5.2.

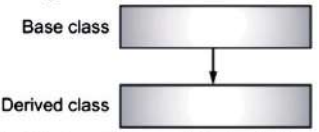


Fig. 5.2: Base Class and Derived Class

- Some example of inheritance is shown in Table 5.1.

Table 5.1: Examples of inheritance

Base Class	Derived Class
Shape	Circle Triangle Rectangle
Employee	Teacher Clerk
Media	Book Tape
Student	Undergraduate Postgraduate

- When a class inherits another, the member of the base class becomes members of the derived class.
- The **syntax** of derived class definition is as follows:

```
class <derived_class_name>: <access modifier> <base_class_name>
{
   // body of class
};
```

It contains:

- Keyword class.
- Derived class name which is user define name for new class.
- The : (colon) indicates that the derived_class_name is derived from the base_class_name.
- The access modifier is optional. If no access modifier is present, the access modifier is private by default. If written, then it must be public, private or protected.
- base_class_name is the existing class name.

- **Example:**

```
class base
{
   ..........
   ..........
};
class derived1: private base          //private derivation
{
   ..........
   ..........
};
class derived2: public base           //public derivation
{
   ..........
   ..........
};
class derived3: protected base        //protected derivation
{
   ..........
   ..........
};
class derived: base                   //by default private derivation
{
   ..........
   ..........
};
```

5.2.1 Public Derivation

- When the access modifier for a base class is **public**, all public members of the base class become public members of the derived class and all protected members of the base class becomes protected members of the derived class.

- The members of derived class can access members of the base class if the base class members are public or protected. They can not access private members.
- The Fig. 5.3 shows the base and derived class relationship.

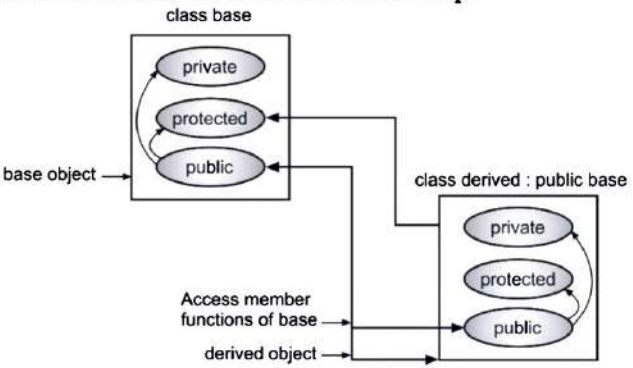


Fig. 5.3: Public Derivation

Program 5.1: Program to demonstrate public derivation.

```
#include<iostream>
using namespace std;
class base
{
   int i, j;
   public:
      void getij()
      {
         cout<<"Enter value of i and j";
         cin>>i;
         cin>>j;
      }
      void show()
      {
         cout<<"i="<<i<<"j="<<j<<"\n";
      }
};
class derived: public base
{
   int k;
   public:
      void getdata()
      {
         getij();
            cout<<"Enter value of k";
            cin>>k;
```

```
            cout<<"k="<<k;
        }
};
int main()
{
    base ob1;        //base object
    derived ob2;     //derived object
    ob1.getij();     //access members of base class
    ob1.show();      //display values of i and j
    ob2.getdata();   //access members of derived class
    ob2.show();      //access members of base class
}
```

Output:

```
Enter value of i and j 2 3
i=2 j=3
Enter value of i and j 5 6
Enter value of k 7
k=7 i=5 j=6
```

- In this program, the base class members are directly access through derived class, because the derived through derived class and derived class is publically derived from base class.
- In inheritance some of the base class data elements and member functions are inherited into the derived class which extends the capability of the existing classes and useful in incremental program development. If the function names of the base and derived class are same then use scope resolution operation (::) and show the scope of the function i.e. to which the class that function belongs.

Program 5.2: Write a program to read data members of a base class i.e. name, person id and the derived class members are height and weight. Display the data of person.

```
#include<iostream>
using namespace std;
class base
{
   private:
      char name[20];
      int person_id;
   public:
      void getdata()
      {
          cout<<"Enter name and person_id";
          cin>>name>>person_id;
      }
```

```
    void display()
    {
        cout<<"name:"<<name<<"  "<<"person_id:"<<person_id;
    }
};
class derived: public base
{
    private:
        int height, weight;
    public:
        void getdata()
        {
            base:: getdata();
            cout<<"Enter height and weight";
            cin>>height>>weight;
        }
        void display()
        {
            base::display();
            cout<<"height:"<<height<<" "<<"weight:"<<weight;
        }
};
int main()
{
    derived obj;
    obj.getdata();
    obj.display();
}
```

Output:

```
Enter name and person_id prajakta 11
Enter height and weight 6 80
name: prajakta  person_id:11 height:6  weight:80
```

5.2.2 Private Derivation

- When the base class is inherited by using the private access modifier, all public and protected members of the base class become private members of the derived class.
- The Fig. 5.4 shows the private derivation of inheritance.

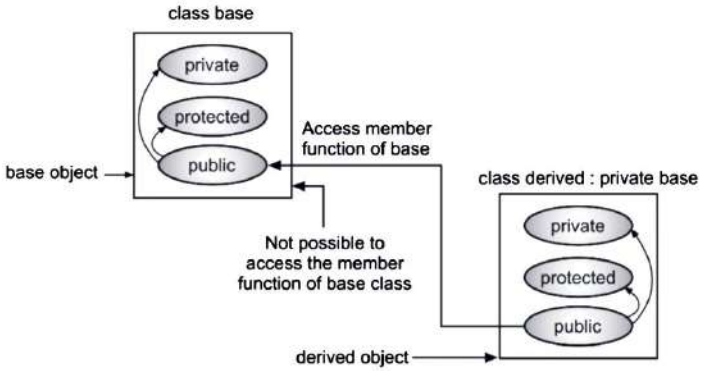


Fig. 5.4: Private Derivation

The following Program 5.3 illustrate this concept.

Program 5.3: Program for private derivation.

```
#include <iostream>
using namespace std;
class Engine
{
   public:
      Engine(int nc)
      {
         cylinder = nc;
      }
      void start()
      {
         cout << getCylinder() <<" cylinder engine started" << endl;
      }
      int getCylinder()
      {
         return cylinder;
      }
   private:
      int cylinder;
};
class Car: private Engine
{
   public:
      Car(int nc = 4): Engine(nc) { }
```

```
        void start()
        {
            cout << "car with " << Engine::getCylinder() <<
            "cylinder engine started" << endl;
            Engine:: start();
        }
};
int main( )
{
    Car c(8);
    c.start();
    return 0;
}
```

Output:

```
car with 8 cylinder engine started
8 cylinder engine started
```

- In this program, derived class object cannot access the functions of base class because they become private. But the public functions of base class access by public functions of derived class.

5.2.3 Protected Members

- The protected keyword provides greater flexibility in the inheritance mechanism. When a member of a class is declared as protected, that member is not accessible by other, non-member elements of program.
- Protected members are as good as private members within the class. Protected member differs from private members when a protected member is inherited. Using the access modifier protected in base class, we could make available the data member in the derived class and restrict its access in other classes or in main().

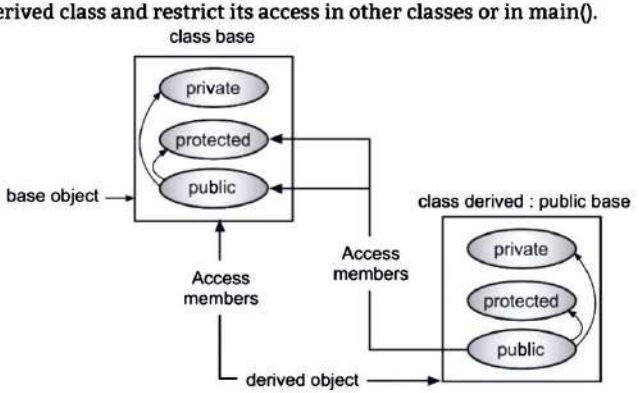


Fig. 5.5: Protected members of base class can be access in derived class

- Data member declare as protected within a base class is accessible by the member within its class and any class immediately derived from it. The Fig. 5.5 shows the public derivation with protected data members.
- If the base class is inherited as public, then the base class protected member becomes protected member of derived class and therefore, can be accessed by derived class. Thus, by using **protected**, we can create class members that are private to their class but can still be inherited and accessed by a derived class.

Program 5.4: Program using protected members.

```
#include<iostream>
using namespace std;
class base
{
   protected:
      int i, j;    // private to base, but accessible by derived class
   public:
      void getij()
      {
          cout<<"Enter value of i and j";
          cin>>i>>j;

      }
      void show()
      {
          cout<<"i="<<i<<" "<<"j="<<j<<"\n";
      }
};
class derived1: public base
{
   int k;
   public:
      void getdata()
      {
         getij();
            k=i*j;
      }
      void showall()
      {
          cout<<"k="<<k;

      }
}; //end of class derived1
```

```
class derived2: public derived1
{
   int x;
   public:
      void get()
      {
         getij();
         x=i*j;
      }  // can access i and j since public inheritance
      void showx()
      {
          cout<<"x="<<x;

      }
}; //end of class derived2
int main()
{
   derived1 ob1;
   derived2 ob2;
   ob1.getdata();
   ob1.showall();
   ob2.get();
   ob2.showx();
}
```

Output:

```
Enter value of i and j 10 20
k=200
Enter value of i and j 15 10
x=150
```

- In the above program, if derived1 class inherits base class privately, then protected member of base becomes private members of derived1 class which cannot inherited in derived2 class. So, when a protected member is inherited in public mode, it becomes protected in the derived class and therefore it is accessible by the member function of the derived class. It is also inherited further.
- A protected member inherited in the private mode becomes private if the derived class and cannot be inherited further.

5.2.4 Protected Derivation

- C++ allows us to use protected as access modifier. When access modifier is protected, all public and protected members of the base class become protected members of the derived class.

Program 5.5: Program for protected derivation.

```
#include<iostream>
using namespace std;
class base
{
   protected:
      int i, j;   //private to base but accessible in derived
   public:
      void getij()
      {
          cin>>i>>j;
      }
      void show()
      {
         cout<<i<<" "<<j<<"\n";
      }
};
class derived: protected base
{
   int k;
   public:
      void getdata()
      {
         getij();
         k = i*j;   // derived may access base protected data members
      }
      void showall()
      {
          cout<<k<<"  ";
          show();

      }
};
int main()
{
   derived ob;
   //ob.getij();    //illegal, getij() is protected member of derived
   ob.getdata();    // correct as public member of derived
   ob.showall();    // correct as public member of derived
   //ob.show();     // illegal
}
```

Output:

```
2   3
6  2  3
```

- In this program, getij( ) and show() are public members of base, but they become protected members of derived. Using derived object is not accessible.

Summary about Access Specifiers:

Table 5.2: Access in a Derived Class

Base class members	Derived Class		
	Public derivation	**Private derivation**	**Protected derivation**
Private	not accessible	not accessible	not accessible
Protected	protected	private	protected
Public	public	private	protected

5.3 TYPES OF INHERITANCE [W - 18, S - 19]

- There are many ways in which we can create derived class. There are five types of inheritance as follows:
  1. Single inheritance
  2. Multilevel inheritance
  3. Multiple inheritance
  4. Hierarchical inheritance and
  5. Hybrid inheritance

5.3.1 Single Inheritance

- When one base class is derived by one child class, then it is known as single inheritance. The existing base class is known as direct base class whereas, they newly created class is called as **singly derived class**. It is shown in Fig. 5.6.

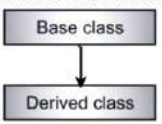


Fig. 5.6: Single Inheritance

- **Example:** Suppose a college want to store information of undergraduate (UG) and postgraduate students (PG). Both are having registration no., name, and address. Each postgraduate student also have additional attributes along with above attributes are placement company name. It can show as in Fig. 5.7.

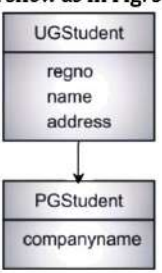


Fig. 5.7

Program 5.6: Program to illustrate single inheritance.

```
#include<iostream>
using namespace std;
class student
{
   protected:
      int regno;
      char name[20];
      char addr[30];
   public:
      void getdata()
      {  cout<<"enter registration no, name and address";
         cin>>regno>>name>>addr;
      }
      void display()
      {
         cout<<"registration no.= "<<regno;
         cout<<"name = "<<name;
         cout<<"address = "<<addr;
      }
};   //end of base class
class PGstudent: public student
{
      private:
            char compname[20];
      public:
            void get_PG_data()
            {
               getdata();
               cout<<"\nEnter the company name=";
               cin>>compname;
            }
            void dispPG()
            {
               cout<<"\n registration no. ="<<regno;
               cout<<"\n name ="<<name;
               cout<<"\n address ="<<addr;
               cout<<"\n company name = "<<compname;
            }
   };
```

```
int main()
{
    student s;
    PGstudent ps;
    s.getdata();
    s.display();
    ps.get_PG_data();
    ps.dispPG();
}
```

Output:

```
Enter registration no, name and address
101 Ram Pune
registration no = 101  name = Ram     addr = Pune
Enter registration no., name & address
401 Sita Pune
Enter the company name = zenex
Registration no = 401
name = Sita
address = Pune
Company name = Zenex
```

Program 5.7: A base class contains the data of employee name, empno and sex. The derived class contains basic salary. The derived class has been declared as an array of class objects. The program illustrate the single inheritance

```
#include<iostream.h>
class base
{
   private:
      char name[20];
      int empno;
      char sex;
   public:
      void get();
      void disp();
};
class derived: public base
{
   private:
      float bsalary;
   public:
      void get();
      void disp();
};
```

```
void base:: get()
{
   cout<<"\n Enter name:";
   cin>>name;
   cout<<"\n Enter Employee id:";
   cin>>empno;
   cout<<"\n Enter Sex:";
   cin>>sex;
}
void base:: disp()
{
   cout<<"name ="<<name;
   cout<<"Emp no. is ="<<empno;
   cout<<"sex="<<sex;
}
void derived:: get()
{
   base:: get();
   cout<<"Enter basic salary: \n";
   cin>>bsalary;
}
void derived:: disp()
{
   base::disp();
   cout<<"basic salary = "<<bsalary;
}
void main()
{
   derived ob[10];
   int i, n;
   cout<<"How many employees?";
   cin>>n;
   for (i=0;i<n;i++)
   {
      ob[i].get();
   }
   cout<<endl;
   cout<<"name Empno.sex bsalary \n";
```

```
    for (i=0;i<n;i++)
    {
        ob[i].disp();
        cout<<endl;
    }
}
```

Output:

```
How many employees? 2

Enter name: Prajakta
Enter Employee id: 101
Enter Sex: f
Enter basic salary: 10000

Enter name: Pranjal
Enter Employee id: 102
Enter Sex: m
Enter basic salary: 40000

name            Empno              sex          bsalary
name =abc    Emp no. is =101       sex=f        basic salary = 10000
name =xyz    Emp no. is =102       sex=m        basic salary = 40000
```

5.3.2 Multilevel Inheritance

- When a base class is derived by a child class which further derived by another child class, then it is known as Multilevel Inheritance.
- The class which provides link between two classes is known as intermediate base class. This is illustrate in Fig. 5.8.

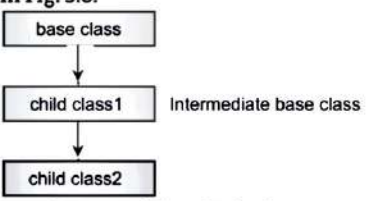


Fig. 5.8: Multilevel Inheritance

Practically, we can have more than two levels also.

- **Syntax of multilevel inheritance:**

```
class base
{
    .....
    .....
};
```

```
class child1: public base
{
    .....
    .....
};
class child2: public child1
{
    .....
    .....
};
```

- Consider an example of bank. In a bank, different customers have savings account. Some customers may have taken loan from the bank. So the bank always maintains information about bank depositors and borrowers. This can be shown using the following Fig. 5.9.

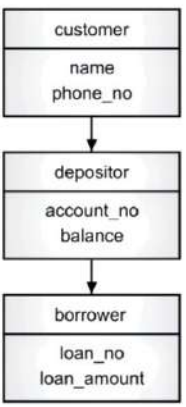


Fig. 5.9: Example of Bank

Program 5.8: Program to illustrate use of multilevel inheritance.

```
#include<iostream>
using namespace std;
class customer       // base class
{
   protected:
      char name[30];
      int phone_no;
}; // end of base class
```

```
class depositor: public customer     // child class1
{
   private:
      int account_no;
      float balance;
   public:
      void get_depositor_data( )
      {
         cout<<"\n Enter name & phone no ";
         cin>>name>>phone_no;
         cout<<"\n Enter account no and balance ";
         cin>>account_no>>balance;
      }
      void display_depositor_data( )
      {
         cout<<"\n Name = "<<name;
         cout<<"\n Phone no = "<<phone_no;
         cout<<"\n Account no = "<<account_no;
         cout<<"\n Balance = "<<balance;
      }
}; // end of child class1
class borrower: public depositor            // child class2
{
   protected:
      int loan_no;
      float loan_amount;
   public:
      void get_loan_data( )
      {
         cout<<"\n Enter loan no and loan amount";
         cin>>loan_no>>loan_amount;
      }
      void display_loan_data( )
      {
         cout<<"\n Loan no = "<<loan_no;
         cout<<"\n Loan amount = "<<loan_amount;
      }
}; // end of child class2
```

```
int main( )
{
   int choice;
   cout<<"1-Read & display depositor information "<<endl;
   cout<<"2-Read & display depositor & borrower information"<<endl;
   cout<<"Enter your choice = ";
   cin>>choice;
   switch (choice)
   {
      case 1:
         depositor d;
         d.get_depositor_data( );
         d.display_depositor_data( );
         break;
      case 2:
         borrower b;
         b.get_depositor_data( );
         b.get_loan_data( );
         b.display_depositor_data( );
         b.display_loan_data( );
         break;
   }
}  // end of main( )
```

Output:

```
1-Read & display depositor information
2-Read & display depositor & borrower information
Enter your choice = 1
Enter name & phone no Prajakta 98776677
Enter account no and balance 123456789 10000
Name = Prajakta
Phone no = 98776677
Account no = 123456789
Balance = 10000
```

- In the above program, though the data members of depositor class are private, they can be accessed indirectly by the borrower class through the member functions of the depositor class indirectly.

Program 5.9: Consider the example of student data. Base class is student. Class test is intermediate base class and result is child class. The class result inherits the details of marks obtained in the test and the roll no. of student from base class through multilevel inheritance. [S - 18]

```
#include<iostream>
using namespace std;
class student
{
   protected:
      int roll;
   public:
      void get(int x)
      {
         roll = x;
      }
      void display()
      {
         cout<<"roll="<<roll;
      }
};
class test: public student
{
   protected:
      int mark1, mark2;
   public:
      void getmark(int a, int b)
      {
         mark1 = a;
         mark2 = b;
      }
      void dispmark()
      {
         cout<<"mark1 = \n"<<mark1<<"mark2 ="<<mark2;
      }
};
class result: public test
{
   private:
      int total;
```

```
    public:
        void displayall()
        {
            total = mark1 + mark2;
            display();
            dispmark();
            cout<<"total="<<total;
        }
};
int main()
{
    result r;
    r.get(10);
    r.getmark(60,75);
    r.displayall();
}
```

Output:

```
roll = 10
mark1 = 60
mark2 = 75
total = 135
```

5.3.3 Multiple Inheritance

- When more than one base classes are inherited by a derived class, then such type of inheritance is called as multiple inheritance. Fig. 5.10 illustrates the point.

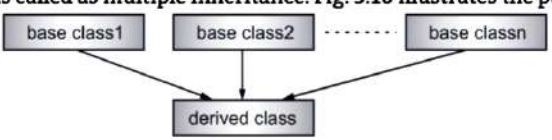


Fig. 5.10: Multiple inheritance

Example: A child inherits the features from father and intelligence from mother.

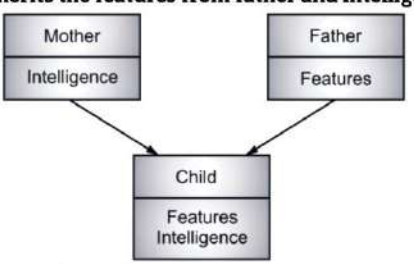


Fig. 5.11: Multiple Inheritance

- **Syntax of Multiple Inheritance:**

1. **Multiple inheritance with all public derivation:**

```
class base1
{
   .....
   .....
   .....
};
class base2
{
   .....
   .....
   .....
};
class derived: public base1, public base2
{
   .....
   .....
   .....
};
```

Here access specifier may be public or private and the base classes are separated by commas.

2. **Multiple inheritance with all private derivation:**

```
class base1
{
   .....
   .....
   .....
};
class base2
{
   .....
   .....
   .....
};
class derived: private base1, private base2
{
   .....
   .....
   .....
};
```

3. Multiple inheritance with all mixed derivation:

```
class base1
{
   .....
   .....
   .....
};
class base2
{
   .....
   .....
   .....
};
class base3
{
   .....
   .....
   .....
};
class derived: pubic base1, public base2, private base3
{
   .....
   .....
   .....
};
```

Let us discuss the program how multiple inheritance works.

Program 5.10: Program for multiple inheritance.

```
#include<iostream>
using namespace std;
class basex
{
   protected:
      int x;
   public:
      void get_x (int a) {x = a;}
};
class basey
{
   protected:
      int y;
   public:
      void get_y (int b) {y = b;}
};
```

```
class derived: public basex, public basey
{
   public:
      void display()
      {
         cout<<"x = " <<x<<"\n";
         cout<<"y = "<<y<<"\n";
         cout<<"x+y = "<<x+y<<"\n";
      }
};
int main()
{
   derived ob;
   ob.get_x(50);
   ob.get_y(60);
   ob.display();
}
```

Output:

```
x = 50
y = 60
x + y = 110
```

Program 5.11: Consider an example to print bio-data of a diploma student having personal and academic information. Program to illustrate use of multiple inheritance. [S - 19]

```
#include<iostream>
#include<cstring>
using namespace std;
class personal                                    // base class1
{
   private:
      char name [30], email_id[50];
   protected:
   void get_personal_info( )
   {
      cout<<"\n Enter name & email id of a person = ";
      cin>>name>>email_id;
   }
```

```
    void display_personal_info( )
    {
        cout<<"\n Name = "<<name;
        cout<<"\n Email_id = "<<email_id;
    }
};  // end of base class1
class academic                                  // base class 2
{
    private:
        float tenth_marks;
        char tenth_class[20];
    protected:
        void get_academic_info( )
        {
            cout<<"\n Enter tenth marks";
            cin>>tenth_marks;
        }
        void find_class( )
        {
            if (tenth_marks>=70)
                strcpy (tenth_class, "Distinction");
            else
            if (tenth_marks>=60)
                strcpy (tenth_class, "First class");
            else
            if (tenth_marks>=50)
                strcpy (tenth_class, "Second class");
            else
            if (tenth_marks>=40)
                strcpy (tenth_class, "Pass class");
            else
                strcpy (tenth_class, "Fail");
        }
        void display_academic_info( )
        {
            cout<<"\n Tenth marks = "<<tenth_marks;
            find_class( );
            cout<<"\n Tenth class = ";
            cout<<tenth_class;
        }
}; // end of base class2
```

```
class bio_data: private personal, private academic      // Derived class
{
   public:
      void display_biodata( );
      void get_info( );
}; // end of derived class
   // Member functions of derived class
void bio_data::display_biodata( )
{
   cout<<"BIO DATA "<<endl;
   cout<<"Personal info = "<<endl;
   display_personal_info( );
   cout<<"Academic info= "<<endl;
   display_academic_info( );
}
void bio_data::get_info( )
{
   get_personal_info( );
   get_academic_info( );
}
int main( )
{
   bio_data b;
   // First read info
   b.get_info( );
   // Now print biodata
   b.display_biodata( );
}
```

Output:

```
Enter name & email id of a person = Prajakta abc@yahoo
Enter tenth marks 80.5
BIO DATA
Personal info =
Name = Prajakta
Email_id = abc@yahoo
Academic info=
Tenth marks = 80.5
Tenth class = Distinction
```

Ambiguity in Multiple Inheritance:

- One common cause of ambiguity in multiple inheritance is when two or more base classes have methods with the same name, while there is no method of that name in the derived class. In this case objects of the derived class have no way of knowing which of the parent methods is to be executed. The scope resolution operator can be used to resolve the ambiguity.
- Another element of ambiguity in multiple inheritance arises when two or more derived classes inherit the same member from a common base class. If now another derived class inherits from these classes, two copies of the original member function could be inherited.
- Fig. 5.12 shows one possible scenario that leads to inheritance of multiple copies.
- One possible solution to this duplicity is the use of the scope resolution operator. Another solution of this problem is to declare the base classes to be virtual at the time of they are inherited.

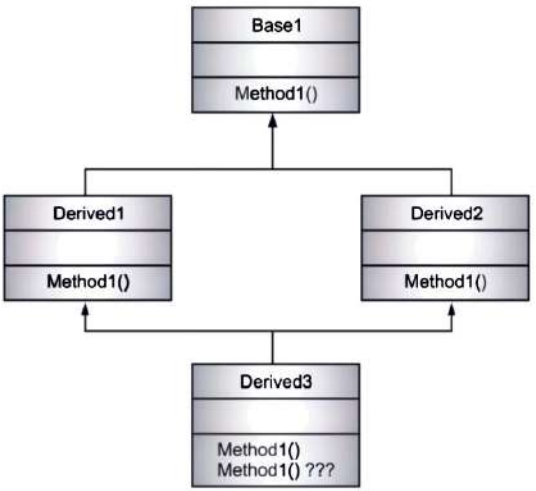


Fig. 5.12: Inheritance of Multiple Copies

Advantages of Multiple Inheritance:

1. Derived class typically represents a combination of its base classes.
2. It has rich semantics
3. It has ability to directly express complex structures.

5.3.4 Hierarchical Inheritance

- When one base class is inherited by more than one derived classes, it is known as hierarchical inheritance. In this, many programming problems can be cast into a hierarchy where certain features of one level are shared by many other below that level.
- The Fig. 5.13 illustrate this point.

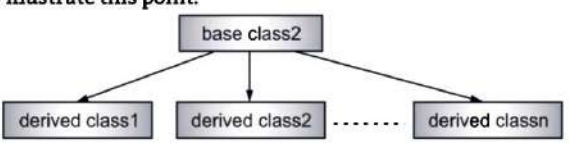


Fig. 5.13: Hierarchical Inheritance

- **Syntax of Hierarchical Inheritance:**

```
class base
{
   .....
   .....
   .....
};
class derived1: public base
{
   .....
   .....
   .....
};
class derived2: public base
{
   .....
   .....
   .....
};
```

- **Example:** Consider an example of an organisation having full time and part time employees. Depending on it, the employee salary is calculated. The hierarchy is:

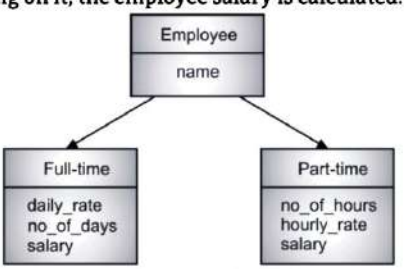


Fig. 5.14: Hierarchical Inheritance

Program 5.12: Program to illustrate use of hierarchical inheritance.

```
#include<iostream>
using namespace std;
class employee  // base class
{
   protected:
      char name[30];
}; // end of base class
```

```
class full_time_emp: protected employee
{
   protected:
      float daily_rate, salary;
      int no_of_days;
   public:
      void read_emp_data( );
      void display_emp_data( );
      void calculate_salary( );
}; // end of derived class1

void full_time_emp::read_emp_data( )
{
   cout<<endl<<"Enter name of employee =";
   cin>>name;
   cout<<endl<<"Enter working rate for one day = ";
   cin>>daily_rate;
   cout<<endl<<"Enter no. of working days of employee=";
   cin>>no_of_days;
}
void full_time_emp::calculate_salary( )
{
   salary = daily_rate *no_of_days;
}
void full_time_emp::display_emp_data( )
{
   cout<<endl<<"For the full time employee:=";
   cout<<endl<<"Name="<<name;
   cout<<endl<<"Daily rate ="<<daily_rate;
   cout<<endl<<"No. of working days ="<<no_of_days;
   cout<<endl<<"Salary ="<<salary;
}
class part_time_emp:protected employee                    // derived class2
{
   protected:
         float hourly_rate, salary;
         int no_of_hours;
   public:
         void read_emp_data( );
         void display_emp_data( );
         void calculate_salary( );
}; // end of derived class 2
```

```
void part_time_emp::read_emp_data( )
{
   cout<<endl<<"Enter name of employee=";
   cin>>name;
   cout<<endl<<"Enter working rate for one hour =";
   cin>>hourly_rate;
   cout<<endl<<"Enter no. of working hours for employee =";
   cin>> no_of_hours;
}
void part_time_emp::calculate_salary( )
{
   salary = hourly_rate * no_of_hours;
}
void part_time_emp::display_emp_data( )
{
   cout<<endl<<"For the part time employee";
   cout<<endl<<"Name ="<<name;
   cout<<endl<<"Hourly rate ="<<hourly_rate;
   cout<<endl<<"No. of working hours="<<no_of_hours;
   cout<<endl<<"Salary="<<salary;
}
int main( )
{
   int choice;
   cout<<endl<<"You have following choices";
   cout<<endl<<"1-Read & display information of full time employee";
   cout<<endl<<"2-Read & display information of part time employee";
   cout<<endl<<"Enter your choice=";
   cin>>choice;
   switch(choice)
   {
      case 1:
            full_time_emp ft;
            // First read data
            ft.read_emp_data( );
            ft.calculate_salary( );
            // Now display data
            ft.display_emp_data( );
            break;
```

```
        case 2:
                part_time_emp pt;
                // First read data
                pt.read_emp_data( );
                pt.calculate_salary( );
                // Now display data
                pt.display_emp_data( );
                break;
    }
}
```

Output:

```
You have following choices
1-Read & display information of full time employee
2-Read & display information of part time employee
Enter your choice=1
Enter name of employee = Prajakta
Enter working rate for one day = 250
Enter no. of working days of employee=30
For the full time employee:=
Name=Prajakta
Daily rate =250
No. of working days =30
Salary =7500
```

- In the above program, both derived classes have same names for the member functions. You can also use different names.
- Also in the above program, you can use array of objects to store information about more than one employee.

5.3.5 Hybrid Inheritance

- Sometimes, we may require to combine two or more types of inheritance to design a program; the result is known as Hybrid Inheritance. Fig. 5.15 illustrates this point.

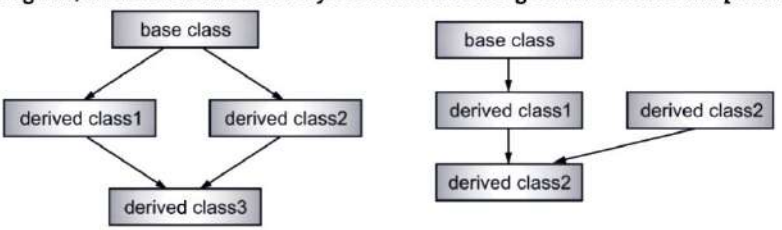


Fig. 5.15: Hybrid Inheritance

- **Example:** Consider an example where we have three classes' student, test and result. But some authority wants that some weightage should be given for sports to finalize the result. This is shown in Fig. 5.16.

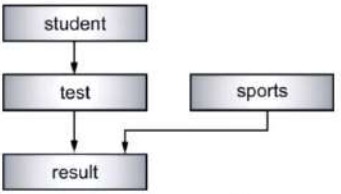


Fig. 5.16: Hybrid Inheritance (Multiple and Multilevel)

Program 5.13: Program for hybrid inheritance.

```
#include<iostream.h>
class student
{
   protected:
      int rollno;
   public:
      void get (int x)
      { rollno = x; }
      void display()
      { cout<<rollno; }
};
class test: public student
{
   protected:
      int mark1, mark2;
   public:
      void getmark(int a, int b)
      {
         mark1 = a;
         mark2 = b;
      }
      void dispmark()
      { cout<<mark1<<"  "<<mark2; }
};
class sport
{
   protected:
      int smark;
```

```
    public:
        void getsm(int s)
        { smark = s; }
        void disps()
        { cout<<smark<<"\n"; }
};
class result: public test, public sport
{
    private:
        int total;
    public:
        void displaytotal()
        {
            total = mark1 + mark2 + smark;
            display();
            dispmark();
            disps();
            cout<<"total = "<<total;
        }
};
void main()
{
    result r;
    r.get(10);
    r.getmark(50, 60);
    r.getsm(70);
    r.displaytotal();
}
```

Output:

```
10    50    60    70
total = 180
```

5.4 VIRTUAL BASE CLASS [W - 18, S - 19]

- Consider a situation where multilevel, multiple and hierarchical all the three kinds of inheritance are involved. This is illustrate in Fig. 5.17.
- In the Fig. 5.17, the base class is inherited by both Derived1 and Derived2. Derived3 directly inherits both Derived1 and Derived2. All the public and protected members of Base are inherited into Derived3 twice through both Derived1 and Derived2. Therefore, Derived3 would have duplicate sets of members inherited. This causes ambiguity when a member of Base is used by Derived3.

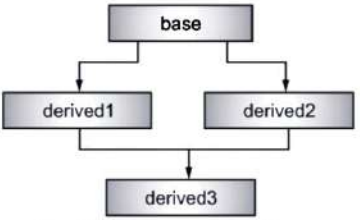


Fig. 5.17: Multipath Inheritance

- To resolve this ambiguity, C++ includes a mechanism by which only one copy of Base will be included in Derived3. This feature is called a virtual base class. When a class is made virtual, C++ take necessary care to inherit only one copy of that class. The keyword virtual precedes the base class access specifier when it is inherited by a derived class.
- The situation in Fig. 5.17 can be declared as:

```
class Base
{ ..... };
class Derived1: virtual public Base
{ ..... };
class Derived2: virtual public Base
{ ..... };
class Derived3: public Derived1, public Derived2
{ ..... };
```

Program 5.14: Program using virtual base class.

```
#include<iostream.h>
//using namespace std;
class base
{
   public:
   int x;
};
class derived1: virtual public base
{
   public:
   int y;
};
class derived2: virtual public base
{
   public:
   int z;
};
```

```
/* derived3 inherits derived1 and derived2. However, only one copy of base
is present */
class derived3: public derived1, public derived2
{
   public:
   int mult()
   { return x * y * z; }
};
void main()
{
   derived3 d;
   d.x = 10;
   d.y = 20;
   d.z = 30;
   cout<<"product is"<<d.mult();
}
```

Output:

```
product is 6000
```

Note:

- If derived1 and derived2 classes had not inherited base as virtual, then statement d.x = 10; becomes ambiguous and a program would have resulted as compile-time error.
- If virtual base classes are used when an object inherits the base more than once, only one base class is present in the object.
- If normal base classes are used when an object inherits the base more than once, the multiple copies will be found; and it is compile-time error.

Program 5.15: To calculate the total mark of a student using the concept of virtual base class.

```
#include <iostream>
using namespace std;
class student
{
  int rno;
  public:
  void getnumber()
  {
      cout<<"Enter Roll No:";
      cin>>rno;
   }
```

```
    void putnumber()
    {
    cout<<"\n\n\tRoll No:"<<rno<<"\n";
    }
};
class test:virtual public student
{
  public:
    int part1,part2;
    void getmarks()
    {
        cout<<"Enter Marks\n";
       cout<<"Part1:";
       cin>>part1;
       cout<<"Part2:";
       cin>>part2;
    }
    void putmarks()
    {
       cout<<"\tMarks Obtained\n";
       cout<<"\n\tPart1:"<<part1;
       cout<<"\n\tPart2:"<<part2;
    }
};
class sports:public virtual student
{
  public:
     int score;
     void getscore()
     {
        cout<<"Enter Sports Score:";
       cin>>score;
     }
     void putscore()
     {
       cout<<"\n\tSports Score is:"<<score;
     }
};
```

```
class result:public test,public sports
{
   int total;
   public:
   void display()
   {
      total=part1+part2+score;
      putnumber();
      putmarks();
      putscore();
      cout<<"\n\tTotal Score:"<<total;
   }
};
int main()
{
   result obj;
   obj.getnumber();
   obj.getmarks();
   obj.getscore();
   obj.display();
}
```

Output:

```
Enter Roll No: 151
Enter Marks

Part1: 90
Part2: 80
Enter Sports Score: 80

Roll No: 151
Marks Obtained
Part1: 90
Part2: 80
Sports Score is: 80
Total Score is: 250
```

5.5 ABSTRACT CLASS [S - 18]

- A class that contains at least one pure virtual function is said to be abstract. A pure virtual function is a member function i.e., declared in an abstract class, but defined in a derived class.
- An abstract class does not create any object and it contains one or more functions for which there is no definition (i.e., a pure virtual function).

- We can create pointers and references to an abstract class. This allows abstract classes to support runtime polymorphism, which depends upon base class pointers and references to select the proper virtual function.
- An abstract class is used only to derive other classes. For example: Shape is abstract base class which derives square, circle and rectangle.

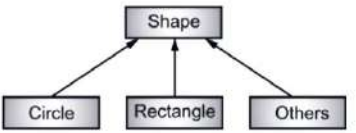


Fig. 5.18

Program 5.16: Following program shows use of abstract classes.

```
#include <iostream>
using namespace std;
// Abstract class
class Shape
{
    protected:
       float l;
    public:
       void getData()
       {
           cin >> l;
       }
       // virtual Function
       virtual float calculateArea() = 0;
};
class Square: public Shape
{
    public:
       float calculateArea()
       {   return l*l;  }
};
class Circle: public Shape
{
    public:
       float calculateArea()
       { return 3.14*l*l; }
};
```

```
int main()
{
    Square s;
    Circle c;
    cout << "Enter length to calculate the area of a square: ";
    s.getData();
    cout<<"Area of square: " << s.calculateArea();
    cout<<"\nEnter radius to calculate the area of a circle: ";
    c.getData();
    cout << "Area of circle: " << c.calculateArea();
    return 0;
}
```

Output:

```
Enter length to calculate the area of a square: 4
Area of square: 16
Enter radius to calculate the area of a circle: 5
Area of circle: 78.5
```

5.6 CONSTRUCTORS IN DERIVED CLASS

- There are two major questions that arise relative to constructors and destructors when inheritance is involved. First, when are base class and derived class constructor and destructor functions called? Second, how can parameters be passed to base class constructor functions?
- As long as no base class constructor takes any argument, the derived class need not have a constructor function. If a base class contains a constructor with one or more arguments then it is mandatory for a derived class to have a constructor pass the arguments to the base class constructor.

1. Order of calling constructors:

- When both the derived and base classes contain constructors, then base constructor is executed first and then the constructor is the derived class is executed.
- In case of multiple and multilevel inheritance, base classes are constructed in the order in which they appear in the declaration of derived class.
- When a derived object is destroyed its destructor is called first and then base class destructor.
- Constructor's functions are executed in their order of derivation. Destructor functions are executed in reverse order of derivation.

Program 5.17: Program for order of calling constructor.

```
#include<iostream.h>
class base
{
   public:
      base()
      {
         cout<<"constructing base \n";
      }
```

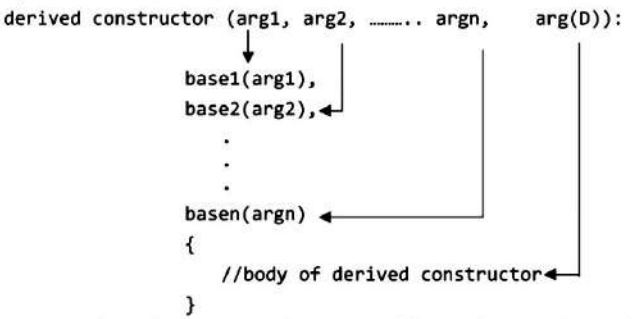
```
        ~base()
        {
            cout<<"destructing base \n";
        }
};
class derived1: public base
{
    public:
        derived1()
        {
            cout<<"constructing derived1 \n";
        }
        ~derived1()
        {
            cout<<"destructing derived1 \n";
        }
};
class derived2: public derived1
{
    public:
        derived2()
        {
            cout<<"constructing derived2 \n";
        }
        ~derived2()
        {
            cout<<"destructing derived2 \n";
        }
};
int main()
{
    derived2 ob;
    //construct and destruct ob
    return 0;
}
```

Output:

```
constructing base
constructing derived1
constructing derived2
destructing derived2
destructing derived1
destructing base
```

2. Passing parameter to base class constructor:

- In case where derived class constructor requires one or more parameters, you simply use standard parameterized syntax. However, how do we pass arguments to a constructor in a base class?
- The constructor of the derived class receives the entire list of arguments with their values and passes them to base constructor. Following is the general syntax of derived constructor.
- **The syntax of derived class constructor:**



- Here, base1 through basen are the names of base classes inherited by derived class. A colon separates the derived class constructor declaration from base_class specification and base class specifications are separated from each other by commas.
- In case of multiple base classes arg1, arg2 .....argn represents the actual parameters that are passed to the base constructor and arg(D) provides the parameters that are necessary to initialize the number of derived class.
- **For example:**

```
derived (int a, int b, int x)
base1(a);        //call constructor of base1
base2(b);        //call constructor of base2
{
   t = x;        //executes its own body
}
```

Program 5.18: Program for passing parameter to base constructor.

```
#include <iostream>
using namespace std;
class BaseClass1
{
   public:
      BaseClass1()
      {
```

```
        cout << "BaseClass1 constructor." << endl;
      }
};
class BaseClass2
{
   public:
      BaseClass2()
      {
        cout << "BaseClass2 constructor." << endl;
      }
};
class BaseClass3
{
   public:
      BaseClass3()
      {
        cout << "BaseClass3 constructor." << endl;
      }
};
class DerivedClass: public BaseClass1, public BaseClass2, public BaseClass3
{
   public:
      DerivedClass()
      {
        cout << "DerivedClass constructor." << endl;
      }
};
int main()
{
    DerivedClass dc;
}
```

Output:

```
BaseClass1 constructor.
BaseClass2 constructor.
BaseClass3 constructor.
DerivedClass constructor.
```

- In inheritance, destructors are executed in reverse order of constructor execution. The destructors are executed when an object goes out of scope.
- Base class constructors are called first and the derived class constructors are called next in single inheritance.

- Destructor is called in reverse sequence of constructor invocation. The destructor of the derived class is called first and the destructor of the base is called next.

Program 5.19: Program for constructor and destructors in derived classes.

```
#include<iostream>
using namespace std;
class base
{
   public:
    base()
    {
        cout<<"base class constructor"<<endl;
    } ~base()
    {
        cout<<"base class destructor"<<endl;
    }
};
class derived:public base
{
   public:
    derived()
    {
        cout<<"derived class constructor"<<endl;
    } ~derived()
    {
        cout<<"derived class destructor"<<endl;
    }
};
int main()
{
    derived d;
    return 0;
}
```

Output:

```
base class constructor
derived class constructor
derived class destructor
base class destructor
```

5.6.1 Constructors for Virtual Base Classes

- Constructors for virtual base classes are invoked before any non-virtual base class. For example,

```
class A: public B, virtual C
{
    .....
    .....
};
```

- Virtual base class C constructors first executed then B's constructor and then A's constructor.
- If the hierarchy contains multiple virtual base classes, then the virtual base class constructors are invoked in the order in which they are declared.

Table 5.3: Order of invocation of constructors

Method of Inheritance	Order of Execution

`class D: public B`
`{`
`  .....`
`};`
	B ( ): base constructor
	D ( ): derived constructor

`class D: public B1, public B2`
`{`
`  .....`
`};`
	B1 ( ): base constructor
	B2 ( ): base constructor
	D ( ): derived constructor

`class D: public B2, public B1`
`{`
`  .....`
`};`
	B2 ( ): base constructor
	B1 ( ): base constructor
	D ( ): derived constructor

`class D: public B1, virtual B2`
`{`
`  .....`
`};`
	B2 ( ): virtual base constructor
	B1 ( ): base constructor
	D ( ): derived constructor

`class D1: public B`
`{`
`  .....`
`};`
`class D2: public D1`
`{`
`  .....`
`};`
	B ( ): superbase constructor
	D1 ( ): base constructor
	D2 ( ): derived constructor

Summary

- Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.
- The technique by which features and properties of an old class is transferred into a new class is called inheritance or derivation. The old class is referred to as the base class or super class and new one is called the derived class or subclass.
- A derived class with only one base class is called single inheritance and one with several base classes is called multiple inheritance. A class may be inherited by more than one class is known as hierarchical inheritance.
- When a class is derived from another one of more base classes, this process is termed hybrid inheritance.
- The mechanism of deriving a class from another 'derived class' is known as multilevel inheritance.
- When classes are declared virtual, the compiler takes essential caution to avoid duplication of member variables. Thus, we make a class virtual if it is a base class that has been used by more than one derived class as their base class.
- When deriving a class from a base class, the base class may be inherited through public, protected or private.
- When deriving from a protected base class, public and protected members of the base class become protected members of the derived class.
- When deriving a class from a public base class, public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class.
- A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.
- When deriving from a private base class, public and protected members of the base class become private members of the derived class.
- The execution of constructors takes place from base class to derived class. The execution of destructors is in opposite order as compared with constructors i.e., from derived class to base class.
- If more than one subclass is derived from the same base class, then the base class is called a virtual base class.
- A class containing a pure virtual function is called an abstract class. It is called abstract because we cannot define objects of a class containing a pure virtual function. It exists only for the purpose of defining classes that are derived from it.

Check Your Understanding

1. The process of creating new classes from an existing class is called as .....
   (a) polymorphism (b) polyinheritance
   (c) inheritance (d) none
2. If a base class is privately inherited by a derived class become ..... member to derived class.
   (a) public (b) private
   (c) visible (d) none
3. A class which is used only to derive other classes is known as ..... class.
   (a) abstract (b) virtual
   (c) derived (d) none.
4. Hybrid inheritance is .....
   (a) multiple inheritance (b) multilevel inheritance
   (c) multipath inheritance (d) both (a) & (b)
5. A class is inherited known as .....
   (a) base class (b) hybrid
   (c) derived class (d) both (a) & (b)
6. Following ..... operator is used for resolve ambiguity.
   (a) + (b) ::
   (c) – (d) none of these
7. Inheritance helps in making a general class into a more specific class.
   (a) True (b) False
8. Inheritance facilitates the creation of class libraries.
   (a) True (b) False
9. Defining a derived class requires some changes in the base class.
   (a) True (b) False
10. A base class is never used to create objects.
    (a) True (b) False

Answers

1. (c)	2. (b)	3. (a)	4. (d)	5. (a)	6. (b)	7. (b)	8. (a)	9. (b)	10. (b)

Practice Questions

Q.I Answer the following questions in short:

1. What is main advantage of Inheritance?
2. List out different types of inheritance.
3. What is the difference between the Base and Derived Classes?
4. Differentiate between Public and Private Inheritance.
5. What is a Virtual Base Class?

Q.II Answer the following questions:

1. Explain different types of Inheritance with example.
2. What are advantages and disadvantages of declaring Inheritance?
3. What is the scope of the accessibility of variables in the protected section and in the private section of a class?
4. When can Multiple Inheritance lead to ambiguity?
5. How is a Single Inheritance different from Multiple Inheritance?
3. When do we make a class Virtual?
4. Write short note on:
   (i) Multiple Inheritance
   (ii) Base Class and Derived Class
5. Can a Member Function in the Derived Class have the same name as that of one in the Base Class?

Q.III Define the term:

1. Inheritance
2. Single level inheritance
3. Multiple level inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

Previous Exam Questions

April 2018

1. What is an Abstract Class? **[2 M]**

Ans. Refer to Section 5.5.

2. Create a base class student(rollno, name) which derives two classes test(mark1, mark2), sport(score). Class result(total_marks, grade) inherits both test and sport classes. Write C++ program to calculate total_marks, percentage and to display marksheet. **[4 M]**

Ans. Refer to Program 5.9.

3. Trace output of the following program. Assume there is no error: **[4 M]**

```
#include<iostream>
using namespace std;
class P
{
   public:
   void print( )
   {
      cout<<"Inside P";
   }
};
```

```
class Q: public P
{
   public:
   void print( )
   {
      cout<<"Inside Q";
   }
};
class R: public Q
{
};
int main(void)
{
   R r;
   r.print( );
   return 0;
}
```

Output:

```
Inside Q
```

October 2018

1. What is inheritance? Explain its three types. **[4 M]**

Ans. Refer to Section 5.1, 5.3

2. Write a program which will implement the concept of virtual base class. **[4 M]**

Ans. Refer to Section 5.4

3. Trace the output of the following program and explain it. Assume there is no syntax error: **[4 M]**

```
#include <iostream>
using namespace std;
class Base1
{
    public:
    ~ Base1 ( )
    {
        cout << "Base1's destructor" << endl;
    }
};
class Base2
{
    public:
    ~ Base2 ( )
    {
        cout << "Base 2's destructor" << endl;
    }
};
```

```
class Derived: public Base1, public Base2
{
    public:
    ~ Derived ( )
    {
        cout << "Derived's destructor" << endl;
    }
};
int main ( )
{
    Derived d;
    return 0;
}
```

Output:

```
Derived's destructor
Base 2's destructor
Base1's destructor
```

April 2019

1. What is inheritance? Explain it with its types. [4 M]

Ans. Refer to Section 5.1, 5.3

2. Explain virtual base class with suitable diagram. [4 M]

Ans. Refer to Section 5.4

3. Design a C++ program to create two base classes personnel (name, address, e-mail-id, DOB) and academic (10th std marks, 12th std marks, class obtained). Derive a class Bio_data from both these classes and prepare a bio data of a student having personal and academic information. [4 M]

Ans. Refer to Program 5.11

4. Trace the output of the following program and explain it. Assume there is no syntax error: [4 M]

```
#include <iostream>
using namespace std;
class Base1
{
    public:
    Base1()
    {
        cout << "Base1's constructor called" << endl;
    }
};
```

```
class Base2
{
    public:
    Base2()
    {
        cout << "Base2's constructor called" << endl;
    }
};
class Derived: public Base2, public Base1
{
    public:
    Derived()
    {
        cout << "Derived's constructor called" <<endl;
    }
};
int main()
{
    Derived d;
    return 0;
}
```

Output:

```
Base2's constructor called
Base1's constructor called
Derived's constructor called
```

6... Polymorphism

Objectives...

- To study about Compile time and run time polymorphism.
- To study function overloading.
- To learn about operator overloading.

6.1 INTRODUCTION [S - 18, W - 18]

- Polymorphism means "one name, many forms". Polymorphism is the process of defining a number of objects at different classes into group and call the method to carry out the operation of the objects using different function calls.
- There are two types of polymorphism:
  1. **Compile time polymorphism:** This is also called as **early** or **static** binding. Selection of an appropriate function for a particular call at the compile time itself. For example, function overloading and operator overloading.
  2. **Run-time polymorphism:** This is also called as **dynamic binding** or **late binding**. Sometimes, a situation occurs where function name and prototype is same in both the base class and in the derived class. Compiler does not know what to do, which function to call. In this class appropriate member function is selected to run time. For example, virtual function.

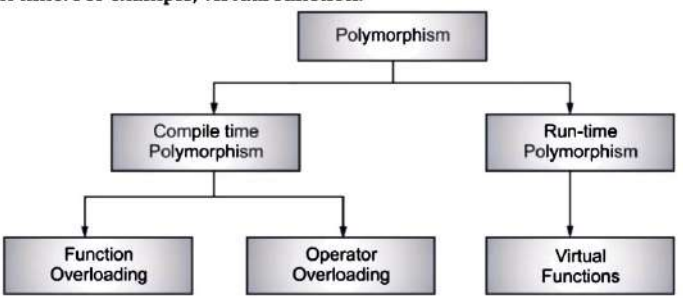


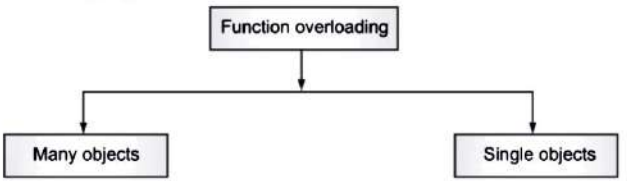
Fig. 6.1: Types of Polymorphism

6.2 COMPILE TIME POLYMORPHISM [S - 19]

- Polymorphism, in C++, is implemented through overloaded functions and overloaded operators.
- Function Overloading is also referred to as functional polymorphism. The same function can perform a wide variety of tasks.
- The same function can handle different data types. When many functions with the same name but different argument lists are defined, then the function to be invoked corresponding to a function call is known during compile time.
- When the source code is compiled, the functions to be invoked are bound to the compiler during compile time, as to invoke which function depending upon the type and number of arguments. Such a phenomenon is referred to early binding, static linking or compile time polymorphism.

6.2.1 Function Overloading [S - 19]

- C++ provides the facility of function overloading in which we can define multiple functions with the same name but the types of argument are different.
- For example, we can have a function add with two floating point numbers as argument, another function add with 3 integers, one add function with 4 doubles and so on. But function name is similar. This is called as **Function Overloading**.
- An overloaded function, is a function with the same name as another function, but with different parameter types.
- The function overloading can be performed using single and many objects. This is shown in following Fig. 6.2.



- Same function name with different argument list.
- Different objects are created.
- A call is given to different arg-list functions with each object.

- Same function name with different argument list.
- One object is created.
- A call is given to different arg-list function with one object.

Fig. 6.2: Function Overloading

- **Rules for overloaded functions:**
  1. The argument list of each of the function instances must be different.
  2. The compiler does not use the return type of the function to distinguish between function instances.
- The Program 6.1 explains the concept of function overloading with many objects and Program 6.2 explains the concept with one (single) object.

Program 6.1: Program for function overloading with many objects.

```
#include<iostream.h>
class test
{
   public:
   void addnum (int p, int q, int r)
    {
      cout<<"sum is"<<p+q+r<<"\n";
    }
    void addnum (float p, float q)
    {
      cout<<"sum is"<<p + q <<"\n";
    }
    void addnum(int p, double q)
    {
      cout<<"sum is"<<p + q<<"\n";
    }
};
int main()
{
   test ob1, ob2, ob3;
   ob1.addnum (5, 8, 10);
   ob2.addnum (2.5, 3.6);
   ob3.addnum (25, 3.5);
}
```

Output:

```
sum is 23
sum is 5.6
sum is 28.5
Press any key to continue . . .
```

- The difference in return type is not a consideration for function overloading.

Program 6.2: Program for function overloading with single object.

```
#include<iostream.h>
class test
{
   public:
   void addnum(int p, int q, int r)
   {
    cout<<"sum is"<<p+q+r<<"\n";
   }
```

```
    void addnum (float p, float q)
    {
     cout<<"sum is"<<p+q<<"\n";
    }
    void addnum(int p, double q)
    {
    cout<<"sum is"<<p+q<<"\n";
    }
};
int main()
{
    test ob1;
    ob1.addnum (5, 8, 10);
    ob1.addnum (2.5, 3.6);
    ob1.addnum (25, 3.5);
}
```

Output:

```
sum is 23
sum is 5.6
sum is 28.5
Press any key to continue . . .
```

6.2.2 Operator Overloading

- Operator overloading is closely related to function overloading.
- In C++, you can overload most operators so that they perform special operations relative to classes that you create.
- Actually, C++ tries to make user defined data types behave in much the same way as built-in types.
- For instance, C++ permits us to add two variables of user defined types with the same syntax that is applied to the basic types. This means that C++ has the ability to provide the operators with a special meaning for a data type. The mechanism of giving such special meanings to an operator is known as **'Operator Overloading'**.
- Operators can be overloaded by creating **'operator functions'**.
- An operator function defines the operations that the overloaded operator will perform relative to the class upon which it will work.
- An operator function is created using the keyword **'operator'**.
- A operator function's general form is:

```
return_type class_name:: operator op (arg-list)
{
    // operations
}
```

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

- For example, if you are overloading the / operator, use 'operator /'. When a unary operator is overloaded, arg-list is empty.
- When a binary operator is overloaded, arg-list will contain one parameter.
- The process of overloading involves the following steps:
  1. First, create a class that defines the data type that is to be used in the overloading operation.
  2. Declare the operator function operator op ( ) in the public part of the class. It may be either a member function or a friend function.
  3. Define the operator function to implement the required operations.
- Almost all C++ operators can be overloaded.
- Operators that can be overloaded except the following:

  . Member access operator

  .* Pointer to member operator

  :: Scope resolution operator

  ?: Conditional operator

  sizeof sizeofoperator.
- The *precedence* of an operator cannot be changed by overloading. This can lead to awkward situations in which an operator is overloaded in a manner for which its fixed precedence is inappropriate. However, parentheses can be used to force the order of evaluation of overloaded operators in an expression.
- The associativity of an operator cannot be changed by overloading.
- It is not possible to change "*arity*" of an operator (i.e. the number of operands an operator takes): Overloaded *unary operators* remain as unary operators; overloaded *binary operators* remain as binary operators. C++ only ternary operator (?:) cannot be overloaded.
- The meaning of an operator works on objects of built-in types cannot be changed by operator overloading. For example, change the meaning of how + adds two integers.
- Operator overloading only works with objects of user defined types or with a mixture of an object of a user defined type and an object of a built-in type.
- Overloading an assignment operator with an addition operator to allow statements like:

```
object 2 = object 2 + object 1
```

  Does not imply that + = operator is also overloaded to allow statements such as,

```
object 2 + = object 1;
```

- Such behaviour can be achieved by *explicitly overloading* the + = operator for that class.
- Different types of operators are:

  1. Arithmetic operators:

  +, −, /, *, %, =, ++, − −, % is called modulus operator used for getting remainder of division.

2. **Relational operators:**
   These operators are used for checking for condition. Here, L.H.S. value is compared with R.H.S. Operators are: <, >, <=, >=, = = , !=
3. **Logical operators:**
   These operators work similarly to logic gates AND, OR, NOT, XOR. Operators are: &&, || , !.
4. **Bitwise operators:**
   These operators operate on every bit of a byte. Operators are: &, |

- According to the operands required for operation all operators are divided into three categories:

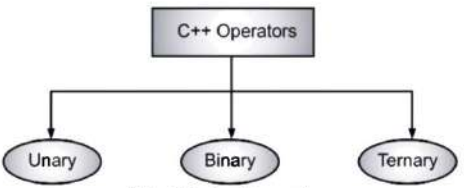


Fig. 6.3: C++ Operator

1. **Unary:**
   It requires only one operand to perform operation. For example: `i++`
   Here, i is the single operand to perform increment operation.
2. **Binary:**
   It requires two operands to perform operation. For example: `a + b;`
   Here, a and b are two operands to perform addition operation.
3. **Ternary operator:**
   It requires three operands to perform operation. Ternary operator is ?,:
   For example: `x > 5? y = 2: y = 1;`
   This line compares x value. If x > 5, y = 2. If x is not greater than y = 1.

- Overloading means assigning more than one meaning to operator.
- For example, + sign can be used for:
  1. Addition of two integers
  2. Addition of two complex numbers.
  3. Addition of two characters.
- **Steps for operator overloading are given below:**
  1. Create a member function under a public section.
  2. The member function is called as operator overloaded function so, syntax of declaration than other normal member function.

     Syntax:

```
return_type operator op( )
{
   //details of operation
}
```

     Here, `operator` is a keyword which tells it is operator overloading function and `op` is any operator, For example, +.

3. When we want to call operator overloading function the syntax is:

   ```
   operator sign object_name;
   ```

   For example: For calling + sign's operator overloaded function we will use, `+a1;`

   Here, `a1` is the object of a class where operator overloading function is declared.

6.2.3 Rules for Overloading Operators [W-18]

- Although it looks simple to redefine the operators, there are certain restrictions and/or limitations in overloading them. Some of them are listed below:

1. Only existing operators can be overloaded. New operators cannot be created.
2. The overloaded operator must have at least one operand that is of user defined type.
3. We cannot change the basic meaning of an operator. That is we cannot redefine the plus (+) operator to subtract one value from the other.
4. Overloaded operators follow the syntax rules of the original operators that cannot be overridden.
5. There are some operators that cannot be overloaded such as .*,:: etc.
6. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values. But those overloaded by means of a friend function take one reference argument.
7. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
8. When using binary operators overloaded through a member function, the left-hand operand must be an object of the relevant class.
9. Binary arithmetic operators such as +, −, *, / must explicitly return a value. They must not attempt to change their own arguments.
10. We cannot use friend functions to overload certain operators, which are listed below. However, member functions can be used to overload them.

    = Assignment operator

    () Function call operator

    [] Subscripting operator

    −> Class member access operator.

    Where, a friend cannot be used.

6.2.4 Operator Overloading Unary and Binary

1. Overloading Unary Operator:

- The operators can be overloaded using two different functions i.e. through member functions and friend functions.
- A unary operator overloaded using a member function takes no argument, whereas an overloaded unary operator declared as friend function takes one argument.

- The unary operators that can be overloaded are shown below.

Table 6.1: Unary Operator

Operators	Meaning
->	Indirect member operator
!	Logical negation
*	Pointer reference
&	Address operator
~	Ones complement
->*	Indirect pointer to member
+	Addition
-	Subtraction
++	Incrementer
--	Decrementer
-	Unary minus.

- First we will consider the unary minus (-) operator. A minus operator, when used as a unary, takes just one operand.
- We know that this operator changes the sign of an operand when applied to a basic data item. We will see here how to overload this operator so that it can be applied to an object in much the same way as is applied to an int or float variable.
- The unary minus when applied to an object should change the sign of each of its data items.

Program 6.3: Program for how the unary minus operator is overloaded.

```
#include<iostream.h>
class space
{
   int x;
   int y;
   int z;
   public:
      void getdata (int a, int b, int c);
      void display (void);
      void operator - ( );  // overload unary minus
};
void space:: getdata (int a, int b, int c)
{
   x = a;
   y = b;
   z = c;
}
```

```
void space:: display (void)
{
   cout << x << " ";
   cout << y << " ";
   cout << z << "\n";
}
void space:: operator - ( ) // defining operator - ( )
{
   x = - x;
   y = - y;
   z = - z;
}
int main ( )
{  space s;
   s.getdata (10, -20, 30);
   cout << "s:  before overloading:";
   s.display ( );
   - s;                    // activates operator - ( )
   cout << "s: After overloading:";
   s.display ( );
}
```

Output:

```
s: before overloading:10 -20 30
s: After overloading:-10 20 -30
```

- Note that, the function operator – ( ) takes no argument. Then what does this operator function do? It changes the sign of data members of the objects.
- Since, this function is a member function of the same class; it can directly access the members of the object which activated it.
- Remember a statement like:

```
s2 = - s1;
```

Will not work because, the function operator – () does not return any value.

- It can work if the function is modified to return an object.
- It is possible to overload a unary minus operator using a friend function as follows:

```
friend void operator - (space & s);   // declaration
void operator - (space & S)   // definition
{
   s.x = -s.x;
   s.y = -s.y;
   s.z = -s.z;
}
```

- Note that, the argument is passed by reference. It will not work if we pass argument by value because only a copy of the object that activated the call is passed to operator – ().
- Therefore, the changes made inside the operator function will not reflect in the called object.

Overloading of Increment Operator:

- We know that C++ supports the operator that is used for incrementing and decrementing by 1. These operators can be used either prefix or postfix.
- Generally, overloading of these operators cannot be distinguished between prefix or postfix operation. But whenever a postfix operation is overloaded, it takes a single argument along with a member function of a class object.

Program 6.4: Program to generate Fibonacci series by overloading a prefix operator.

```
#include<iostream.h>
class fibo
{
   private:
      int fib0, fib1, fib2;
   public:
      fibo(); // default constructor
      void operator ++();  // overloading declaration
      void disp();
};
fibo:: fibo()
{
   fib0 = 0;
   fib1 = 1;
   fib2 = fib0 + fib1;
}
void fibo:: disp()
{
cout<<fib2<<"\t";
 }
void fibo:: operator ++()
{
   fib0 = fib1;
   fib1 = fib2;
   fib2 = fib0 + fib1;
}
```

```
int main()
{
   fibo ob1;
   int n;
   cout<<"How many numbers you want?:";
   cin>>n;
   for(int i=0;i<=n;++i)
   {
      ob1.disp();
      ++ob1;
   }
}
```

Output:

```
How many numbers you want?:
4
1       2       3       5       8
```

2. Overloading Binary Operator:

- Binary operators are operator which require two operands to perform operation.
- We have just seen how to overload a unary operator. The same mechanism can be used to overload a binary operator.
- To add two numbers generally we use statement like:

```
C = sum (A, B)      // functional notation
```

- The functional notation can be replaced by a natural looking expression,

```
C = A + B           // arithmetic notation
```

By overloading the + operator using an operator + ( ) function.

- The binary operators overloaded through member function take one argument which is formal.
- The Table 6.2 shows the binary operators which can be overloaded. The binary overloaded operator function takes the first object as an implicit operand and the second operand must be passed explicitly.
- **The syntax for overloading a binary operator is:**

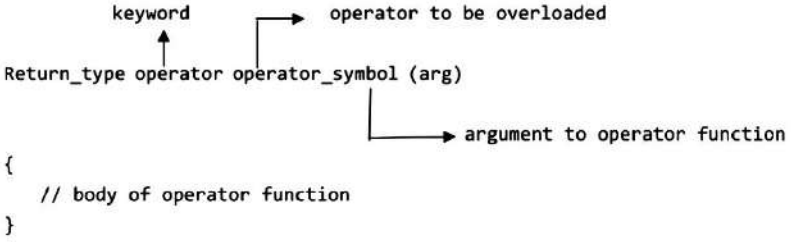


Table 6.2: Binary Operators

Operator	Meaning
[ ]	Array element reference
( )	Function call
new	New operator
delete	Delete operator
*	Multiplication
/	Division
%	Modulus
+	Addition
–	Subtraction
<<	Left shift
>>	Right shift
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal
!=	Not equal
&	Bitwise AND
^	Bitwise XOR
&&	Logical AND
|	Logical OR
=	Assignment
*=	Multiply and assign
/=	Divide and assign
%=	Modulus and assign
+=	Add and assign
–=	Subtract and assign
<<=	Shift left and assign
>>=	Shift right and assign
&=	Bitwise AND and assign
:=	Bitwise OR and assign
^=	Bitwise one's complement and assign

Program 6.5: Program for binary operator overloading.

```
#include<iostream>
using namespace std;
class integer
{
   private:
      int val;
   public:
      integer(); //constructor1
      integer(int one); //constructor2
      integer operator + (integer objb); //operator function
      void disp();
};
integer:: integer()
{
   val = 0;
}
integer::integer(int one)
{
   val = one;
}
integer integer::operator + (integer objb)
{
   integer objsum;
   objsum.val = val + objb.val;
   return(objsum);
}
void integer::disp()
{
   cout<<"value="<<val<<endl;
}
int main()
{
   integer obj1 (11);
   integer obj2 (22);
    integer objsum;
   objsum = obj1 + obj2; //operator overloading
   obj1.disp();
   obj2.disp();
   objsum.disp();
}
```

Output:

```
value = 11
value = 22
value = 33
```

6.2.5 Operator Overloading Using Friend Function

- We have already seen that private members of a class cannot be accessed through outside functions. This is possible just by using friend function concept.
- Friend functions may be used in place of member functions for overloading a binary operator. The difference is friend function requires two arguments to be explicitly passed to it while a member function requires only one.

Overloading Binary Operators using 'friend' Functions:

- The syntax for declaring and defining a friend operator function to overload binary operator is as follows:

```
class class_name
{
   public:
      friend return_type operator op(arg1, arg2);
};
return_type operator op (arg1, arg2, … argN)
{
   //function definition ……..
}
```

Program 6.6: Program for overloading binary operator using friend function.

```
#include<iostream.h>
class binopr
{
   private:
      int x;
   public:
      binopr() { } // constructor
      binopr (int); // parameterized constructor
      void disp();
      friend binopr operator + (binopr,binopr);
};
binopr:: binopr(int a)
{
   x=a;
}
void binopr:: disp()
{
   cout<<x;
}
```

```
// overloading definition with friend function
binopr operator + (binopr P, binopr S)
{
   return (P.x + S.x);
}
int main()
{
   binopr ob1, ob2, ob3;
   ob1 = binopr (15);
   ob2 = binopr (28);
   ob3 = ob1 + ob2;
   ob3.disp();
}
```

Output:

```
43
Press any key to continue . . .
```

- The overloading definition can also be written using following code:

```
binopr operator + (binopr P, binopr S)
{
   S.x = P.x + S.x;
   return(S.x);
}
```

- The implementation of overloading + operator (binary) is shown in Fig. 6.4.

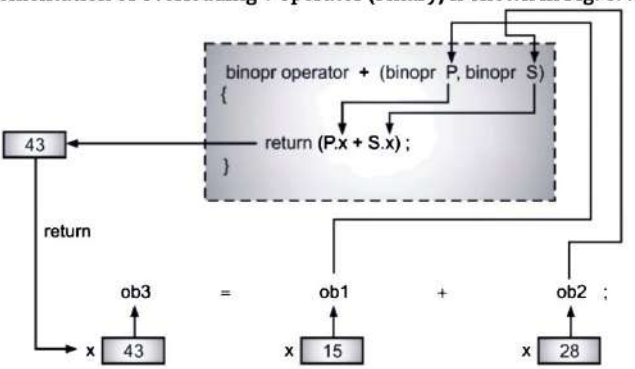


Fig. 6.4: Implementation of overloaded + operator using friend function

Overloading Unary Operators using 'friend' Functions:

- Here, the operator function is not the member function of any class. Hence, in this scenario if an operator has to be overloaded to make it work on a class object, the operator function has to be declared as a friend to that class.
- It is to be noted that, when an operator function is friend to a class, then we can call that function as a "friend operator function."
- The syntax for declaring and defining a friend operator function to overload an unary operator is as follows:

```
class class_name
{
   public:
      friend return_type operator op(arg);
};
return_type operator op(arg)
{
   //function definition .....
}
```

Program 6.7: Program for overloading unary operators using friend function.

```
#include<iostream>
using namespace std;
class OverUnaOpr
{
   private:
      int x;
      int y;
   public:
      friend void operator - (OverUnaOpr &); // prototype
      void display( );
      OverUnaOpr( )// constructor
      {
         x = 6;
         y = 7;
      }
};
void operator - (OverUnaOpr &a) // operator function definition
{
   a.x = -a.x;
   a.y = -a.y;
}
```

```
void OverUnaOpr:: display ( )
{
   cout << "x = " << x << endl;
   cout << "y = " << y << endl;
}
int main()
{
   OverUnaOpr s;
   s.display();
   -s;          //Overloading unary '-' operator
   s.display();
   return 0;
}
```

Output:

```
x = 6
y = 7
x = - 6
y = - 7
```

6.2.6 Overloading Insertion (<<) and Extraction (>>) Operators

- We have already used the objects **cin** and **cout** (pre-defined in iostream file) for the input and output of data of various types. This is possible only because the operators >> and << are overloaded to recognize all the basic C++ types.
- The >> operator is overloaded in the **istream** class and << operator is overloaded in the **ostream** class. The istream class overloads the >> operator for the standard types [int, long, double, float, char, and char * (string)].
- For example, the statement, `cin >> x;` calls the appropriate >> operator function for the istream cin defined in iostream.h. And uses it to direct this input stream into the memory location represented by the variable x.
- Similarly, the ostream class overloads the << operator, which allows the statement, `cout << x;` to send the value of x to ostream cout for output.
- Actually, istream provides the generic code for formatting the data after it is extracted from the input stream. Similarly, ostream provides the generic code for formatting the data before it is inserted to the output stream.

Program 6.8: Accept and display the information using extraction and insertion operators. [S - 19]

```
#include<iostream>
using namespace std;
class info
{
   private:
         int roll;
         char name[20];
```

```
    public:
            info()
            {
               roll = 0;
               name [0] = '0';
            }
            friend istream & operator>>(istream &, info&);
            friend ostream & operator<<(ostream &, info&);
};
istream & operator>>(istream &S, info &d)
{
    cout<<"Enter Rollno";
    S>>d.roll;
    cout<<"Enter name";
    S>>d.name;
    return S;
}
ostream & operator<<(ostream &S, info &d)
{
    S<<d.roll;
    S<<d.name;
    return S;
}
int main ()
{
    info ob1;
    cin>>ob1;
    cout<<ob1;
}
```

Output:

```
Enter rollno: 101
Enter name: Prajakta
101 Prajakta.
```

6.2.7 String Manipulation Using Operator Overloading

- C++ permits us to create own definitions of operators that can be used to manipulate the strings very much similar to the decimal numbers.
- Strings can be defined as class objects which can be then manipulated like the built-in types.

- For example:

```
string S1 = string S2 + string S3;
if(string S1 > = string S2)
string = string S1
```

- A typical string class will like this,

```
class string
{
   char * p;    /* pointer to string */
   int len;     /* length to string */
   public:
   /* member function for initialize and manipulate strings */
   ..........
   ..........
   ..........
};
```

Program 6.9: Providing a reversing operator for string class by overloading operator ~. For example: If string = C++ then output = ++C.

```
#include<iostream>
#include<cstring>
using namespace std;
class String
{
   private:
      char str[40];
   public:
       void getdata()
       {
          cout<<"Enter string to be reversed";
          cin >> str;
       }
       void operator ~ ()
       {
          strrev (str);
       }
       void display()
       {
          cout <<"Reversed string = "<< str;
       }
};
```

```
int main()
{
   String s1;
   s1.getdata(),
   ~ s1;
   s1.display();
   return 0;
}
```

Output:

```
Enter string to be reversed
MAHESH
Reversed string = HSEHAM
Press any key to continue . . .
```

Program 6.10: Overload the operator ! to find out the length of the string. [S - 18, 19; W - 18]

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class String
{
   private:
      char str[40];
   public:
      void getdata()
      {
         cout<<"Enter the string";
         cin >> str;
      }
      int operator ! ()
      {
         int l;
         l = strlen(str);
         return(l);
      }
      void display();
} s1;
void String:: display()
{
   cout <<"length of string=";
   cout << ! s1;
}
```

```
int main()
{
    s1.getdata();
    s1.display();
    getch();
    return 0;
}
```

Output:

```
Enter the string
MAHESH
length of string=6
Press any key to continue . . .
```

Program 6.11: Write a program to overload the + operator so that two strings can be concatenated. For example: s1 = abc, s2 = pqr then abc + pqr = abcpqr.

```
#include<iostream.h>
#include<conio.h>
class String
{
    char str[20]; //member variable for string input
    public:
        void input() //member function
        {
            cout<<"Enter your string: ";
            cin.getline(str,20);
        }
        void display() //member function for output
        {
            cout<<"String: "<<str;
        }
        String operator+(String s) //overloading
        {
            String obj;
            strcat(str,s.str);
            strcpy(obj.str,str);
            return obj;
        }
};
int main()
{
    String str1,str2,str3; //creating three objects
    str1.input();
    str2.input();
    str3=str1+str2;
    str3.display();
}
```

6.3 RUNTIME POLYMORPHISM [S - 19]

- Run-time polymorphism is also called as **dynamic binding** or **late binding**. Sometimes, a situation occurs where function name and prototype is same in both the base class and in the derived class. Compiler does not know what to do, which function to call. In this class appropriate member function is selected to run time. For example, virtual function.
- To achieve run-time polymorphism C++ supports a mechanism known as **virtual functions**. Dynamic binding uses the concept of pointers, i.e. it requires to use of pointer to object.

6.3.1 'this' Pointer

- Every object in C++ has access to its own address through an important pointer called 'this' pointer.
- In other words, "The member functions of each and every object have access to a pointer named 'this', which points to the object itself.
- 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.
- 'this' is a pointer that points to the object for which the function was called.
- 'this' pointer is predefined pointer variable within every class.
- 'this' pointer is a pointer pointing to the current object of specified class. "this" is a keyword in C++.
- **Syntax:**

```
class_name * this;
```

- Every object has access to its own address through a pointer called **this**. When a member function is called, 'this' pointer is automatically passed as an implicit argument to that function.
- For example: The function call,

`s.sample();` will set the pointer this to the address of the object `s`.

- One of the important application of "this" pointer is to return the object to which it is pointing. For example the following statement,

```
return * this;
```

Inside a member function will return the object that invokes a member function.

Program 6.12: Write a program to class test having data members as name of a student and marks. Accept and display information by using 'this' pointer.

```
#include<iostream>
using namespace std;
class test
{
   private:
      char name[40];
      float marks;
```

```
    public:
    void get()
    {
        cout << "\n Enter name & marks";
        cin >> name >> marks;
    }
    void display()
    {
        this -> get();
        cout << "\n Name =" << this -> name;
        cout << "\n Marks = "<< this -> marks;
    }
};
int main()
{
    test t1, t2;
    t1.display();
    t2.display();
    return 0;
}
```

Output:

```
Enter name & marks
Prajakta 80
Name =Prajakta
Marks = 80
```

6.3.2 Pointer to Objects [S - 18]

- Now we are aware of accessing class members with the help of pointers.
- We can also make a pointer to point to an object created by a class similar to that of normal variables.
- **For example:**

```
class sample
{
    int x, y;
    public:
        void get()
        {
            cin >> x >> y;
        }
```

```
        void put()
        {
          cout << x << y;
        }
    }
```

- Now lets declare a object and a pointer variable of class sample,

```
sample s;
sample * ptr;
```

- Now initialize the pointer `ptr` with the address of object `s`.

```
ptr = &s;
```

- With the help of pointer we can access the class members in the same manner as with object.
- Only, the difference is, while accessing class member with the help of objects the (.) dot operator is used where as at a time of pointer (→) arrow operator would be used.
- The following two statements are equivalent.

```
1. s.get()
2. ptr → get();
```

Program 6.13: Write a program to declare class product having data member as product name and product price. Accept and display using pointer to the object.

```
#include<iostream>
#include<conio.h>
class product
{
   protected:
      char pname[40];
      int price;
   public:
      void get()
      {
         cout << "Enter product name and price";
         cin >> pname >> price;
      }
      void display()
      {
         cout << "Product name = \t"<< pname<<endl;
         cout << "Price = \t"<< price;
      }
};
```

```
int main()
{
    product p1, p2, *ptr;
    ptr = &p1;
    p1.get();
    p1.display();
    ptr = &p2;
    p2.get();
    p2.display();
    getch();
    return 0;
}
```

Output:

```
Enter product name and price
Nokia
26500
Product name = Nokia
Price = 26500
```

6.3.3 Pointer to Derived Classes

- We can use pointers not only to the base objects but also to the objects of derived classes pointers to the objects of a base class are type compatible to the objects of derived class.
- Therefore, a single pointer variable can be made to point to objects belonging to different classes.
- For example, if B is a base class and D is derived class from B, then a pointer declared as a pointer to B can also be pointed to D.
- Consider the following declarations:

```
B * cptr;
B b;
D d;
cptr = &b;
```

- We can make `cptr` to point to the object `d` as follows:

```
cptr = &d;
```

- This is perfectly valid with C++ because `d` is an object derived from the class B. However, there is a problem in using `cptr` to access the public members of the derived class D.
- Using `cptr` we can access only those members inherited from B and not the members that originally belongs to D. In case a member of D has the same name as one of the members of B, any reference to that member by `cptr` will always access the base class members.

- Although C++ permits a base pointer to point to any object derived from that base, the pointer cannot be directly used to access all the members of the derived class. We may have to use another pointer declared as a pointer to derived type.

Program 6.14: Program for pointer to derived class.

```
#include <iostream>
using namespace std;
class BaseClass
 {
    int x;
    public:
       void setx(int i)
    {
       x = i;
    }
    int getx()
    {
       return x;
    }
 };
class DerivedClass: public BaseClass
{
    int y;
     public:
        void sety(int i)
        {
          y = i;
        }
        int gety()
        {
           return y;
        }
};
int main()
{
    BaseClass *p;                          // pointer to BaseClass type
    BaseClass baseObject;                  // object of BaseClass
    DerivedClass derivedObject;            // object of DerivedClass
    p = &baseObject;                      // use p to access BaseClass object
    p->setx(10);                          // access BaseClass object
```

```
    cout << "Base object x: " << p->getx();
    p = &derivedObject;              // point to DerivedClass object
    p->setx(99);                     // access DerivedClass object
    derivedObject.sety(88);          // can't use p to set y, so do it directl
    cout << "Derived object x: " << p->getx();
    cout << "Derived object y: " << derivedObject.gety();
    return 0;
}
```

Output:

```
Base object x: 10
Derived object x: 99
Derived object y: 88
Press any key to continue.
```

6.3.4 Virtual Functions [W-18]

- Virtual function is a member function that is declared within a base class and redefined by a derived class. A virtual function is declared by preceding the function declaration in the base class with the keyword **virtual.**

  Example: `virtual void show(); //show function is virtual.`

- When virtual functions access normally, it behaves just like any other type of class member function. However, virtual function is used to support runtime polymorphism when it is accessed via a pointer. A base class pointer can be used to point to an object of any class derived from that base. When different objects are pointed to different versions of the virtual functions are executed. Following program shows the usage of it.

Program 6.15: Program for virtual function.

```
#include<iostream.h>
class base
{
   public:
      virtual void display()
      { cout<<"This is base virtual function \n"; }
};
class derived1: public base
{
   public:
      void display()
      { cout<<"This is derived1 virtual function \n"; }
};
```

```
class derived2: public base
{
   public:
      void display()
      { cout<<"This is derived2 virtual function \n"; }
};
void main()
{
   base * p, b;     // p is base class pointer and b is object of
   derived1 d1;     base
   derived2 d2;
   p = &b;          // point to base
   p → display(); // access base function
   p → & d1;       // point to derived1
   p → display(); // access derived1 function
   p = &d2;         // point to derived2
   p → display(); // access derived2 function
}
```

Output:

```
This is base virtual function
This is derived1 virtual function
This is derived2 virtual function
```

Rules for Virtual Functions:

- The virtual function must be member of same class.
- Virtual functions are accessed by using object pointers.
- They cannot be static members i.e. must be non-static members.
- The prototype of the base class version of a virtual function and all the derived class versions must be identical.
- We can have virtual destructors but do not virtual constructors.
- We cannot use a pointer to a derived class to access object of the base type i.e. reverse is not true.
- If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class.
- A virtual function can be a friend of another class.
- Because of the restrictions and differences between function overloading and virtual function redefinition, the term overriding is used to describe virtual functions redefinition by a derived class.

6.3.5 Pure Virtual Functions [S - 18, 19]

- Most of the times, the virtual function inside the base class is rarely used for performing any task. It only serves as a placeholder. Such a functions are called do-noting functions; which are pure virtual functions.
- A pure virtual function is a virtual function that has no definition within the base class.
- **Syntax:**

```
virtual <return_type> <function_name> <arg_list> = 0;
```

- When a virtual function is made pure, any derived class must provide its own definition. If the derived class fails to override the pure virtual function, compile time error will occur. A class containing such a pure virtual function is called an **Abstract Class.**
- We can define pure virtual function as "The functions which are only declared but not defined in the base class are called as pure virtual function".

Properties of Pure Virtual Functions:

- A pure virtual function has no implementation in the base class, hence a class with pure virtual function cannot be instantiated.
- A pure virtual member function can be invoked by its derived class.
- It is just placeholder for derived class. The derived class is supposed to fill this empty function.

Program 6.16: Program for a pure virtual function.

```
#include <iostream>
using namespace std;
class number
{
   protected:
      int val;
   public:
      void set_val(int i)
      {
          val = i ;

      }
      virtual void show( ) = 0;     // pure virtual function
};
class hextype: public number
{
   public:
      void show( )
      {
      cout<<hex<<val<<"\n";
      }
};
```

```
class dectype: public number
{
   public:
      void show( )
      {
          cout<<val<<"\n";
       }
};
class octtype: public number
{
   public:
      void show()
      {
          cout<<oct<<val<<"\n";
       }
};
int main ()
{
   dectype d;
   hextype h;
   octtype o;
   d.set_val(20);
   d.show();     // displays 20 - decimal
   h.set_val(20);
   h.show();     // displays 14 - hexadecimal
   o.set_val(20);
   o.show();     // displays 24 - octal
   return 0;
}
```

Output:

```
20
14
24
```

- In the above program, as shown the base class, **number** contains an integer called **val**, the function **setval( )** and the pure virtual function **show( )**. The derived classes **hextype**, **dectype** and **octtype** inherit **number** and redefine **show( )** so that it outputs the value of **val** in each respective number base (i.e. hexadecimal, decimal or octal).

Summary

- Polymorphism means "one name, many forms". Polymorphism is the process of defining a number of objects at different classes into group and call the method to carry out the operation of the objects using different function calls.
- There are two types of polymorphism:
  1. **Compile time polymorphism:** This is also called as early or static binding selection of an appropriate function for a particular call at the compile time itself.
  2. **Run-time polymorphism:** This is also called as dynamic binding or late binding. Sometimes, a situation occurs where function name and prototype is same in both the base class and in the derived class.
- Operator overloading refers to overloading of one operator for many different purpose.
- Operator overloading is a compile time polymorphism in which a single operator is overloaded to give user defined meaning to it. Operator overloading provides a flexibility option for creating new definitions of C++ operators.
- Operator overloading is done using operator functions. Operator functions can be a member function or a friend function.
- Operator overloading is an idea of giving special meaning to an existing operator in C++ without changing its original meaning.
- Operator overloading mainly comes into picture when the C++ operators are being operated on class objects.
- "The member functions of each and every object have access to a pointer named 'this', which points to the object itself.
- 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.
- A virtual function is a member function that you expect to be redefined in derived classes. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.
- A pure virtual function is a virtual function in C++ for which we need not to write any function definition and only we have to declare it. It is declared by assigning 0 in the declaration. An abstract class is a class in C++ which have at least one pure virtual function.

Check Your Understanding

1. In operator overloading ..... in the basic meaning of operator is not allowed.
   (a) increment (b) addition
   (c) change (d) none
2. If we overload only prefix ++ operator, the postfix ++ operator .....
   (a) does not work (b) works naturally
   (c) works arbitrarily (d) works as if prefix ++ operator

3. When the compiler decides binding as overload member, it is called ..... binding.
   (a) local (b) static
   (c) dynamic (d) safe
4. Following which operator cannot be overloaded?
   (a) + (b) *
   (c) :: (d) both (a) and (b)
5. Operator in C++ can be overloaded by .....
   (a) string function (b) member function
   (c) operator function (d) both (a) & (b)
6. Which of the following is true about this pointer?
   (a) It is passed as a hidden argument to all function calls
   (b) It is passed as a hidden argument to all non-static function calls
   (c) It is passed as a hidden argument to all static functions
   (d) None of the above
7. Polymorphism means .....
   (a) Many forms (b) only one form
   (c) hiding data (d) none of them

Answers

1. (c)	2. (d)	3. (b)	4. (c)	5. (c)	6. (b)	7. (a)

Practice Questions

Q.I Answer the following questions in short:

1. What is Operator Overloading?
2. What is meant by 'this' Pointer?
3. Explain rules for operating overloading.
4. What is function overloading?
5. What is Virtual Function?

Q.II Answer the following questions:

1. Why it is necessary to overload an Operator?
2. Explain the rules for Overloading Operators.
3. Explain the usage of this Pointer.
4. What are the C++ operators that can be used for binary and unary applications?
5. Write a C++ program for a class integer which contains an integer as a data member overload the ! operator to find the factorial of an integral.
6. Write short note on: (a) Virtual Function, (b) Pure Virtual Function.
7. Explain this Pointer with example.
8. Write short note on Pointer to Objects.
9. Describe the term Pointer to derived class in detail.
10. What is Pure Virtual Function? Explain with suitable example.
11. What is Polymorphism? What is the difference between Compile Time and Runtime Polymorphism.

Q.III Define the term:

1. Polymorphism
2. Compile time polymorphism
3. Run time polymorphism
4. this pointer
5. Operator overloading

Previous Exam Questions

April 2018

1. Define polymorphism. Explain its types. [4 M]

Ans. Refer to Section 6.1.

2. Explain pointer to object with example. [4 M]

Ans. Refer to Section 6.3.2.

3. Explain pure virtual function with example. [4 M]

Ans. Refer to Section 6.3.5.

4. Define class string using operator overload "= =" to compare two strings. [4 M]

Ans. Refer to Program 6.10.

5. Trace output of the following program and explain it. Assume there is no syntax error: [4 M]

```
#include<iostream.h>
class space
{
   int x, y, z;
   public:
   void getdata(int a, int b, int c)
   {
      x = a;
      y = b;
      z = c;
   }
   void display( )
   {
      cout<< x <<" ";
      cout<< y <<" ";
      cout<< z <<" ";
   }
   void operator - ( )
   {
      x = -x;
      y = -y;
      z = -z;
   }
};
```

```
int main( )
{
   space S;
   S.getdata(10, -20, 30);
   cout<< "S:";
   S.display( );
   -S;
   cout<< "S:";
   S.display( );
   return 0;
}
```

Output: S:10 -20 30 S:-10 20 -30

October 2018

1. What is polymorphism? [2 M]

Ans. Refer to Section 6.1.

2. What is virtual function? [2 M]

Ans. Refer to Section 6.3.4.

3. Write a program to overload unary operator! (NOT). [4 M]

Ans. Refer to Program 6.10.

4. Explain rules for overloading operators. [4 M]

Ans. Refer to Section 6.2.3.

April 2019

1. Define pure virtual function. [2 M]

Ans. Refer to Section 6.3.5.

2. Explain function overloading with example. [4 M]

Ans. Refer to Section 6.2.1.

3. Write a program to overload unary operator. [4 M]

Ans. Refer to Program 6.10.

4. Explain compile time polymorphism and runtime polymorphism with example. [4 M]

Ans. Refer to Section 6.2, 6.3.

5. Create a class time which contains data members as hours, minutes and seconds. Write a C++ program using operator overloading for the following: [4 M]

(i) = = to check whether two times same or not.

(ii) >> to accept time

(iii) << to display time.

Ans. Refer to Program 6.8.

■■■

7...

Managing Console I/O Operations

Objectives...

- To understand C++ streams and C++ stream classes.
- To learn about Unformatted I/O operations and Formatted console I/O operations.
- To study Output formatting using manipulators.
- To understand user defined manipulators.

7.1 INTRODUCTION

- C++ defines its own object oriented I/O system; it has its own object oriented way of handling data input and output. These Input/Output operations are carried out using different stream classes and their related functions.
- Until now we have used the object of the istream class (cin) and object of ostream class (cout); with operators >> and << respectively. For formatting purpose we use different streams and their functions in C++.

7.2 C++ STREAMS [S - 18, 19]

- A stream is sequence of bytes.
- Stream acts either as a source from which the input data can be obtained or as a destination to which the output data can be sent.
- In C++ to manage the data flow we have different stream classes, (Refer Fig. 7.1)
- Fig. 7.1 shows stream classes for console I/O.

1. **istream class:** It is a derived class of `ios` and hence, inherits the properties of `ios`. It defines input functions such as `get( )`, `getline( )` and `read( )`. In addition, it has an overloaded member function, stream extraction operator >>, to read data from a standard input device to the memory items.
2. **ostream class:** It is a derived class of `ios` and hence inherits the properties of `ios`. It defines output functions such as `put( )` and `write( )`. In addition, it has an

overloaded member function, stream insertion operator <<, to write data from memory to a standard output device. `cerr` object is of ostream class used for displaying error messages.

3. **iostream class:** It is derived from multiple base classes, istream and ostream, which are inturn inherited from class `ios`. It supports both input and output stream operations.

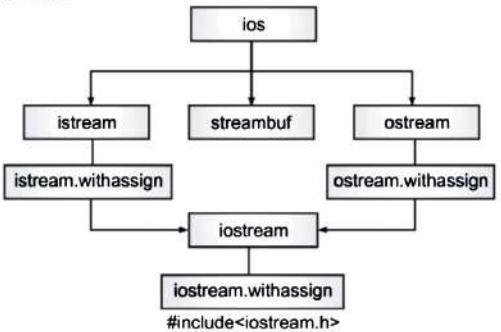


Fig. 7.1: Stream class for console I/O

- The classes' istream_withassign, ostream_withassign and iostream_withassign add the assignment operators to their parent classes.

7.3 C++ STREAM CLASSES

- The Input/Output system in C++ is designed to work with a wide variety of devices including terminals, disk and tape drives.
- The Input/Output system supplies an interface to the programmer that is independent of the actual device being accessed.
- This interface is known as stream. C++ Input/Output occurs in a stream of bytes.
- A stream is simply a sequence of bytes. It is a one way transmission path that is used to connect a file stored on a physical device such as a disk or CD-ROM, to a program.
- It acts either as a source from which the input data can be obtained or as a destination to which the output data can be sent.
- The source stream, which provides data to the program, is called the **input stream** and the destination stream that receives output from the program is called the **output stream**.
- In input operation, the bytes flow from a device (For example, a keyboard, a disk drive, a network connection) to main memory.
- In output operation, bytes flow from main memory to a device (For example, a display screen, a printer, a disk drive, a network connection). In other words, a program extracts the bytes from an input stream and inserts bytes into an output stream as shown in Fig. 7.2.
- The data in the input stream can come from the keyboard or any other storage device.
- Similarly, the data in the output stream can go to the screen or any other storage devices.

- Stream acts as an interface between the program and the Input/Output device. Therefore, a C++ program handles data independent of the devices used.

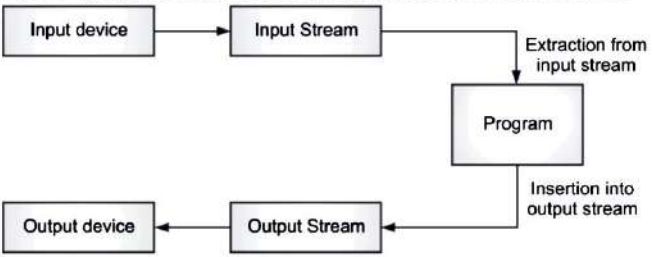


Fig. 7.2: Data Streams in C++

7.4 UNFORMATTED I/O OPERATIONS

7.4.1 Unformatted I/O Operators

- The <<, >> operators for shifting are overloaded in ostream and istream class respectively.
- The general format for reading the data from the keyboard is:

```
cin >> var1, >> var2 >> ..... >> varn;
```

Where, var1 and var2 are any, C++ predefined variables.

- This statement in the program will make the compiler to wait for the input data from the keyboard.
- The operator >> reads the data character by character and assigns it to the indicated location.
- The reading for the variable will be terminated at the encounter of a white space or a character that does not match with destination type.
- The general form for displaying data on screen is:

```
cout << data1 << data2 << ..... <<datan;
```

7.4.2 Unformatted I/O Functions

- Unformatted I/O functions are listed below:

1. get(): [W-18]

- The class istream defines a member function get() which is used to handle single character Input/Output operations at a time.
- There are two types of get() functions.
  - `get(char *);`
  - `get(void);`
  - We can use any one of the prototype to read the data from a keyboard, it is similar to that of `cin` object.

- Only the difference is that `cin` will terminates when encounters space, tab or new line character where as with get() function. We can read these all.
- **`get(char *)`:** Versions assigns the inputted character to its argument.
- **`get (void)`:** Type returns the input character.

Program 7.1: Program for get function.

```
#include<iostream>
using namespace std;
int main()
{
   char c;
   cout <<"Enter a character:";
   cin.get(c);
   cout <<"Entered character is:";
   cout <<c;
   return (0);
}
```

Output:

```
Enter a character:A
Entered character is:A
Press any key to continue . . .
```

- We can write the same program by making a use of get(void) version as.

```
#include<iostream>
using namespace std;
int main()
{
   char c;
   cout <<"Enter character:";
   c = cin.get();
   cout<<"Entered character is:";
   cout << c;
   return(0);
}
```

Output:

```
Enter a character:A
Entered character is:A
Press any key to continue . . .
```

- The value returned by the function get() is assigned to the variable c.

2. put(): [W-18]

- The class ostream defines the member function put() which is used to output a line of text on a screen character by character.
- The function put(), outputs one character at a time.
- **Syntax:**

```
cout.put(char);
```

- **Example:**

```
cout.put('s');      //Displays the character s,
cout.put(ch);       //Displays the contents of a variable ch.
```

- The variable ch must contain character value. We can also use number as an argument to the put() function like:

```
cout.put(65);       //Displays character A. Since, the ASCII value of character is A is 65.
```

Program 7.2: Program for put function.

```
#include<iostream>
using namespace std;
int main()
{
   char c;
   cout <<"Enter a character:";
   cout.put(c);
   cin.get(c);
   cout <<"Entered character is:";
   cout <<c;
   return (0);
}
```

Output:

```
Enter a character:M
Entered character is:M
Press any key to continue . . .
```

3. getline(): [S - 18]

- We can read line of text more efficiently by making a use of getline() function.
- The getline() function reads a whole line of text that ends with a newline character.
- **Syntax:**

```
cin.getline(line, size);
```

- This statement calls the built in function getline() which reads character input into the variable line.
- This function will stop further as soon as it encounters a new line character "\n" or after reading size-1 character.

- This function reads the newline character but it does not save it. Consider the following example:

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

And suppose the entered value is: `programming with C++ is fun`

- Then output would be `programming with C++`
- As we are mentioning the size as 20 so, getline() function will read only 20 characters including white spaces.
- We can also read the string using the overloaded operator >> with object `cin` as,

```
cin >> title;
```

- But `cin` can only read the strings that do not contain any white spaces, which means `cin` can read only single word at a time not a series of words line "programming with C++ is fun".

Program 7.3: Program for getline function.

```
#include<iostream.h>
#include<conio.h>
int main()
{
   int size = 40;
   char title [50];
   cout << "Enter title:";
   cin.getline(title, size);
   cout<<"Entered title is:";
   cout <<"\n" << title;
   getch();
   return(0);
}
```

Output:

```
Enter title: Computer
Entered title is:Computer
```

4. write():

- Write function is used to write a set of characters into the file.
- The write() is a member function of ostream class it is used to display entire line on the screen.
- **Syntax:**

```
cout.write (line, size);
```

Where, line is the variable of which contents have to be display and size is number of characters to be display. One important thing to note about write() function is, it does not stop displaying the characters automatically when NULL character is encountered. If the size is greater than the length of line, then it displays beyond the bounds of line.

5. **read():**

- The read function is used to read a set of characters form the file.
- **Syntax:**

```
fileObject_name.read((char *) and variable_name;
size of (variable_name));
```

Program 7.4: Program for write function.

```
#include<iostream.h>
#include<conio.h>
int main()
{
   char name [10];
   cout << "Enter name:";
   cin >> name;
   cout<<"Entered Name is:";
   cout.write (name, 10);
   getch();
   return(0);
}
```

Output:

```
Enter name: Mahesh
Entered Name is: Mahesh
Press any key to continue . . .
```

6. **peek():**
   It reads particular char from file.
7. **putback():**
   It places the control back to previous character obtained by get().
8. **ignore():**
   It skips number of characters while reading or writing in a file.
9. **sizeof():**
   It gives size of memory required for storage.
   For example: `sizeof(c);` where c is a character so it requires 1 byte of storage.

7.5 FORMATTED CONSOLE I/O OPERATIONS [S - 18]

- There are two types of functions are available in C++ for formatting the output.
- These are:
  1. IOS class functions.
  2. Manipulators.
- The IOS class contains large number of functions that would help us to format the output in number of ways.

- The IOS class functions are:
  - `width()` : To specify the required field size for displaying an output value.
  - `precision()` : To specify the number of digits to be displayed after the decimal point.
  - `fill()` : To specify a character i.e. used to fill the unused portion of a field.
  - `setf()` : To specify format flags that can control the form of output display.
  - `unsetf()` : To clear the flags specified.
- The functions used for formatting are listed below:

1. Field width:

- width( ) function is used to define the width of a field necessary for the output of an item.
- The width function sets the field width. As width is the member function, you have to use an object to invoke it.
- **Syntax:**

```
cout.width (w);
```

Where w is the field width.

- Width function can also be used with input function. It is given by,

```
cin.width (w);
```

- For example,

```
cout.width (6);
cout<<241<<18<<"\n"
```

Will give the output:

			2	4	1	1	8

- The value 241 is printed right justified in first six columns. The specification width (6) does not retain the setting for printing the number 18.
- This can be written as:

```
cout.width (6);
cout<<241
cout.width (5);
cout<<18
```

This will give the output as:

			2	4	1	.	.	.	1	8

Program 7.5: Program for the use of width function for character array.

```
#include<iostream.h>
int main ( )
{
   int w = 4;
```

```
    char string [10];
    cout<<"Enter a sentence:\n";
    cin.width(5);
    while (cin>>string)
    {
       cout.width (w++);
       cout<<string<<endl;
       cin.width (5);
    }
    return 0;
}
```

Output:

```
Enter a sentence:
ABC LMN PQR XYZ
ABC
LMN
PQR
XYZ
Press any key to continue . . .
```

2. Setting Precision:

- We can control the precision of floating point numbers i.e. the number of digits to the right of the decimal point.
- This can be done either by using **setprecision()** stream manipulator or **precision()** member function.
- The precision member function with no arguments return the current precision setting.
- **Syntax:**

```
cout.precision(d);
```

Where, d is the number of digits to the right of the decimal point.

- For example:

```
cout.precision(4);
cout<<sqrt (2)<<"\n";
cout<<3.14159<<"\n";
cout<<6.40000<<"\n";
```

- The outputs will be,

```
1.4112    (truncated value)
3.1416    (rounded to nearest value)
6.4       (no trailing zeros)
```

- We can also combine the field specification with the precision setting.

```
cout.precision (3);
cout.width (6);
cout<<2.24689;
```

- The output will be:

	2	.	2	4	7

3. Filling and Padding:

- Filling and Padding are used to fill the unused positions of the field.
- We can use fill ( ) function to fill the unused position by any desired character.
- **Syntax:**

```
cout.fill (ch);
```

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

- Consider the following example:

```
cout.fill ("*");
cout.width(10);
cout<<4280<<"\n";
```

The output will be:

*	*	*	*	*	*	4	2	8	0

- These characters are inserted as padding.

4. Formatting flags, Bit fields:

- The set flag function, setf() is used for printing a left-justified number or for getting a floating point number printed in the scientific notation.
- **Syntax:**

```
cout.setf (arg1, arg2);
```

- The arg1 is one of the formatting flags defined in class ios.
- The formatting flags specify the format action required for the output.
- Arg2 is another ios constant, which is also known as bit field, specifies the group to which formatting flag belongs.
- There are three bit fields and each has a group of format flags which are mutually exclusive.
- Flags and bit fields for setf( ) functions are given in following table.

Table 7.1: Flags and bit fields for setf( ) functions

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

- Consider the following example:

```
cout.fill ('*');
cout.self (ios:: left, ios::adjustfield);
cout.width (12);
cout<<"TONY"<<"\n";
```

- The output will be:

T	O	N	Y	*	*	*	*	*	*	*	*

- When, we print the number trailing with zero, they will be truncated.
- For example, if we want to print 312.42, 29.00, 18.20, with seven positions and three digit precision, the output will be:

		3	1	2	.	4		2

						2	9

				1	8	.	2

- To remove such problems, i.e. if we want to display trailing zeros, we can use the function **setf ()** with the flag ios:: showpoint as a single argument. It is written as:

```
cout.setf (ios:: trailing zero);
```

- This function will display trailing zeros and trailing decimal point. Similarly, we can also display plus sign before the number. The function used for this is,

```
cout.self (ios:: for sign);
```

- Consider the following example:

```
cout.setf (ios::showpoint);
cout.setf (ios::showpos);
cout.precision (3);
cout.setf (ios::fixed, ios::floatfield);
cout.setf (ios::internal, ios::adjustfield);
cout.width (10);
cout<<100.5 << "\n";
```

- The output will be:

+			1	0	0	.	5	0	0

- The flags showpoint and showpos do not have any bit fields and therefore are used as single arguments in setf().

- This is possible because the setf() has been declared as an overloaded function in class IOS. Some other flags that do not have bit field are showbase, uppercase, skipws, unitbuf, stdio.

  `ios:: showbase` helps to use base indicator on output.

  `ios:: uppercase` helps to use uppercase letters.

  `ios:: skipws` skips white space on input.

  `ios:: unitbuf` flush all streams after insertion.

  `ios:: stdio` flushes stdout and stdin after insertion.

7.6 OUTPUT FORMATTING USING MANIPULATORS

- Manipulators are special functions that can be included in the I/O statements to alter the format parameters of a stream.
- To access these manipulators, the file **iomanip.h** has to be included in the program.
- Manipulators perform formatting tasks. They provide the same features as IOS member functions and flags. But some manipulators are more convenient to use. We can concatenate several manipulators together.

Table 7.2: Manipulators and equivalent I/O functions

Manipulators	Equivalent I/O function
setw()	width()
setprecision()	precision()
setfill()	fill()
setioflags()	setf()
resetioflags()	unsetf()

1. Setting of field width:

- setw() is used to define the width of field necessary for the output for a variable.
- For example: `setw(6);` This function will reserve 6 digits for a number.

```
setw(6);
cout << 1234;
```

Output:

		1	2	3	4

2. Setting precision for float numbers:

- We can control number of digits to be displayed after decimal point for float numbers by using setprecision() function.

```
setprecision(2);
```

- This function will display only two digits after a decimal point.
- **For example:**

```
setprecision(2);
cout<<3.14159;
```

Output: `3.14`

3. Filling of unused positions:

- Filling of unused positions of the field can be done by using setfill().

```
setfill('*');
```

- Here, unused positions will be filled by using * sign.

```
setw(6);
setfill ('*');
cout << 1234;
```

Output:

*	*	1	2	3	4

4. setbase():

- It is used to show the base of a number, for example:

```
setbase(10)
```

7.6.1 User Defined Manipulators

- The general form for creating a manipulator without any argument is,

```
ostream and manipulator (ostream and output)
{
   - - - - - - - - - -
   - - - - - - - - - -      //code
   - - - - - - - - - -
   return output
}
```

Here, manipulator is the name of manipulator under creation. Consider the following example which defines a manipulator called money that displays 'dollor'.

```
ostream and money (ostream and output)
{
   output <<"Dollors ($)";
   return output;
}
```

- Now the statement,

```
cout << 400 << money;
```

Will display the output as,

```
400 Dollors ($)
```

Summary

- A stream is sequence of bytes.
- Stream acts either as a source from which the input data can be obtained or as a destination to which the output data can be sent.
- **istream class:** It is a derived class of *ios* and hence, inherits the properties of *ios*. It defines input functions such as *get( ), getline( )* and *read( )*.
- **ostream class:** It is a derived class of *ios* and hence inherits the properties of *ios*. It defines output functions such as *put( )* and *write( )*.
- **iostream class:** It is derived from multiple base classes, istream and ostream, which are inturn inherited from class *ios*.

- The source stream, which provides data to the program, is called the **input stream** and the destination stream that receives output from the program is called the **output stream.**
- The <<, >> operators for shifting are overloaded in ostream and istream class respectively.
- The class istream defines a member function get() which is used to handle single character Input/Output operations at a time.
- The class ostream defines the member function put() which is used to output a line of text on a screen character by character.
- The getline () function reads a whole line of text that ends with a newline character.
- Write function is used to write a set of characters into the file.
- The read function is used to read a set of characters form the file.
- width ( ) function is used to define the width of a field necessary for the output of an item.
- We can use fill ( ) function to fill the unused position by any desired character.

Check Your Understanding

1. The stream that supplies data to a program is called ..... stream.
   (a) input (b) output
   (c) I/O (d) none
2. The stream that receives data from the program is called ..... stream.
   (a) input (b) output
   (c) I/O (d) none
3. The read() and write() functions handle data in ..... form.
   (a) text (b) binary
   (c) character (d) none
4. The ios::ate mode allows us to write data anywhere in the file.
   (a) true (b) false
5. The functions write( ) and get( ) are similar.
   (a) true (b) false
6. The output pointer is called put pointer.
   (a) true (b) false
7. The programmer must create cin, cout, cerr, clog object explicitly.
   (a) true (b) false

Answers

1. (a)	2. (b)	3. (b)	4. (b)	5. (b)	6. (a)	7. (b)

Practice Questions

Q.I Answer the following questions in short:

1. What is a Stream? Enlist various Stream Classes.
2. What are the operators used by unformatted I/O?
3. What is meant by Manipulators?
4. List various unformatted I/O operations.
5. Differentiate between put() and write() function.

Q.II Answer the following questions:

1. Describe various unformatted I/O operations.
2. Explain the following:
   (i) getline()
   (ii) setw()
3. Explain setf() in detail.
4. Write short note on: User Defined Manipulators.
5. Explain the following function:
   (i) read
   (ii) write
6. What are the IOS class function? Explain them.
7. Write short notes on:
   (i) Streams
   (ii) Manipulators.

Q.III Define the term:

1. Stream
2. Manipulators

Previous Exam Questions

April 2018

1. What is stream concept in C++? **[2 M]**

Ans. Refer to Section 7.2.

2. What is the difference between cin and getline( )? **[2 M]**

Ans. Refer to Section 7.4.2.

3. Explain any four formatted input/output functions. **[4 M]**

Ans. Refer to Section 7.5.

October 2018

1. Write uses of get( ) and put( ). [2 M]

Ans. Refer to Section 7.4.2.

April 2019

1. What is stream? Explain a stream class hierarchy. [4 M]

Ans. Refer to Section 7.2.

■■■

8... Working with Files

Objectives...

- To learn Stream Classes for File operations.
- To study different File operations.
- To understand File updating with random access.
- To learn Error handling during file operations.
- To study command line argument.

8.1 INTRODUCTION [S - 18]

- It is difficult to handle large amount of data only with use of keyboard and screen. So, we need to store data on secondary storage device as data structures called files.
- A computer system stores programs and data in secondary storage in the form of files. A file is a collection of related information.
- A file is a sequence of bits, bytes, lines or records whose meaning is defined by its creator and user. A file is named and is referred to by its name.
- To define a file properly, it is necessary to consider the operations which can be performed on files. The Operating System (OS) provides most of the essential file manipulation services such as create, open, write, read, rewind, close and delete.
- A program typically involves data communication between the console and the program or between the files and program or even both. The program must at least perform data exchange between processor and main memory.
- Computer programs are written to store the data (a write operation on file) and to read the data (a read operation on file).
- A computer program involves following two types of data communication:
  - Data transfer between the keyboard/monitor (console) and the program.
  - Data transfer between the program and the disk file.
- Fig. 8.1 shows the concept of data communication.

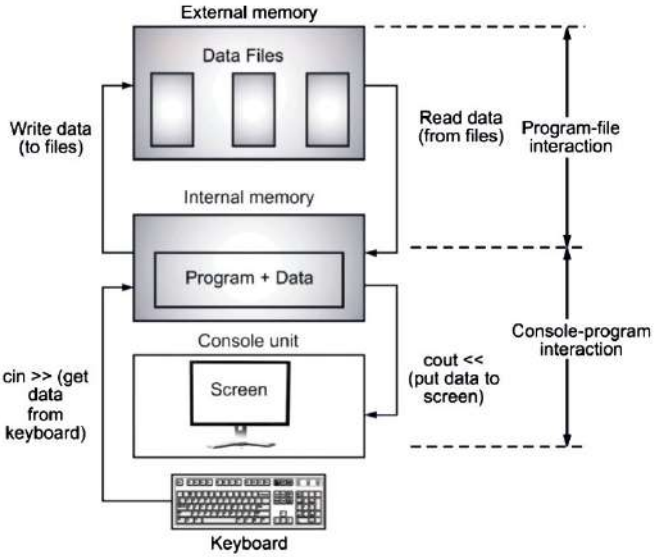


Fig. 8.1: Console Program and File Interaction

- C++ uses the concept of stream and stream classes to implement its I/O operations with the console and disk files.
- Files are either text files or binary files. A text files can be created using sophisticated word processor. Text files can be easily read or printed. Binary files are data files and program files.
- Binary files can not be easily read or write but size of binary file is smaller than that of text file. The speed at which programs can read data from binary file is very high as compare to text file.
- In this chapter, we will discuss various methods available for storing and retrieving the data from files. Using files we can transfer data from program to file on secondary storage or vice versa.

8.2 STREAM CLASSES FOR FILE OPERATIONS

- The file handling techniques of C++ support file manipulation in the form of stream objects. The stream object `cin` and `cout` are used extensively to deal with the standard input and output devices.
- The C++ input/output system support a hierarchy of classes that are used to manipulate both the console and disk files, called stream classes.
- The input/output system of C++ has a group of classes that defines all the file handling functions. These classes are:
  - `ifstream`: For handling input files.
  - `ofstream`: For handling output files.
  - `fstream`: For handling files on which both input and output can be performed.

- Input stream is declared as a class ifstream. It is derived from istream. Output stream is declared as a class ofstream. It is derived from ostream. Input/Output stream should belong to fstream class. It is derived from iostream.
- The hierarchy of C++ file stream classes is shown in Fig. 8.2

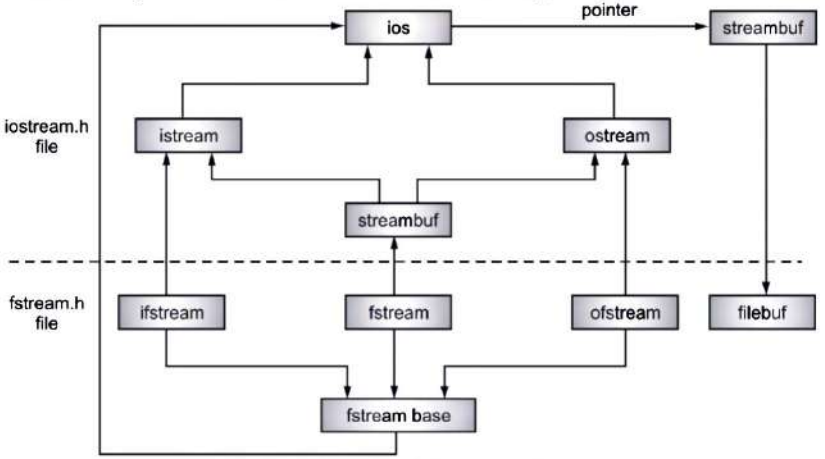


Fig. 8.2: Hierarchy of File Stream Classes

- The classes ifstream, ofstream, and fstream are designed exclusively to manage the disk files and their declaration exists in the header file fstream.h. To use these classes, include the following statement is used in the program,

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

- C++ views each file as a sequence of bytes. Each file ends either with an end of file marker or at a specific byte number. For file processing iostream.h and fstream.h must be included. fstream is used for read (ifstream) and write (ofstream).
- The functionalities of istream, ostream and iostream are inherited to ifstream, ofstream and fstream respectively. Details of file stream classes are given in the following table.

Table 8.1: Shows the details of file stream classes

Class	Contents
`filebuf`	Its purpose is to set the file buffers to read and write. Contains close() and open() as its members.
`fstreambase`	It is the base class of fstream, ifstream and ofstream classes. Contains close() and open() functions.
`ifstream`	It provides input operations. Contains open() with default input mode. Inherits the functions get(), getline(), read(), seekg() and tellg() from istream.

contd. ...

`ofstream`	It provides output operations. Contains open() with default output mode. Inherits put(), seekp(), tellp() and write() from ostream.
`fstream`	It allows simultaneous input and output operations. Contains open() with default input mode. Inherits all the functions from istream and ostream classes through iostream.

Methods for File Stream Classes:

- To associate a file with a stream, we can provide the file name when initializing a file stream object or we can first create a file stream object, then use the **open()** method to associate the stream with a file.
- The **close()** method terminates the connection between a stream and a file. The class constructors and the open() method take an optional second argument that provides the file mode.
- The file mode determines such things as whether the file is to be read and/or written to whether opening a file for writing truncates it or not, whether attempting to open a non-existing file is an error or not and whether to use the binary or text mode.
- The I/O class library provides a variety of useful methods. The istream class defines versions of the **extraction operator (>>)** that recognize all the basic C++ types and that convert character input to those types.
- The **get()** family of methods and the getline() method provide further support for single-character input and for string input.
- Similarly, the ostream class defines versions of the **insertion operator (<<)** that recognize all the basic C++ types and that convert them to suitable character output. The **put()** method provides further support for single-character output.
- A text file stores all information in character form. For example, numeric values are converted to character representations. The usual insertion and extraction operators, along with **get()** and **getline()**, support this mode.
- A binary file stores all information using the same binary representation the computer uses internally. Binary files store data, particularly floating point values, more accurately and compactly than text files, but they are less portable. The **read()** and **write()** methods support binary input and output.
- The **seekg()** and **seekp()** functions provide random access for files. These class methods let us to position a file pointer relative to the beginning of a file, relative to the end or relative to the current position. The **tellg()** and **tellp()** methods report the current file position.

8.3 FILE OPERATIONS

- There are various operations available on data files:
  1. Opening and closing files.
  2. Reading and writing files.
  3. Detecting End of file.
  4. Updating of file.

8.3.1 Opening a File [S - 18]

- Generally, if we are using files in the program, we must think about the name of file, the structure of file and opening mode of file.
- The file name is a string with at the most *eight* characters.
- It can have optional three characters for extension and both i.e. file name and extension are separated by period (•).
- To open a file:
  1. File stream is created and then linked to the file, which has two part: primary and extension.

     For example: `x.cpp` – Here x, is a filename and .cpp is extension.
  2. A filestream is defined by using classes ifstream, ofstream and fstream which are present in header file fstream.h. These classes are used depending on operations read or write.
  3. When a file is opened for write only, a new file is created if there is no file exist of that name.
- When we want to write certain text or information, the file should be created and opened, and even if we want to read the contents from the file, the file should be opened.
- In C++, a file can be opened in two ways:
  1. Opening file using constructor.
  2. Opening file using open( ) member function.

8.3.1.1 Opening File using Constructor

- In order to perform file operations, the file should be open. As we know, constructor is used to initialize an object while it is being created.
- Here, a constructor is used to initialize the file name which is used with the file stream object.
- The creation of file using file stream object involves the following steps:
  1. **Create a file stream object using stream classes as follows:**

```
ifstream ob1 ;        //input
ofstream ob2;         //output
fstream ob;           //input and output
```

  2. **File stream is identified by a file name:**

     For example, `ifstream ob1("stud.dat");`
- This statement declares **ob1** as an **ifstream** object and attaches it to the file **stud.dat** for reading (input) as shown in Fig. 8.3.

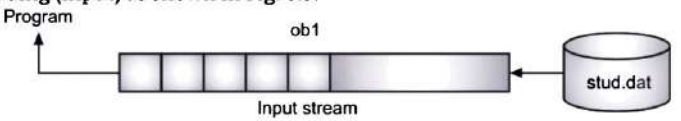


Fig. 8.3: Input Stream Object

- The statement `ofstream ob2("stud.dat");` opens a file in output mode i.e. in write mode. After opening a file, it is attached to output stream ob2 as shown in Fig. 8.4.

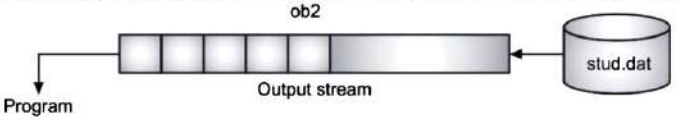


Fig. 8.4: Output Stream Object

Program 8.1: Program for open file in output mode.

```
#include<fstream.h>
#include<iostream.h>
int main()
{
   ofstream ob1("stud.dat");
   ifstream ob2 ("stud.dat");
   char str[80];
   //writing string to the file
   ob1<<"This is student database";
   //Read string from the file
   while (ob2)
   {
      ob2.getline(str, 80);      //read string using ob2
      cout<<str;                 //write on console
   }
   return 0;
}
```

Output:

```
This is student database
```

- Here ob1, ob2 are objects. Two different streams are working on the same file. This is shown in Fig. 8.5 where reading and writing are the operations performed on **stud.dat**.

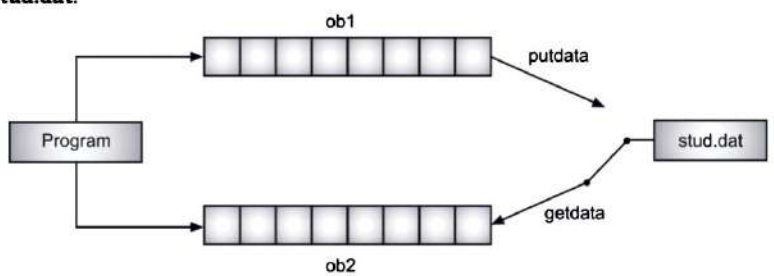


Fig. 8.5: Two file streams working on stud.dat

- When a stream object expires, the connection with a file is closed automatically i.e. when the program terminates, compiler automatically closes a file. Here, to print the message into file we use the statement.

```
ob1 << "This is student database";
```

8.3.1.2 Open File Explicitly using Open( )

- The file is open explicitly using open() function instead of constructor.
- **The syntax for opening a file is:**

```
stream_class   stream_object;
stream_object.open ("filename");
```

For example:

```
ofstream ofile;                  //create stream for output
ofile.open("stud");              //connect stream to stud
--------------
--------------
ofile.close();                   //close the file stud
```

- The stream is connected to only one file at a time. Hence, using same stream object we cannot open many files simultaneously. If we want to open two files, think first file should be close before we open the second file. If we want to open many files same time then we can use different stream_objects to open files.

For example:

1.
```
ofstream ofile;
ofile.open ("Teacher");
..........
..........
ofile.close();
ofile.open("stud");
..........
..........
ofile.close();
```

2.
```
ofstream ofile, ofout;
ofile.open("Teacher");
ofout.open("stud");
..........
..........
```

Program 8.2: Program for opening two files with same stream and storing the contents in it.

```
#include<iostream.h>
#include<fstream.h>
int main()
{
   ofstream outfile;
```

```
    outfile.open("Empname");
    outfile<<"Ganesh\n";
    outfile<<"Mahesh\n";
    outfile<<"Harish\n";
    outfile<<"Dinesh\n";
    outfile.close();                    //closing the file
    outfile.open("ACC");                //opening the file
    outfile<<"001";
    outfile<<"022";
    outfile<<"333";
    outfile<<"444";
    outfile.close();
    //Reading from a file
    char line[20];
    ifstream infile;                    //create input stream
    infile.open("Empname");
    cout<<"Name of Employees:\n";
    while(infile)
    {
       infile>>line;                    //or infile.getline(line,20);
       cout<<line<<"\n";
    }
    infile.close();
    infile.open("ACC");
    cout<<"\n the account numbers:\n";
    while(infile)
    {
       infile>>line;
       cout<<line<<"\n";
    }
    infile.close();
}
```

Output:

```
Name of Employees
Ganesh
Mahesh
Harish
Dinesh
The account numbers
001
022
333
444
```

More about open( ): File Modes:

- C++ provides a mechanism of opening a file in different modes where second argument is explicitly passed.
- The **syntax** is:

```
stream_object.open("filename", openingmode);
```

- Here, the open( ) function takes two arguments, the first argument is file name and second argument is used for the file mode or file mode parameter, which are shown in Table. 8.1. We know that, ios::in & ios::out are default values for opening of ifstream and ofstream.
- The file modes are shown in Table. 8.2.

Table 8.2: File Open Modes

Parameter	Meaning
ios::ate	Go to end-of-file on opening.
ios::app	Append to end-of-file or all writes occurs at end of file.
ios::in	open file for reading only.
ios::out	open file for writing only.
ios::binary	Binary file (open file is binary mode).
ios::trunc	Delete contents of the file if it exists.
ios::nocreate	open fails if file does not exist.
ios::noreplace	open files if the file already exists.

- The various file modes in above table are explained below:
  1. **ios::ate:** It is used to sets a pointer to the end-of-file at opening. It allows to add or modify the existing data.
  2. **ios::app:** It is used to sets a pointer to the end-of-file similar to ate, but it add the data to the end of file.
  3. **ios::in:** This mode should be specified for input files.
  4. **ios::out:** This mode should be specified for output files.

     For the fstream class, mode should be explicitly specified.

     For example:
     ```
     fstream f;
     f.open("Emp", ios::in | ios::out);
     ```
  5. **ios::binary:** It is the most used mode for large applications and databases as compared to text file reading and writing.
  6. **ios::trunc:** It is a default mode if the file is opened for output.
  7. **ios::nocreate:** It is relevant with direct access files.
  8. **ios::noreplace:** This mode not allow us to open a file with the same name hence avoid destroying the previous file.
  9. We can separate more than one modes using bitwise OR or | symbol.

     For example, `out.open("Employee", ios::in | ios::app)`

Program 8.3: Program for ios::trunc file open mode.

```
#include<fstream.h>
#include<iomanip.h>
int main()
{
   float a = 122.25, b = 27, c = 33.5;
   ofstream fout;
   fout.open("num.data", ios::trunc);
   fout<<setw(6)<<a<<endl;
   fout<<setw(6)<<b<<endl;
   fout<<setw(6)<<c<<endl;
}
```

Output:

```
122.25
27.00
33.50
```

8.3.2 Closing a File

- To close a file we should use the input and output stream object name.
- For example,

```
ob1.close();
ob2.close();
```

- Here, the member function close() is used to close a file which has been open either for reading or writing or both purposes. The close() function is not contain any arguments. It is called explicitly as above. It also called automatically by the destructor function.
- Program 8.4 uses a single file for both writing and reading the data. First it takes data from the keyboard and writes it to the file. After writing is completed, the file is closed. The program again opens the same file, reads the information already written to it and displays the same on the screen.

Program 8.4: Program uses single file for both writing and reading data.

```
#include<iostream.h>
#include<fstream.h>
int main()
{
   ofstream ofile ("Employee");
   cout<<"Enter emp.name";
   char name[20];
   cin>>name;
   ofile <<name<<"\n";  //write to file employee
```

```
    cout<<"Enter salary of emp";
    double salary;
    cin>>salary;
    ofile<<salary<<"\n";
    ofile.close();
    ifstream ifile ("Employee");
    ifile>>name;
    ifile>>salary;
    cout<<"employee name:"<<name<<"\n";
    cout<<"employee salary:"<<salary<<"\n";
    ifile.close();
}
```

Output:

```
Enter emp name
Mahesh
Enter salary of emp
26500
employee name: Mahesh
employee salary:26500
Press any key to continue . . .
```

- When a file is opened for write only, a new file is created if there is no file exist of that name. If a file is exists already, then its contents are deleted.

8.3.3 Detecting the End-of-File

- When file is read, it is necessary to detect the end-of-file.
- The eof( ) function is used to check whether, file pointer is reached at end of file or not.
- It is member function of **ios** class. If eof is detected, then function return a non-zero value otherwise returns zero.
- **Syntax:**

```
ifstream in1;
in1.open("Hindiname");
while (! in1.eof())
{
    ----------------
}
```

- We can also write,

```
if(in1.eof()!=0) {exit (1);}
```

- Here, if end-of-file is detected then function returns non-zero value. This is illustrating in program 8.5.

Program 8.5: Program for eof().

```
#include<iostream.h>
#include<fstream.h>
#include<stdlib.h>
int main()
{
   char line[20];
   ifstream ifile1, ifile2;
   ifile1.open("Employee");
   ifile2.open("ACC");
   for(int i=1;i<=5;i++)
   {
      if(ifile1.eof()!=0)
      {
       cout<<"Exit from Employee";
       exit(1);
      }
      ifile1>>line;
      cout<<"The account no. of employee"<<line<<"is";
      if(ifile2.eof()!=0)
      {
       cout<<"exit from ACC";
       exit(1);
      }
      ifile2>>line;
      cout<<line<<"\n";
   }
}
```

Output:

```
The account no. of employee Ganesh is 001
The account no. of employee Mahesh is 022
The account no. of employee Harish is 333
The account no. of employee Dinesh is 444
```

8.3.4 Updating a File

- We can add a new item, deleting an item, display an item, modifying (updating) an item in a program.

Program 8.6: Program illustrate the concept of updating files.

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main()
{
   int k=0;
   string line;
   string find;
   char name[25];
   int id=0;
   float gpa=0;
   ofstream myfile;
   myfile.open("data.txt");
   while(k!=3){
   cout<<"press 1 for adding data"<<endl;
   cout<<"press 2 for update "<<endl;
   cin>>k;
   if(k==1)
   {
      cout<<"enter ID "<<endl;
      cin>>id;
      cout<<"enter Name"<<endl;
      cin>>name;
      cout<<"enter GPA "<<endl;
      cin>>gpa;
      myfile<<name<<endl;
      myfile<<id<<endl;
      myfile<<gpa<<endl<<endl<<endl;
   }
   if(k==2)
   {
      cout<<"name u want to update "<<endl;
      cin>>find;
      ifstream file;
      file.open("data.txt");
      while (!file.eof() && line!=find)
      {
      getline(file,line);
      }
```

```
        cout<<"enter ID "<<endl;
        cin>>id;
        cout<<"enter Name"<<endl;
        cin>>name;
        cout<<"enter GPA "<<endl;
        cin>>gpa;
        myfile<<name<<endl;
        myfile<<id<<endl;
        myfile<<gpa<<endl<<endl<<endl;
    }
    if(k==3){
    myfile.close();
    }
    }
    return 0;
}
```

8.4 FILE POINTERS AND THEIR MANIPULATORS

- To understand the working of files, (binary and text) we have to study 'file pointer'. When we open any file, the operating system maintains a pointer for that file.
- There are two pointers associated with each file, called file pointers. These are get pointer (input pointer) and put pointer (output pointer). These pointers are move through a file when read and write operations are perform.
- The get pointer points to the current reading operation and put pointers points to current writing operations.
- When the file is opened in read only mode the input pointers is initalised to point to the beginning of the file. So we can read file from start which is shown in Fig. 8.6.

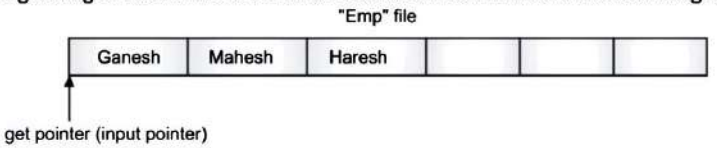


Fig. 8.6: Read Mode

- When the file is open in write only mode, then output pointer is set to beginning of the file for writing. If file is already exists then the existing contents are deleted as shown in Fig. 8.7.

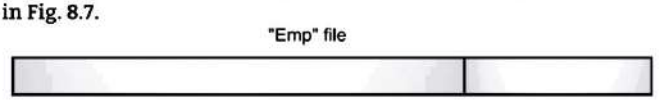


Fig. 8.7: Write Mode

- When the file is open is append mode, then the new data is added at the end of the file and output pointer are set to end of the file as shown in Fig. 8.8.

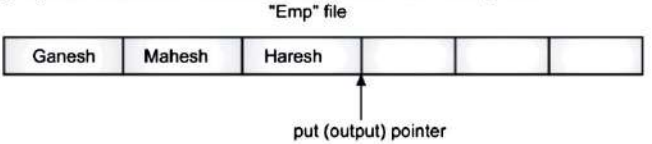


Fig. 8.8: Append Mode

- This is how the default actions take place with input and output pointers.

8.4.1 Functions for Manipulation of File Pointers

1. **seekg( ):**
   - This is member function of ifstream. It moves get pointer (input) to a specified location.
   - **Syntax:** `seekg (position);`
   - **For example:** `in.seekg(20);` Where in is ifstream object
   - It moves the file pointer to byte number 20. i.e. the pointer points to the 21st byte in the file since first byte is zero.
2. **seekp( ):**
   - It is member function of ofstream.
   - It moves put pointer to a specified location.
3. **tellg( ):** [W-18]
   - It is member function of ifstream. It gives current position of a get pointer.
   - **For example:** `long n = in.tellg( );`

     Here, variable n stores the position of get pointer.
4. **tellp( ):** [W-18]
   - It is member function of ofstream. It gives the current position of put pointer.
   - For Example:

```
out.open ("stud", ios::app);
long p = out.tellp();
```

   - Here, the output pointer is moved to the end of the file "stud" and the value of P gives the number of types in the file. We also specify the offset with seekg() and seekp() functions.
   - **Syntax:**

```
seekg (offset, pointer direction);
seekp (offset, pointer direction);
```

- Where, offset is the number of bytes the file pointer is to be moved from the location specified by `pointer_direction`. The `pointer_direction` is one of the following defined in the ios class:

`ios::beg`	Seek from beginning of the file.
`ios::cur`	Seek from current location.
`ios::end`	Seek from end of file.

- For example, [S - 18]

(i) `in.seekg(o, ios::beg);` //Go to start

(ii) `in.seekg(m, ios::beg);` //move to $(m+1)^{th}$ byte in the file

(iii) `in.seekg(-m, ios::endl);` //go backward by m byte from the end

(iv) `in.seekpg(m, ios::cur);` //go forward m byte from current position.

(v) `in.seekg(-10, ios::end);` //moves the get pointer 10 bytes backward from the end of the file.

Program 8.7: Functions for manipulation of file pointer.

```
#include<fstream.h>
#include<iostream.h>
int main()
{
   ifstream infile;
   infile.open("stud",ios::in "ios::binary);
   if(!infile)
   {
      cerr<<"Error opening"<<endl;
      exit(0);
   infile.seekg (o, ios::end);
   cout<<"File size ="<<infile.tellg();
   infile.close();
   return 0;
}
```

Output:

```
File size = 371
```

- Here, keyword "cerr" is used for displaying error.

8.5 FILE UPDATING WITH RANDOM ACCESS

- Random access means moving the file pointer directly to any location in the file instead of moving it sequentially.
- The random access technique is often used with the database files.

- Fig. 8.9 shows the classification of files.

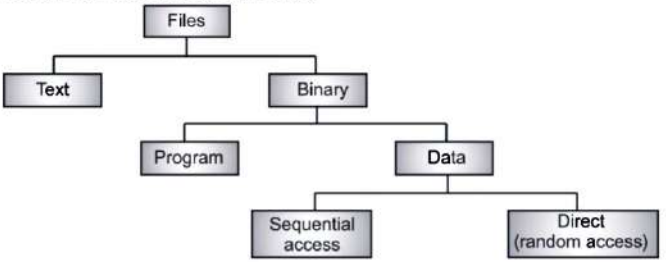


Fig. 8.9: Classification of Files

- In sequential access, we must open a file for reading and use file pointer to access records one by one sequentially.
- Binary data files can be accessed in direct or random manner. These files should be opened in binary mode.
- Random access performs the operations on file such as adding a new item, displaying the contents of a file, deleting an existing item, modifying an existing item, read and write.
- These requires that a file pointer to be move to an absolute position as specified by the parameter move.
- This can performed by using seekg( ), seekp( ), tellg( ), tellp( ), read( ) and write( ) functions. We use the following formulae in program.
- **Formula:**
  1. To find size of object (*l*),
     ```
     int l = size of (object);
     ```
  2. To find the location at which m^{th} object is stored,
     ```
     int location = m * sizeof (object);
     ```
     or `int location = m * l;`
  3. To find total number of objects in file,
     ```
     int n = filesize / sizeof (object);
     ```

Program 8.8: Program for random access approach.

```
//Basic File handling with add, modify, display operations
#include<iostream.h>
#include<fstream.h>
//using namespace std;
class student
{
   int roll;
   char name[20];
   public:
```

```
void getdata()
{
   cout<<endl<<"Enter Roll no:";
   cin>>roll;
   cout<<end1<<"Enter Name:";
   cin>>name;
}
void putdata()
{
   cout<<end1<<"Roll no: "<<roll;
   cout<<end1<<"Name: "<<Name;
};
int main()
{
  ifstream f;
  int n,i;
  char ch;
  student s[5],s1[5];
  clrscr();
  f.open("temp.txt",ios::out   ios::in);
  cout<<"writing data into file"
  for (i=0;i<2;i++)
  {
     s[i].getdata();
     f.write((char*)&s[i],sizeof(s[i]));
  }
  f.seekg(0);
  cout<<"reading from file & display on console"
  for(i=0;i<2;i++)
  {
     f.read((char*)&s[i],sizeof(s1[i]));
     s1[i].putdata();
     cout<<endl<<endl;
  }
  f.close();
  cout<<endl<<endl<<"Do you wish to update the file?(Y/N):";
  cin>>ch;
```

```
        cout<<"ADDING A RECORD"
        if(ch=='Y'||  ch=='y')
        {
           f.open("temp.txt",ios::app || ios::out);
           s[i].getdata();
           f.write((char*) &s[i],sizeof(s[i]));
           f.close();
           clrscr();
           f.open("temp.txt",ios::in);
           f.seekg (0, ios::beg);
           cout<<"Updated file is::"<<endl;
           for(i=0;i<3;i++)
           {
              f.read ((char*)&s1[i],sizeof(s1[i]));
              s1[i].putdata();
              cout<<endl;
           }
           cout<<"finding filesize & no. of objects"
           int l = f.tellg();
           int m = l/size of (s1[pi]);
           cout<<"no.of objects="<<m<<endl;
           cout<<"file size = "<<l<<"bytes"<<endl;
           f.close();
        }
        else
         f.close():
        return 0;
     }
```

Output:

```
Writing data into file
   200Riya  //file contents
   201Rucha
reading from file and display on console
   200Riya
   201Rucha
Do you wish to update the file? (Y/N): Y
ADDING A RECORD
   202Deepa
```

```
updated file is:
    200Riya
    201Rucha
    202Deepa
finding file size & no. of objects
no. of objects = 3
file size = 9 bytes
```

8.6 ERROR HANDLING DURING FILE OPERATIONS [W - 18, S - 19]

Errors Handling:

- When we use files different errors occurs such as:
  1. A file to be opened does not exist.
  2. Invalid filename.
  3. A filename used for a new file may already exist.
  4. Insufficient disc space.
- Hence, once the file is opened it has to be closed and reopen is possible using same stream.
- We can use following functions for error handling during file operation:
  - clearerr function
  - feof function
  - ferror function
  - perror function

1. The clearerr function:

- The clearerr function clears the end-of-file and error indicators for the stream pointed to by stream.

```
#include <stdio.h>
void clearerr(FILE *stream);
```

2. The feof function:

- The feof function tests the end-of-file indicator for the stream pointed to by stream and returns nonzero if and only if the end-of-file indicator is set for stream, otherwise it returns zero.

```
#include <stdio.h>
int feof(FILE *stream);
```

3. The ferror function

- The ferror function tests the error indicator for the stream pointed to by stream and returns nonzero if and only if the error indicator is set for stream, otherwise it returns zero.

```
#include <stdio.h>
int ferror(FILE *stream);
```

4. The perror function

- The perror function maps the error number in the integer expression errno to an error message. It writes a sequence of characters to the standard error stream thus: first, if s is not a null pointer and the character pointed to by s is not the null character, the string pointed to by s followed by a colon (:) and a space; then an appropriate error message string followed by a new-line character. The contents of the error message are the same as those returned by the strerror function with the argument errno, which are implementation defined.

```
#include <stdio.h>
void perror(const char *s);
```

8.7 COMMAND LINE ARGUMENTS

- File names may be supplied as argument to the main() at the time of invoking the program, these argument are known as Command Line Argument.
- These arguments are called as command line argument because they are passed from the command line during run time.
- The main() function look like this:

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

- In above the function main( ) can have two arguments, named argc (Argument counter) and argv (Argument vector), where argv is an array of pointers to strings and argc is an integer whose value is equal to the number of strings to which argv points.
- When the program is executed, the strings on the command line are passed to main( ).
- For example, `filecpy source.c destination.c`
- Here, argc contain:

  argv[0] → base address of string filecpy.

  argv[1] → base address of string "source.c".

  argc[2] → base address of string "destination.c".

Program 8.9: Program to copy file using command line argument.

```
#include<iostream.h>
#include<fstream.h>
int main (int argc, char * argv[])
{
   if (argc ! = 3)
   {
      cout<<"Bad command or file name";
      exit(1);
   }
   char ch;
   ifstream fin;
   fin.open (argc[1]);
```

```
    if(fin.fail())
    {
       cout<<"could not open";
       exit(1);
    }
    ofstream fout;
    fout.open (argv[2]);
    if(fout.fail())
    {
       cout<<"could not open destination";
       exit(1);
    }
    while(fin)
    {
       fin.get(ch);
       fout.put(ch);
    }
return 0;
}
```

Output:

```
This is file copy program        //source.c
This is file copy program        //destination.c
```

Program 8.10: Counting the number of vowels in the text file. [S - 18]

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
void main()
{
   int count;
   ifstream in;
   in.open ("xyz.txt")
   ofstream out;
   out.open ("x.text")
   while (!in.eof())
   {
      char ch = (char) in.get();
      if (ch=='a' ||  ch=='e' || ch=='i'||  ch='=o' ||  ch=='u')
      count ++;
   }
   cout<<"no. of vowels ="<<count;
   in.close()'
}
```

Output:

```
This is program to                    //input file
Count number of vowels
No. of vowels  =  12                  //on screen
```

Program 8.11: Write a program to count number of characters, lines and words in a file.

[S - 19]

```
#include<iostream.h>
#include<fstream.h>
#include<stdlib.h>
void main()
{
   char ch;
   int noc = 0; nol = 0; now = 0;
   ifstream myfile;
   myfile.open("countf.dat");
   while(1)
   {
      myfile.get(ch);
      if(ch==eof)
      break;
      noc++;
      if((ch=='\b') || (ch=='\t'))
      now++;
      if(ch=='\n')
      nol++;
   }
   cout<<"no. of characters = "<<noc<<endl;
   cout<<"no. of words = "<<now<<endl;
   cout<<"no. of lines = "<<nol<<endl;
}
```

Output:

```
This is C++ program //input file of counting characters words and lines
no. of characters    =  36
no. of words         =  9
no. of lines         =  2
```

Program 8.12: A program to perform copy from one text file to another text file.

[W - 18, S - 19]

```
#include<fstream.h>
void main()
{
   char source[50]; desti[50];
   char ch;
   cout<<"enter source file:"<<endl;
   cin>>source;
   cout<<"Enter destination file:"<<endl;
   cin>>desti;
   ifstream in(source);
   if(!in)
   {
      cerr<<"error opening file";
      exit (0);
   }
   ofstream out (disti);
   while (in)
   {
       in.get(ch);
       out.put(ch);
   }
}
```

Output:

```
This is file copy program        //input file
This is file copy program        //output file
```

Summary

- A file is a collection of related or associated data on a particular area of the storage device. Text files and binary files are the two different types of files.
- A data of a file is stored in the form of readable and printable characters then the file is known as text file. A file contains non-readable characters in binary code then the file is called binary file.
- The data in a text file is stored in the form of characters, i.e. in ASCII code. In binary files the data is stored in terms of a sequence of bytes, i.e., in 0's and 1's.
- File handling concept in C++ language is used for store a data permanently in computer. Using file handling we can store the data in secondary memory (hard disk).
- A stream is general name given to a flow of data.

- The stream that supplies data to the program is known as the input stream and the one that receives data from the program is known as the output stream. The input stream reads data from the file and the output stream writes data to the file. The third type of the stream takes input as well as gives output to the file.
- The C++ I/O system contains classes such as ifstream, ofstream and fstream to deal with file handling. These classes are derived from fstream base class and a declared in a header file iostream.
- The fstream class inherits all the functions from istream and ostream classes through iostream class defined inside iostream.h file. Also fstream inherits from the ios class, which is also the base class for istream and ostream classes.
- The ifstream class is derived from the istream class. Therefore, all the functions of ifstream class can operate on fstream class.
- A file can be opened in two ways by using the constructor function of a class and using the member function open() of the class.
- The function open() does the following:
  - It prepares the file for input or output operation.
  - It places the file pointer at the appropriate location in the file depending upon the mode in which file has been opened.
- The close() function closes all the opened stream files.
- The purpose of the close() function is:
  - It disconnects the stream with the file on the storage device.
  - No input or output operation can be performed on a file after it has been closed.
- The eof() function is used to check whether the end of a file character is reached or not. When the end of the file is reached, it returns non-zero value, otherwise it returns a zero.
- File names may be supplied as arguments to the main() function at the time of invoking the program. These arguments are known as command-line arguments.
- The file stream classes support some predefined functions that navigate the position of the file pointers. The relevant functions are seekg(), seekp(), tellg() and tellp().

Check Your Understanding

1. The stream that supplies data to a program is called ..... stream.
   (a) input (b) output
   (c) I/O (d) none of these
2. The stream that receives data from the program is called ..... stream.
   (a) input (b) output
   (c) I/O (d) none of these

3. Open() function is used to open ..... files which use the same stream objects.
   (a) multiple (b) console
   (c) print (d) none of these
4. ..... function is used to reached end of file.
   (a) exit() (b) close()
   (c) eof() (d) both (a) & (b)
5. The ..... function closes all the opened stream files.
   (a) exit() (b) close()
   (c) eof() (d) both (a) & (c)
6. A stream may be connected to more than one file at a time.
   (a) True (b) False
7. The ios::ate mode allows us to write data anywhere in the file.
   (a) True (b) False

Answers

1. (a)	2. (b)	3. (a)	4. (c)	5. (b)	6. (a)	7. (a)

Practice Questions

Q.I Answer the following questions in short:

1. What is the purpose of input stream and output stream?
2. What are the types of files?
3. Differentiate between seekg( ) and seekp( ).
4. Enlist various file operations.
5. Differentiate between ios::ate and ios::app.

Q.II Answer the following questions:

1. With the help of diagram explain file operations.
2. Explain various classes available for file Operations.
3. Differentiate between opening a file with a Constructor function and opening a file with open( ) function.
4. What is a File Mode? Describe the various file mode options available.
5. Write a C++ program that reads a text file and creates another file that is identical except that every sequence of consecutive blank spaces is replaced by a single space.
6. Write a C++ program that will create a data file containing the list of telephone numbers and name. Use a class object to store each set of data.
7. Write a C++ menu-driven program that will access the file created in exercise 2 and implement the following tasks:
   (a) Find telephone number of a specified person.
   (b) Find name of a specified telephone number.
   (c) Update the telephone number.

8. Write a menu-driven program that will allow us to create a file with following fields or information:
   (a) book-name
   (b) code
   (c) author-name
   (d) price and implement the following tasks
   (e) add a record
   (f) modify a record
   (g) search a record.
9. Write a C++ program to count no. of vowels in a text file.
10. Write a C++ program to count no. of characters and word from text file.

Q.III Define the term:

1. File
2. Stream
3. Input stream
4. Output stream

Previous Exam Questions

April 2018

1. Write seekg( ) function to: **[2 M]**

(i) Move get pointer 10 bytes backward from end of file.

Ans. Refer to Section 8.4.1.

(ii) Move get pointer to start of file.

Ans. Refer to Section 8.4.1.

2. What is file in C++? Explain two methods of opening a file with syntax. **[4 M]**

Ans. Refer to Section 8.1, 8.3.1.

3. Write a C++ program to display number of vowels present in a given file. **[4 M]**

Ans. Refer to Program 8.10.

October 2018

1. Define the following: **[2 M]**

(i) tellg( )

Ans. Refer to Section 8.4.1.

(ii) tellp( )

Ans. Refer to Section 8.4.1.

2. Write a program to append contents of one file to another file. **[4 M]**

Ans. Refer to Program 8.12.

3. Explain functions for error handing during file operation. **[4 M]**

Ans. Refer to Section 8.6.

April 2019

1. Write a C++ program to read contents of a text file and count number of characters, words and lines in a file. **[4 M]**

Ans. Refer to Section 8.11.

2. Explain various errors handling functions used during file operations. **[4 M]**

Ans. Refer to Section 8.6.

3. Write a C++ program to copy the content of one file to another file. **[4 M]**

Ans. Refer to Section 8.12.

■■■

9... Templates

Objectives...

- To study Class Template and Class Template with multiple parameters.
- To learn about Function Template and Function Template with multiple parameters.
- To understand Exception Handling.

9.1 INTRODUCTION

- Templates are a feature of the C++ programming language that allows functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one.
- Templates are a way of making your classes more abstract by letting you define the behavior of the class without actually knowing what datatype will be handled by the operations of the class.
- Templates can be used in conjunction with abstract datatypes in order to allow them to handle any type of data.
- For example, you could make a templated stack class that can handle a stack of any datatype, rather than having to create a stack class for every different datatype for which you want the stack to function.
- The ability to have a single class that can handle several different datatypes means the code is easier to maintain, and it makes classes more reusable.
- **The basic syntax for declaring a templated class is as follows:**

```
template <class a_type> class a_class {...};
```

Where,

'class': The keyword 'class' means that the identifier.

a_type: It will stand for a datatype.

- The templates declared for functions are called as **function templates** and the templates declared for classes are called as **class templates**.
- C++ templates allows to reuse the software components such as functions, classes etc. and also make the source code compact.

- A function template behaves like a function except that the template can have arguments of many different types.
- There are two kinds of templates one is function template and another is class template.
- A class template provides a specification for generating classes based on parameters. Class templates are commonly used to implement containers.
- A class template is instantiated by passing a given set of types to it as template arguments.

9.2 CLASS TEMPLATES [S - 18, W - 18]

- Like function templates, class templates are also used to declare to operate on different data types.
- C++ class templates, creates a class which contain one or more generic data types.
- **The syntax for creating class template is:**

```
template <class T>
class classname
{
   -------------------
   -------------------
}
```

Here, class data items and functions arguments are of template type.

- **For example:**

```
template<class T>
class arr
{
   T a[10];  //This is used to change to any data type
   int i;
   public:
      arr ();
      void add (const T &ele);
      T remove (void);
      void show();
};
```

- Normally, without template we declare class of char array as:

```
class arr
{
   char a[10];
   int i;
   public:
      arr();
      void add (const char & ele);
      char remove (void);
      void show();
}
```

- Here, we can use any data type instead of **char** and writing again one or more classes for respective data type to declare an array. If we write **int** then it becomes array of integer. Drawback is we are writing more number of classes. So we can have class template as shown in above example.
- In the main( ), we can invoke the class template using object as:

```
classname <data type> objectname;
```

Program 9.1: Program for class template.

```
#include <iostream>
using namespace std;
template <class T1>
class mypair
{
   T1 Y, Z;
   public:
      mypair (T1 first, T1 second)
      {
         Y=first;
         Z=second;
      }
      T1 getmax ();
};
template <class T1>
T1 mypair<T1>::getmax ()
{
   T1 retval;
   retval = Y>Z? Y: Z;
   return retval;
}
int main ()
{
   mypair <int> myobject (100, 75);
   cout << myobject.getmax();
   return 0;
}
```

Output:

```
100
```

9.2.1 Member Function Templates

- We can define member function of a template class outside the class by the following syntax:

```
template<class template type>
returntype classname <tempdata type>::function_name
(arguments)
{
   ------------
};
```

Program 9.2: Program for member function template.

```
//Use of class template to multiply two numbers of different data types
#include<iostream.h>
template<class T>
class mult
{
   T result, a, b;
   public:
      void getdata();
      void mresult();
};
template<class T>
void mult <T>::getdata()
{
   cout<<"Enter two values";
   cin>>a>>b;
}
template<class T>
void mult <T>:: mresult()
{
   result = a * b;
   cout<<"the multiplication is: "<<result<<endl;
}
void main()
{
   mult<int>01;
   mult<float>02;
   mult<long>03;
   01.getdata();
   01.mresult();
```

```
        02.getdata();
        02.mresult();
        03.getdata();
        03.mresult();
    }
```

Output:

```
Enter two values
10        20
the multiplication is: 200
Enter two values
10.5      1.5
the multiplication is: 15.75
Enter two values
3276      25
the multiplication is: 81900
```

9.2.2 Class Templates with Multiple Parameters

- The declaration of template classes with multiple parameters is similar to the function template with multiple parameters. However, all the parameters need not be of template type. We can have any type of arguments. The syntax of class template with multiple parameters is shown below.

Syntax:

```
template<class T1, class T2, ……>
class classname
{
    ……
    ……    //body of class
};
```

Program 9.3: Use of multiple arguments in class template.

```
#include <iostream>
using namespace std;
template <class T>
T GetMax (T a, T b)
{
  T result;
  result = (a>b)? a: b;
  return (result);
}
```

```
int main ()
{
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax<int>(i,j);
  n=GetMax<long>(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

Output:

```
6
10
```

9.3 FUNCTION TEMPLATES [W-18]

- C++ provides a facility for function templates that allow you to write one function which is act as template for a number of functions which are performing similar tasks.
- A function template will not use actual type of arguments but it uses generic type as a placeholder implicit when we make a call to a function. For each version of data type, the compiler generates a copy of that function. For example: when integer version is called, the compiler generates **"add"** function with integer data type and so on.
- **The general syntax is:**

```
template <class template_data_type>
return_type func_name (arguments)              //template function
{
   ----------
}
```

- The function template is preceded by keyword **template** and a list of template type arguments, (generic data types). The template function uses variables whose data types are known only when a call is made.

Program 9.4: Write a function template to find the biggest of two numbers. [W-18]

```
#include<iostream.h>
#include<conio.h>
template<class T>
T big (T a, T b)
{
   if (a>b)
   return (a);
   else return (b);
}
```

```
int main()
{
   cout<<"\n bigger no. is:"<<big(20, 15)<<endl;
   cout<<"\n bigger no. is:"<<big(10.5, 20.5)<<endl;
   cout<<"\n bigger is:"<<big('A', 'M')<<endl;
}
```

Output:

```
bigger no. is: 20
bigger no. is: 20.5
bigger is: M
```

- In the above Program 9.4, the function big( ) is function template having T is a generic type which is used as a placeholder. The **big( )** is called using three different data types int, float and char. The compiler creates three version of big( ), one for each data type.

Program 9.5: Swapping using function template.

```
#include<iostream>
using namespace std;
template<class X>
void Swap(X &a, X &b)          //by reference
{
   X temp = a;
   a = b;
   b = temp;
}
int main()
{
   int x, y;
   cout<<"Enter x & y before swapping"<<endl;
   cin>>x>>y;
   Swap(x, y);
   cout<<"after swapping:"<<x<<" "<<y;
   char c1, c2;
   cin>>c1>>c2;
   Swap (c1, c2);
   cout<<"after swapping"<<c1<<" "<<c2;
}
```

Output:

```
Enter x and y before swapping
10    20
after swapping:  20    10
```

9.3.1 Generic Function

- Using a generic function a single general procedure can be applied to a wide range of data.
- A genetic function defines a general set of operations that will be applied to various types of data.
- A generic function is created using the template keyword. The general **syntax** of a template function definition is:

```
template<class T>
ret_type func_name(para_list)
{
    ..........
    ..........    //body of function
}
```

- A specific instance of a generic function is called as a generated function.
- When we only need a type parameter for a specific function rather than an entire class, it is possible to use a generic function. A function template is a generic function description i.e., it defines a function in terms of a generic type for which a specific type, such as int.
- The following program creates a generic function that swaps the values of the two variables with which it is called.

Program 9.6: Program for generic function.

```
#include<iostream>
using namespace std;
template<class Y> void Swap (Y &m, Y &n)
{
   Y temp;
   temp = m;
   m = n;
   n = temp;
}
int main()
{
   int i = 20, j = 30;
   double x = 20.1, y = 33.3;
   char m = 'x', n = 'z';
   cout<<"Original i, j: "<< i <<' '<< j <<'\n';
   cout<<"Original x, y: "<< x <<' '<< y <<'\n';
   cout<<"Original m, n: "<< m <<' '<< n <<'\n';
   Swap(i, j);
```

```
    Swap(x, y);
    Swap(m, n);
    cout<<"Swapped i, j: "<< i <<' '<< j <<'\n';
    cout<<"Swapped x, y: "<< x <<' '<< y <<'\n';
    cout<<"Swapped m, n: "<< m <<' '<< n <<'\n';
    return 0;
}
```

Output:

```
Original i, j: 20 30
Original x, y: 20.1 33.3
Original m, n: x z
Swapped i, j: 30 20
Swapped x, y: 33.3 20.1
Swapped m, n: z x
```

9.3.2 Function Templates with Multiple Parameters

- We can use more than one generic class or generic data type. The syntax for function templates or generic functions with multiple parameter is shown below:

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

The program 9.7 shows the two generic types functions.

Program 9.7: Program for two generic types' functions.

```
#include<iostream>
using namespace std;
template<class T1, class T2>
void display (T1 a, T2 b)
{
    cout<<a<<"  "<<b<<endl;
}
int main()
{
    display (10, 'A');
    display(10.5, 3456);
    return 0;
}
```

Output:

```
10 A
10.5   3456
```

- In this example, the placeholder types T1 and T2 are replaced by the compiler with the data types int, char, float and double, when the compiler generates the specific instances of display().

Note:

1. All template arguments for a function template must be of template type arguments, otherwise, an error will occur.
2. When you create a function template, the compiler generate as many different versions of that function as necessary to handle the various ways that your program calls the function.

9.3.3 Overloaded Function Template

- We can use overloaded function template, in the situation where all possible data types cannot be handled using function templates. A template function may be overloaded either by template functions or ordinary functions of its name. Overloading in such a case is as follows:
  1. Call an ordinary function with same name.
  2. Call a template function with the same name.
- The program 9.8 shows how a template function is overloaded with an explicit function.

Program 9.8: Overloaded function template.

```
#include<iostream.h>
#include<string.h>
template<class T>
void display(T a)
{
   cout<<"Template display ="<<a<<endl;
}
void display(int a)          //overloads the template
{cout<<"Explicit display = "<<a<<endl; }
int main()
{
   display(50);              //overload function display()
   display('A');             //uses template display()
   display(40.5);            //uses template display()
   return 0;
}
```

Output:

```
Explicit display = 50
Template display = A
Template display = 40.5
```

Program 9.9: To find biggest of two strings i.e. big ("A", "B") then compiler give error message. This problem we can solve using overloaded function template.

```
//overloaded function template
#include <iostream>
#include<cstring>
using namespace std;
char *big (char *x, char *y)
{
   if (strcmp (x, y)>0)
   return x;
   else return y;
}
template <class T>
T big (T x, T y)
{
   if(x > y)
   return x;
   else return y;
}
int main()
{
   cout<<"Bigger no. is:"<<big (10, 20)<<endl;
   cout<<"Bigger is:"<<big ("A", "B");
}
```

Output:

```
Bigger no. is: 20
Bigger is: B
```

9.4 INTRODUCTION OF EXCEPTION HANDLING [S - 18, W - 18]

- When we work with program or compile the program, we normally get logic errors or syntactic errors. We can detect these errors by debugging. We often come across the errors other than logic and syntactic errors. They are known as exceptions. Exceptions are run-time errors that a program encounters while executing. For example, Array out of bounds, divide by zero.
- C++ provides a built-in error handling mechanism that is called **Exception Handling.** Using exception handling you can more easily manage and respond to run-time errors.

- The purpose of exception handling mechanism is to detect the exception in the following way:
  1. Find the run-time problem i.e. exception.
  2. Inform that an error has occurred by throwing the exception.
  3. Get the error information from compiler i.e. catch the exception.
  4. Handle the exception in catch block.

9.4.1 Exception Handling [S - 18]

- C++ exception handling mechanism uses three keywords: **try, catch** and **throw**.
  - **try** block contains program statements that we want to monitor for exceptions.
  - When an exception is detected it is thrown using a **throw** statement in the try block.
  - The exception is caught using **catch.** Typically, a catch(...) block is used to log errors and perform special cleanup before program execution is stopped.
  - Any exception must be caught by a catch statement that immediately follows the try statement that throws the exception.
- **The general form of try and catch is given below:**

```
try
{
   ..........                    //try block
   throw exception               //detects and throws exception
   ..........
}
catch(type1 arg)
{
   //catch block
}
catch(type2 arg)
{
   //catch block
}
   .
   .
   .
catch(typeN arg)
{
   //catch block
}
```

9.4.2 Simple try-catch Mechanism

- Let us first discuss try with only one catch block. This simple try-catch mechanism is illustrate in Program 9.10.

Program 9.10: Program for try-catch mechanism.

```
#include<iostream.h>
//using namespace std;
int main()
{
   cout<<"start\n";
   try
   {
      cout<<"try block \n";
      throw 10;        //throw an error
      cout<<"This will not work";
   }
   catch(int i)
   {
      cout<<"exception caught, Number is:";
      cout<<i<<"\n";
   }
      cout <<"end";
      return 0
}
```

Output:

```
start
try block
exception caught, Number is:10
end
```

- In this above Program 9.10:
  - In try block, only two statements will execute, the first cout statement and second throw. Once, the exception is thrown, control passes to the catch block and try block is terminated.
  - When the control passes to catch block, the statements in the catch block are executed.
  - Here, exception is integer exception.

Program 9.11: Simple program for try-catch statement.

```
//simple try-catch
#include<iostream>
using namespace std;
```

```
int main()
{
   int a, b;
   cin>>a>>b;
   int s = a - b;
   try
   {
      if(s! = 0)
      {
         cout<<"Result is="<<a/s<<'\n';
      }
   else
      {
         throw(s);    //throws integer exception
      }
   }
   catch(int i)
   {
      cout<<"exception caught:s="<<s<<endl;
   }
   return 0;
}
```

Output:

```
15    10
Result = 3
10    10
Exception caught: s = 0
```

- This program detects a division by zero error in try block. The program will not abort due to catch block.
- The exception is thrown using the object s.
- Catch statement contains integer type argument, catches the exception.
- If we change the argument in catch statement as **double**, then the exception will not be caught and abnormal termination occur.

9.4.3 Throwing the Exception from Function

- An exception can be thrown from a statement that is outside the try block but within a function that is called from within the try block.

Program 9.12: Program for throwing an exception from a function outside the try block

```
#include<iostream.h>
//using namespace std;
```

```
void funct(int t)
{
   cout<<"Inside function:"<<t<<'\n';
   if(t) throw t;
}
int main()
{
   cout<<"start\n";
   try
   {
   cout<<"Inside try \n";
      funct(0);
      funct(1);
      funct(2);
   }
}
catch(int i)
{
   cout<<"exception caught:"<<i<<"\n";
}
   cout<<"end";
   return 0;
}
```

Output:

```
start
inside try
inside function:0
inside function:1
exception caught:1
end
```

9.4.4 Try-Catch inside a Function

- When try block is inside a function, then each time the function is entered, the exception handling relative to that function is reset.

Program 9.13: Program for try catch inside a function.

```
#include<iostream>
//using namespace std;
void func(int x)
{
```

```
    try
    {
       if(x) throw x;
    }
    catch(int i)
    {
    cout<<"exception caught:"<< i <<"\n";
    }
}
int main()
{
    cout<<"start"<<"\n";
    func(1);                    //for every function call
    func(2);                    //exception handling is reset
    func(0);
    func(3);
    return 0;
}
```

Output:

```
start
exception caught: 1
exception caught: 2
exception caught: 3
```

9.4.5 Multiple Catch Statements

- We can have more than one catch associated with a try. When the exception is thrown, the exception handlers are searched in order for an appropriate match. When the match is found that catch block is executed. When no match is found, the program is terminated.

Program 9.14: Program for multiple catch statements.

```
#include<iostream.h>
//using namespace std;
void func(int x)
{
    try
    {
       if(x==1) throw x;            //int
       else
       if(x==0) throw 'x';          //char
       else
       if(x==-1) throw 1.5;         //double
       cout<<"try ends:\n";
    }
```

```
    catch(char ch)
    { cout<<"caught character:\n"<<ch<<'\n'; }
    catch(int i)
    { cout<<"caught integer:\n"<<i<<"\n"; }
    catch(double d)
    { cout<<"caught double:\n"<<d<<"\n"; }
    cout<<"end of catch block";
}
int main()
{ cout<<"start \n";
    func(1);
    func(0);
    func(-1);
    func(2);
    return0;
}
```

Output:

```
start
caught integer:1
end of catch blockcaught character:x
end of catch blockcaught double:1.5
end of catch blocktry ends: end of catch block
end of catch block
```

- In this program, when x = 1, it throws integer exception. This matches with the second catch block, therefore the second catch will executed. When x = 0, it throws character exception, therefore first catch block is executed. When x = –1, it throws double exception hence 3rd catch block is executed. Finally, when x = 2, program terminates (no. exception thrown).

9.4.6 Catch all Exceptions

- Sometimes, we want an exception handler to catch all exceptions instead of just a certain type. We use the following **syntax**:

```
catch (….)
{
    //process all exceptions
}
```

Here, catch(...) matches any type of data.

Program 9.15: Program for catch all exceptions.

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

```
void func(int x)
{
   try
   {
      if(x==1)throw x;        //int
      if(x==0)throw 'x';      //char
      if(x==-1) throw 1.5;    //double
   }
   catch(…)            //catch all exceptions
   {
      cout<<"caught exception \n";
   }
}
int main()
{
   cout<<"start \n";
   func(0);
   func(1);
   func(-1);
   return 0;
}
```

Output:

```
start
caught exception
caught exception
caught exception
```

- Using catch(...) is a good way to catch all exceptions that you do not want to handle explicitly. In Program 9.15 if we add one more catch block before catch(...) block as:

```
catch (int i)
{
   cout<<"caught:"<< i<<'\n';
}
```

- Then integer exception is also handle. Other two exceptions are handle by using catch(...) and the output is:

```
start
caught exception
caught 1
caught exception
```

9.4.7 Rethrowing an Exception

- Rethrowing an exception means to allow multiple handlers access to the exception. An exception can only be rethrown from within a catch block. When we rethrow an exception, it will not be recaught by the same catch block or any other catch in that group. Rather, it will be caught by an appropriate catch in the outer try-catch block.

Program 9.16: Program for rethrowing an exception.

```
#include<iostream.h>
//using namespace std;
void div (double x, double y)
{
   try
   {
      if(y==0.0) throw y;          //double
      else
      cout<<"Division="<<x/y<<"\n";
   }
   catch(double)
   {
      cout<<"caught double \n";
      throw;                  //Rethrowing exception double
   }
   cout<<"function end \n";
}
//end of function
int main()
{
   cout<<"start\n";
   try
   {
      div(8.6, 2.0);
      div(10.5, 0.0);
   }
   catch(double)
   {
      cout<<"caught double in main\n";
   }
   cout<<"end";
   return 0;
}
```

Output:

```
start
division = 4.3
function end
caught double
caught double in main
end
```

Program 9.17: Write a template program to sort integer and float array elements of size five

```
#include<iostream>
using namespace std;
int n;
#define size 10
template<class T>
void sel(T A[size])
{
    int i,j,min;
    T temp;
    for(i=0;i<n-1;i++)
    {
        min=i;
        for(j=i+1;j<n;j++)
        {
            if(A[j]<A[min])
            min=j;
        }
        temp=A[i];
        A[i]=A[min];
        A[min]=temp;
    }
    cout<<"\nSorted array:";
    for(i=0;i<n;i++)
    {
        cout<<" "<<A[i];
    }
}
```

```
int main()
{
    int A[size];
    float B[size];
    int i;

    cout<<"\nEnter total no of int elements:";
    cin>>n;
    cout<<"\nEnter int elements:";
    for(i=0;i<n;i++)
    {
        cin>>A[i];
    }
    sel(A);

    cout<<"\nEnter total no of float elements:";
    cin>>n;
    cout<<"\nEnter float elements:";
    for(i=0;i<n;i++)
    {
        cin>>B[i];
    }
    sel(B);
}
```

Output:

```
Enter total no of int elements:5
Enter int elements:5 6 3 8 1
Sorted array: 1 3 5 6 8
Enter total no of float elements:5
Enter float elements: 1.1 5.8 9.03 10.65 5.66
Sorted array: 1.1 5.66 5.8 9.03 10.65
```

Summary

- Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.
- A template is a blueprint or formula for creating a generic class or a function.
- Template in C++ allows us to define generic classes and functions and thus provides support for generic programming.
- Generic programming is an approach where we can define a single function or a class which can work with any data types.

- A template specifies how to construct an individual class or function by providing a blueprint description of classes or functions within the template.
- A template in C++ allows the construction of a family of template functions and classes to perform the same operation on different data types. The templates declared for functions are called function templates and those declared for classes are called class templates. They perform appropriate operations depending on the data type of the parameters passed to them.
- A generic function defines a general set of operations that will be applied to various types of data. The type of data that the function will operate upon is passed to it as a parameter. Through a generic function, a single general procedure can be applied to a wide range of data.
- We can use multiple parameters in both the class templates and function templates.
- Like other functions, template functions can be overloaded.

Check Your Understanding

1. Class template can have ..... parameters.
   (a) any type (b) class type
   (c) both (d) none
2. The actual source code for implementing a template function is created when .....
   (a) the function is actually executed
   (b) the declaration of the function appears
   (c) the definition of the function appears
   (d) the function is invoked.
3. The template argument is preceded by the keyword .....
   (a) class (b) T
   (c) datatype (d) all
4. What is the correct syntax of defining function template/template functions?
   (a) Template <class T> void(T a){cout<<a;}
   (b) Template <class T> void(T a){cout<<a;}
   (c) Template <T> void(T a){cout<<a;}
   (d) Template <T> void(T a){cout<<a;}
5. In how many ways templates concept can be used?
   (a) 1 (b) 2
   (c) 3 (d) 4
6. Function templates require more memory space than normal function.
   (a) True (b) False
7. Templates are processed by the compiler.
   (a) True (b) False

8. Template classes can work with different data types.
   (a) True (b) False

Answers

1. (c)	2. (d)	3. (a)	4. (a)	5. (b)	6. (b)	7. (a)	8. (a)

Practice Questions

Q.I Answer the following Questions in short:

1. A template can be considered as a kind of macro. Then, what is the difference between them?
2. Distinguish between Overloaded Functions and Function Templates.
3. Distinguish between the terms Class Template and Template Class.
4. State which of the following definitions are illegal:

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

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

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

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

5. What is meant by Template? How to declare it?
6. Which keyword used for exception handling?

Q.II Answer the following questions:

1. What is a Function Template? Write a function template for finding the largest number in a given array. The array parameter must be of generic data types.
2. Explain how the Compiler Processes call to a Function Template.
3. What is Class Template? Explain the syntax of a class template with suitable example.
4. Write a template based program for adding objects of the vector class.
5. Explain simple try-catch mechanism.
6. Explain Exception Handling.
7. Write a function template to calculate area of circle.
8. Write short note on: template.
9. With the help of syntax and example describe Class Template.
10. Write short note on: Overloading a function template.

11. Distinguish Class and Function Templates.
12. Write class and function template program for finding minimum value contained in an array.

Q.III Define the term:

1. Template
2. Exception Handling
3. Class Template
4. Function Template

Previous Exam Questions

April 2018

1. Explain in which circumstances catch(....) statement would be used? [2 M]

Ans. Refer to Section 9.4.1.

2. What is exception? Explain how exception is handled in C++. [4 M]

Ans. Refer to Section 9.4.

3. Write a note on class templates. [4 M]

Ans. Refer to Section 9.2.

October 2018

1. Define: [2 M]

(i) class template

Ans. Refer to Section 9.2.

(ii) function template.

Ans. Refer to Section 9.3.

2. Write a C++ program to find maximum and minimum of two integers and two float numbers by using function template. [4 M]

Ans. Refer to Program 9.4.

3. Write a note on exception handling. [4 M]

Ans. Refer to Section 9.4.

April 2019

1. Write a template program to sort integer and float array elements of size five. [4 M]

Ans. Refer to Program 9.17.

■■■